

每周学习进展阶段汇报

---CS131 Stanford University 计算机视觉基础课程

汇报人：胡小婉

时间段：2018 年 1 月 22 日(周一)至 2018 年 1 月 28 日(周日)

Lecture #5: Edge detection

Lecture #06: Edge Detection

1. 边缘检测.....	5
1.1 基本概念:.....	5
1.1.1 图像 k 阶导数:.....	5
1.1.2 Pascal 三角形:.....	5
1.2 Canny 边缘检测.....	6
1.2.1 算法原理.....	6
1.2.2 基本步骤.....	6
1.2.3 平滑处理.....	7
1.2.4 梯度求导.....	7
1.2.5 非极大值抑制.....	8
1.2.6 阈值检测.....	10
1.2.7 双阈值边缘连接.....	10
Tip: 图像的二值化和灰度化.....	11
2. Hough 变换.....	11
2.1 算法原理.....	11
2.2 基本步骤.....	12
2.2.1 直线检测.....	12
2.2.2 极坐标转换.....	13
2.2.3 代码思路.....	14
3. 作业题(hw2).....	15
3.1 Smoothing.....	15
3.2 Finding gradients.....	16
3.3 Non-maximum suppression.....	18
3.4 Double thresholding.....	18
Tips: 用到的一些函数.....	18
3.5 Edge tracking by hysteresis.....	19
3.6 车道检测应用程序.....	20
Tip: 用到的一些函数.....	21

Lecture #6-7: Features and Fitting/Feature Descriptors

Lecture #8: Feature Descriptors and Resizing

1. Panorama Stitching(全景拼接).....	22
2. Harris 角点检测.....	23
2.1 基本原理及思想.....	23

2.2 基本步骤.....	25
2.3 A simple descriptor.....	25
2.3.1 对图像归一化.....	25
2.3.2 match_descriptors.....	25
2.4 特征匹配中的欧氏距离.....	25
Tip:Scipy.spatial.distance.cdist.....	26
2.5 最小二乘法转换矩阵.....	27
Tip1: numpy.linalg.lstsq.....	27
Tip2: np.pad().....	27
3. 图像仿射变换及图像扭曲(Image Warping).....	28
4. RANSAC(Random Sample Consensus)去噪.....	30
4.1. 算法原理.....	30
4.2 实现步骤.....	31
4.3 代码思路.....	31
5. Histogram of Oriented Gradients(HOG).....	32
5.1 基本原理.....	32
5.2 实现步骤.....	32
5.2.1 计算图像梯度.....	33
5.2.2 为每个细胞单元构建梯度方向直方图.....	33
5.2.3 block 归一化.....	33
Tips:一些函数.....	34
6. 作业题(hw3).....	34
6.1Harris Corner Detector.....	34
6.2 Describing and Matching Keypoints.....	36
6.2.1 Creating Descriptors.....	36
6.2.2 Matching Descriptors.....	37
6.3 Transformation Estimation.....	38
6.4 全景拼接:.....	39
6.5Histogram of Oriented Gradients (HOG).....	40
6.5.1 特征点匹配.....	40
6.5.2 全景拼接.....	40

Lecture #9: Image Resizing and Segmentation

1. Seam Carving.....	42
1.1 算法原理.....	42
1.2 基本步骤.....	42
1.2.1. 计算图像能量图.....	42
1.2.2 寻找最小能量线.....	43
1.2.3 移除得到的最小能量线.....	43
2. 部分作业题(hw4).....	44

2.1 计算图像能量.....	44
Tip: 导入后首先要做灰度化处理:.....	45
2.2 Compute cost.....	45
2.3 Finding optimal seams.....	46
一些说明:.....	49

Lecture #5: Edge detection

Lecture #06: Edge Detection

1. 边缘检测

1.1 基本概念:

1.1.1 图像 k 阶导数:

一阶导数的 3 种近似: 前向差分, 后向差分, 中心差分(Backward .Forward

Central)

Backward

$$\frac{df}{dx} = f(x) - f(x-1) = f'(x)$$

$$[0, 1, -1]$$

Forward:

$$\frac{df}{dx} = f(x) - f(x+1) = f'(x)$$

$$[-1, 1, 0]$$

Central:

$$\frac{df}{dx} = f(x+1) - f(x-1) = f'(x)$$

$$[1, 0, -1]$$

二阶:

Gradient vector

$$\nabla f(x, y) = \begin{bmatrix} f_x \\ f_y \end{bmatrix}$$

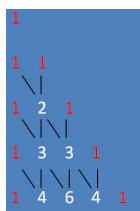
Gradient magnitude

$$|\nabla f(x, y)| = \sqrt{f_x^2 + f_y^2}$$

Gradient direction

$$\theta = \tan^{-1} \left(\frac{\frac{df}{dy}}{\frac{df}{dx}} \right)$$

1.1.2 Pascal 三角形:



由图可得公式： $a[i][j] = a[i-1][j-1] + a[i-1][j]$

一个 k 阶导数核的系数就是 Pascal 三角的第 $k+1$ 行加上间隔反复的系数

PS; projective 投影 euclidean. 欧式 occlusion 遮挡 clutter 杂波 Pascal 三角

1.2 Canny 边缘检测

1.2.1 算法原理

检测阶跃边缘的基本思想是在图像中找出具有局部最大梯度幅值的像素点.检测阶跃边缘的大部分工作集中在寻找能够用于实际图像的梯度数字逼近.由于实际的图像经过了摄像机光学系统和电路系统(带宽限制)固有的低通滤波器的平滑,因此,图像中的阶跃边缘不是十分陡立.图像也受到摄像机噪声和场景中不希望细节的干扰.

图像梯度逼近必须满足两个要求:①逼近必须能够抑制噪声效应,②必须尽量精确地确定边缘的位置.抑制噪声和边缘精确定位是无法同时得到满足的,也就是说,边缘检测算法通过图像平滑算子去除了噪声,但却增加了边缘定位的不确定性;反过来,若提高边缘检测算子对边缘的敏感性,同时也提高了对噪声的敏感性.有一种线性算子可以在抗噪声干扰和精确定位之间选择一个最佳折衷方案,它就是高斯函数的一阶导数,对应于图像的高斯函数和梯度计算一梯度的数值逼近可用 x 和 y 方向上的一阶偏导的有限差分来表示,高斯平滑和梯度逼近相结合的算子不是旋转对称的,而在边缘方向上是对称的,在垂直边缘的方向上是反对称的(沿梯度方向),这也意味着该算子对最急剧变化方向上的边缘特别敏感,但在沿边缘这一方向上是不敏感的,其作用就像一个平滑算子。

Canny 边缘检测器是高斯函数的一阶导数,是对信噪比与定位之乘积的最优化逼近算子

1.2.2 基本步骤

Canny 边缘检测算法通常都是从高斯模糊开始,到基于双阈值实现边缘连接结束但是在实际工程应用中,考虑到输入图像都是彩色图像,最终边缘连接之后的图像要二值化输出显示,所以完整的 Canny 边缘检测算法实现步骤如下:

In this part, you are going to implement Canny edge detector. The Canny edge detection algorithm can be broken down in to five steps:

1. Smoothing
2. Finding gradients
3. Non-maximum suppression
4. Double thresholding
5. Edge tracking by hysteresis

1. 彩色图像转换为灰度图像
2. 对图像进行高斯模糊
3. 计算图像梯度，根据梯度计算图像边缘幅值与角度
4. 非最大信号压制处理（边缘细化）
5. 双阈值边缘连接处理
6. 二值化图像输出结果

canny 算子是一种求最优边缘检测的一套方法。是一种先平滑再求导的方法。

1.2.3 平滑处理

We first smooth the input image by convolving it with a Gaussian kernel. The equation for a Gaussian kernel of size $(2k + 1) \times (2k + 1)$ is given by:

$$h_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i-k)^2 + (j-k)^2}{2\sigma^2}\right), 0 \leq i, j < 2k + 1$$

对图像用高斯滤波器进行平滑处理。

高斯滤波用于对图像进行减噪，采用邻域加权平均的方法计算每一个像素点的值。

1.2.4 梯度求导

The gradient of a 2D scalar function $I : \mathbb{R}^2 \rightarrow \mathbb{R}$ in Cartesian coordinate is defined by:

$$\nabla I(x, y) = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right],$$

where

$$\begin{aligned} \frac{\partial I(x, y)}{\partial x} &= \lim_{\Delta x \rightarrow 0} \frac{I(x + \Delta x, y) - I(x, y)}{\Delta x} \\ \frac{\partial I(x, y)}{\partial y} &= \lim_{\Delta y \rightarrow 0} \frac{I(x, y + \Delta y) - I(x, y)}{\Delta y} \end{aligned}$$

In case of images, we can approximate the partial derivatives by taking differences at one pixel intervals:

$$\begin{aligned} \frac{\partial I(x, y)}{\partial x} &\approx \frac{I(x + 1, y) - I(x - 1, y)}{2} \\ \frac{\partial I(x, y)}{\partial y} &\approx \frac{I(x, y + 1) - I(x, y - 1)}{2} \end{aligned}$$

Note that the partial derivatives can be computed by convolving the image I with some appropriate kernels D_x and D_y :

$$\begin{aligned} \frac{\partial I}{\partial x} &\approx I * D_x = G_x \\ \frac{\partial I}{\partial y} &\approx I * D_y = G_y \end{aligned}$$

利用一阶差分计算边缘的方向与幅值。

计算图像 X 方向与 Y 方向梯度，根据梯度计算图像边缘幅值与角度大小

高斯模糊的目的主要为了整体降低图像噪声，目的是为了更准确计算图像梯度及边缘幅值。计算图像梯度可以选择算子有 Robot 算子、Sobel 算子、Prewitt 算子等。

注意卷积模板如下方所示：

$$H_1 = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \quad H_2 = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$\varphi_1(m, n) = f(m, n) * H_1(x, y)$$

$$\varphi_2(m, n) = f(m, n) * H_2(m, n)$$

$$\varphi(m, n) = \sqrt{\varphi_1^2(m, n) + \varphi_2^2(m, n)}$$

$$\theta_s = \tan^{-1} \frac{\varphi_2(m, n)}{\varphi_1(m, n)}$$

$$G = \sqrt{G_x^2 + G_y^2}$$

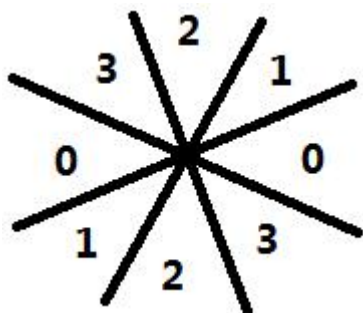
$$\Theta = \arctan\left(\frac{G_y}{G_x}\right)$$

该方法在上方一阶微分边缘检测方法中已有介绍，求法相同。也是为了得到边缘的幅值与方向。

1.2.5 非极大值抑制

仅仅求出全局的梯度方向并不能确定边缘的位置。所以需要对一些梯度值不是最大的点进行抑制，突出真正的边缘。

第二步中，求出的梯度的方向可以是 360° 的，我们将这 360° 的区域划分为 4 个空间，如下图所示。



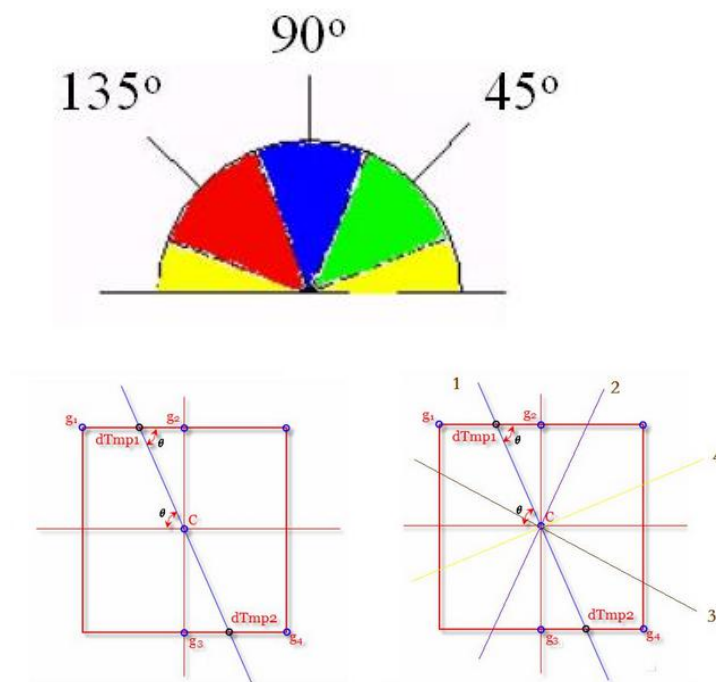
对应到图像中，一个像素点与其相邻的 8 个像素点，也可以划分为 4 个区域。

3	2	1
0		0
1	2	3

现在需要做的是，判断沿着梯度方向的 3 个像素点（中心像素点与另外两个梯度方向区域内的像素点），若中心像素点的梯度值不比其他两个像素点的梯度值大，则将该像素点的灰度值置 0。

非最大信号压制主要目的是实现边缘细化，通过该步处理边缘像素进一步减少。非最大

信号压制主要思想是假设 3x3 的像素区域，中心像素 $P(x,y)$ 根据上一步中计算得到边缘角度值 angle ，可以将角度分为四个离散值 0、45、90、135 分类依据如下：



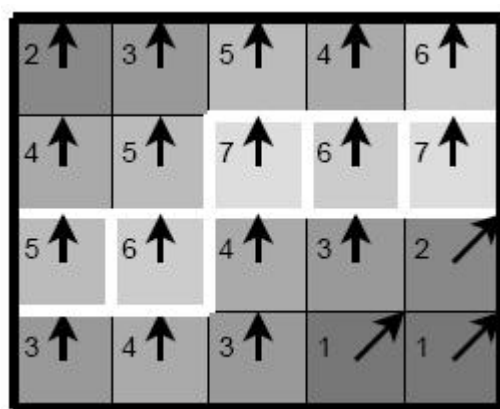
其中黄色区域取值范围为 $0 \sim 22.5$ 与 $157.5 \sim 180$

绿色区域取值范围为 $22.5 \sim 67.5$

蓝色区域取值范围为 $67.5 \sim 112.5$

红色区域取值范围为 $112.5 \sim 157.5$

分别表示上述四个离散角度的取值范围。得到角度之后，比较中心像素角度上相邻两个像素，如果中心像素小于其中任意一个，则舍弃该边缘像素点，否则保留。一个简单的例子如下：



```
def get_neighbors(y, x, H, W):
    """ Return indices of valid neighbors of (y, x)

    Return indices of all the valid neighbors of (y, x) in an array of
    shape (H, W). An index (i, j) of a valid neighbor should satisfy
    the following:
    1. i >= 0 and i < H
    2. j >= 0 and j < W
    3. (i, j) != (y, x)

    Args:
        y, x: location of the pixel
        H, W: size of the image
    Returns:
        neighbors: list of indices of neighboring pixels [(i, j)]
    """
    neighbors = []

    for i in (y - 1, y, y + 1):
        for j in (x - 1, x, x + 1):
            if i >= 0 and i < H and j >= 0 and j < W:
                if (i == y and j == x):
                    continue
                neighbors.append((i, j))

    return neighbors
```

```
# BEGIN YOUR CODE
for i in range(1, H - 1):
    for j in range(1, W - 1):
        if(theta[i][j] == 0 or theta[i][j] == 180 or theta[i][j] == 360):
            m0 = G[i][j]
            m1 = G[i][j - 1]
            m2 = G[i][j + 1]
            if ((m0 < m1) or (m0 < m2)):
                G[i][j] = 0
        if(theta[i][j] == 45 or theta[i][j] == 225):
            m0 = G[i][j]
            m1 = G[i - 1][j + 1]
            m2 = G[i + 1][j - 1]
            if ((m0 < m1) or (m0 < m2)):
                G[i][j] = 0
        if(theta[i][j] == 90 or theta[i][j] == 270):
            m0 = G[i][j]
            m1 = G[i + 1][j]
            m2 = G[i - 1][j]
            if ((m0 < m1) or (m0 < m2)):
                G[i][j] = 0
        if(theta[i][j] == 135 or theta[i][j] == 315):
            m0 = G[i][j]
            m1 = G[i - 1][j - 1]
            m2 = G[i + 1][j + 1]
            if ((m0 < m1) or (m0 < m2)):
                G[i][j] = 0

    for i in range(0, H):
        for j in range(0, W):
            out[i][j] = G[i][j]

# END YOUR CODE
```

1.2.6 阈值检测

非最大信号压制以后，输出的幅值如果直接显示结果可能会少量的非边缘像素被包含到结果中，所以要通过选取阈值进行取舍，传统的基于一个阈值的方法如果选择的阈值较小起不到过滤非边缘的作用，如果选择的阈值过大容易丢失真正的图像边缘，Canny 提出基于双阈值(Fuzzy threshold)方法很好的实现了边缘选取。双阈值选择与边缘连接方法通过假设两个阈值

其中一个为高阈值 TH 另外一个为低阈值 TL 则有

- A. 对于任意边缘像素低于 TL 的则丢弃
- B. 对于任意边缘像素高于 TH 的则保留
- C. 对于任意边缘像素值在 TL 与 TH 之间的，如果能通过边缘连接到一个像素大于 TH 而且边缘所有像素大于最小阈值 TL 的则保留，否则丢弃

1.2.7 双阈值边缘连接

对上一部处理完成的图像作用两个阈值，将梯度值小于阈值的点的灰度值直接置为 0，得到两个新的图像，记作图 1 和图 2。图 2 因为作用的阈值较大，去除了大部分的噪声，同时也有可能去除了有用的边缘信息。图 1 则保留了较多的边缘信息，我们利用图 1 来对图 2 进行补充。

补充的方法：

对图 2 进行扫描，遇到第一个非零灰度的像素点 $p(x, y)$ 时，并以 p 点开始向周边不断扫描连续的非 0 灰度像素点（得到边缘的轮廓线）。当找到了轮廓线的终点（即扫描不到连续的非零灰度像素点的点，记为 q 点），每当找到这样的 q 点，就在图 1 中找到同样位置的像素点，考察其邻近的 8 个像素点，是否有非 0 灰度像素点。如果有，则在图 2 中将该点灰度置为非 0，并作为 p 点，重复整个扫描过程。

当整个过程都无法再继续的时候，我们就认为已经找到了一条边缘的轮廓线。不停重复该过程，直到找到所有的边缘的轮廓线为止。

Tip: 图像的二值化和灰度化

图像的二值化是将图像上的像素点的灰度值设置为 0 或 255，也就是将整个图像呈现出明显的黑白效果。

将 256 个亮度等级的灰度图像通过适当的阈值选取而获得仍然可以反映图像整体和局部特征的二值化图像。在数字图像处理中，二值图像占有非常重要的地位，首先，图像的二值化有利于图像的进一步处理，使图像变得简单，而且数据量减小，能凸显出感兴趣的目标的轮廓。其次，要进行二值图像的处理与分析，首先要把灰度图像二值化，得到二值化图像。

所有灰度大于或等于阈值的像素被判定为属于特定物体，其灰度值为 255 表示，否则这些像素点被排除在物体区域以外，灰度值为 0，表示背景或者例外的物体区域

2. Hough 变换

2.1 算法原理

图像中所有可能的直线像素的检测，可以通过在图像中进行边缘检测子得到，所有边缘幅值超过某个阈值的像素都可以看作是可能的直线像素。

在最一般的情况下，当我们没有任何有关图像中的直线信息，因此，所有方向的直线可能通过任何边缘像素。而在现在实现中，这些直线的数目是无限的，然而，为了实际目标，只能有限数目的直线方向。直线的可能方向定义了参数 K 的一个离散化，因此参数 q 也被采样为有限数目的值。所以参数空间不是连续的，而是被表示为矩形单元，称之为累计数组 (accumulator array) A ，它的元素是累计单元 (Accumulator cells) $A(k, q)$ 。对于每个边缘元素，确定其参数 k 和 q 。这些参数表示了通过此像素的允许方向的直线。对于每条这样的直线，直线参数 k 和 q 的值用来增加累计单元 $A(k, q)$ 的值。如果公式 $y=ax+b$ 所表

示的直线出现在图像中， $A(a,b)$ 的值会被增加很多次，而次数等于直线 $y=ax+b$ 作为可能通过某个边缘像素的直线被检测到的数目。

对于任意像素 P ，通过它的直线可能是任何的方向 k ，但是第二个参数 q 受像素 P 图像坐标和方向 k 所约束。因此，存在于图像中的直线会引起图像中适合的累计单元值变大，而通近边缘像素的其他直线，它们不对应于图像中存在的直线，对于每个边缘像素具有不同的参数 k 和 q ，所以对应的累计单元极少被增加。即：图像中存在的直线可以作为累计数组中的高值累计单元被检测出，检测到的直线参数由累计数组的坐标给出，结果是图像中直线的检测被为累计空间中的局部极值的检测

2.2 基本步骤

直线检测算法可以分为以下几个步骤：

1. 边缘检测，例如 使用 Canny 边缘检测器[2]。
2. 将边缘点映射到霍夫空间并存储在累加器中。
3. 解释累加器产生无限长度的线。 解释是通过阈值和可能的其他约束来完成。
4. 将无穷线转换为有限线。

霍夫变换(Hough Transform)是图像处理中的一种特征提取技术，该过程在一个参数空间中通过计算累计结果的局部最大值得到一个符合该特定形状的集合作为霍夫变换结果。

霍夫变换运用两个坐标空间之间的变换将在一个空间中具有相同形状的曲线或直线映射到另一个坐标空间的一个点上形成峰值，从而把检测任意形状的问题转化为统计峰值问题。

2.2.1 直线检测

$$y = a \cdot x + b$$

直线在直角坐标系下可以用 $y=kx+b$ 表示，霍夫变换的主要思想是将该方程的参数和变量交换，即用 x,y 作为已知量 k,b 作为变量坐标，所以直角坐标系下的直线 $y=kx+b$ 在参数空间表示为点 (k,b) ，而一个点 (x_1,y_1) 在直角坐标系下表示为一条直线 $y_1=x_1 k+b$ ，其中 (k,b) 是该直线上的任意点。为了计算方便，我们将参数空间的坐标表示为极坐标下的 γ 和 θ 。因为同一条直线上的点对应的 (γ, θ) 是相同的，因此可以先将图片进行边缘检测，然后对图像上每一个非零像素点，在参数坐标下变换为一条直线，那么在直角坐标下属于同一条直线的点便在参数空间形成多条直线并内交于一点。因此可用该原理进行直线检测。

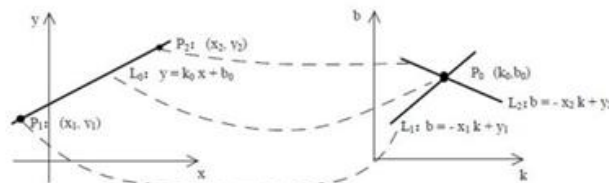
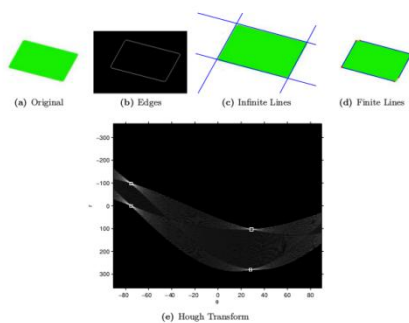


图1 点—线的对偶性



2.2.2 极坐标转换

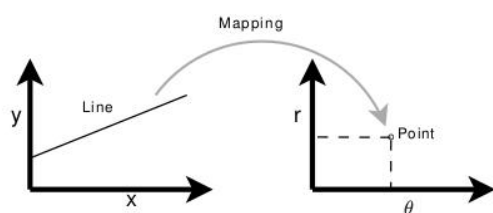


Figure 1: Mapping of one unique line to the Hough space.

我们可以注意到直线的参数方程 $y = kx + q$ 只适合解释 Hough 变换原理，在检测垂直线条和参数的非线性离散化时会遇到困难。如果直线表示成 $s = x \cos \theta + y \sin \theta$ 。Hough 变换就没有这些局限性。直线还是被变换为单个点，因此可用该原理进行直线检测。如下图 2 所示：

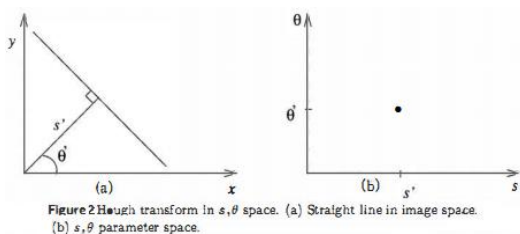
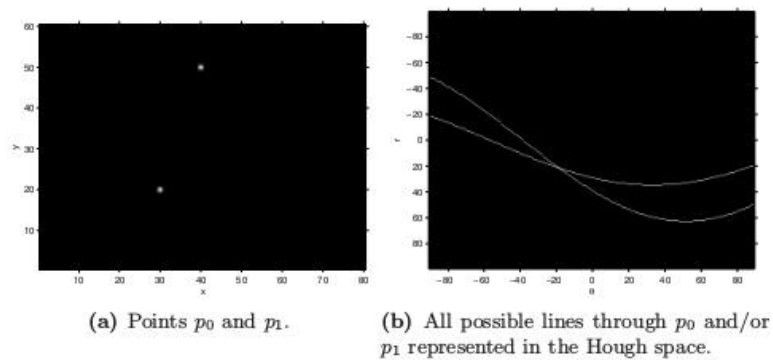


Figure 2: Hough transform in s, θ space. (a) Straight line in image space. (b) s, θ parameter space.

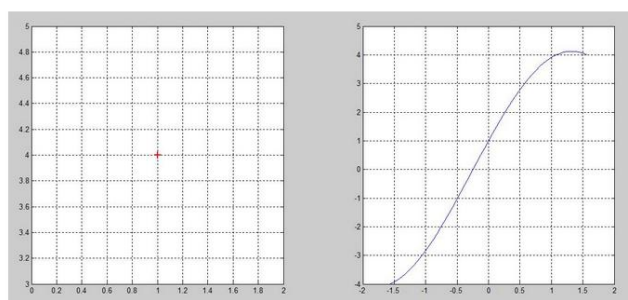
$$r = x \cdot \cos \theta + y \cdot \sin \theta \Leftrightarrow$$

$$y = -\frac{\cos \theta}{\sin \theta} \cdot x + \frac{r}{\sin \theta}$$

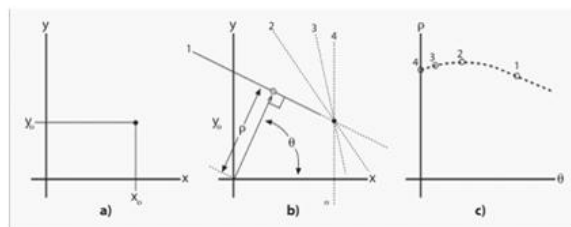
我们要注意到 Hough 变换的重要性质是对图像中直线的残缺部分、噪声以及其它共存的非直线结构不敏感。因为从图像空间到累计空间的变换的 Robustness 引起的，直线残缺的部分只会造成较低的局部极值。



如下图所示，左图直角坐标系中的一个点，对应于右图 $r - \theta$ 空间的一条正弦曲线



2.2.3 代码思路



上图 (a) 所示为原始的图像空间中一个点；(b) 所示为直角坐标系当中为过同一四条直线；(c) 所示为这四条直线在极坐标参数空间可以表示为四个点

为了检测出直角坐标 $X-Y$ 中由点所构成的直线，可以将极坐标 $a-p$ 量化成许多小格。根据直角坐标中每个点的坐标 (x, y) ，在 $a = 0-180^\circ$ 内以小格的步长计算各个 p 值，所得值落在某个小格内，便使该小格的累加计数器加 1。当直角坐标中全部的点都变换后，对小格进行检验，计数值最大的小格，其 (a, p) 值对应于直角坐标中所求直线


```

def hough_transform(img):
    """ Transform points in the input image into Hough space.

    Use the parameterization:
        rho = x * cos(theta) + y * sin(theta)
    to transform a point (x,y) to a sine-like function in Hough space.

    Args:
        img: binary image of shape (H, W)

    Returns:
        accumulator: numpy array of shape (m, n)
        rhos: numpy array of shape (m, )
        thetas: numpy array of shape (n, )
    """
    # Set rho and theta ranges
    w, h = img.shape
    diag_len = int(np.ceil(np.sqrt(w * w + h * h)))
    rhos = np.linspace(-diag_len, diag_len, diag_len * 2.0 + 1)
    thetas = np.deg2rad(np.arange(-90.0, 90.0))
    # Cache some reusable values
    cos_t = np.cos(thetas)
    sin_t = np.sin(thetas)
    num_thetas = len(thetas)

    # Initialize accumulator in the Hough space
    accumulator = np.zeros((2 * diag_len + 1, num_thetas), dtype=np.uint64)
    ys, xs = np.nonzero(img)

    # Transform each point (x, y) in image
    # Find rho corresponding to values in thetas
    # and increment the accumulator in the corresponding coordinate.
    # YOUR CODE HERE
    num_point = len(ys)
    for i in range(0, num_point):
        for m in range(0, num_thetas):
            p = cos_t[m] * xs[i] + sin_t[m] * ys[i] + 1102
            accumulator[p][m] += 1
    # END YOUR CODE

    return accumulator, rhos, thetas

```

3. 作业题(hw2)

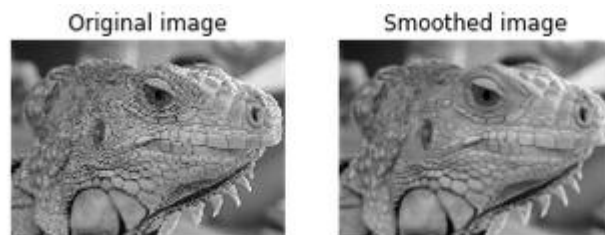
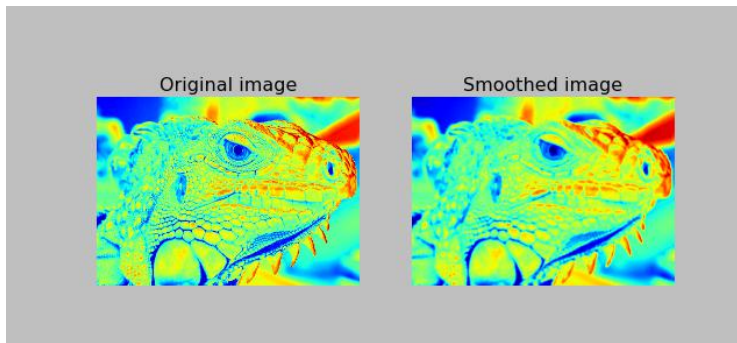
3.1 Smoothing

执行除法出现问题，即出现自动四舍五入的现象导致计算出错，引入函数 from `__future__` import `division`，解决除法问题

We first smooth the input image by convolving it with a Gaussian kernel. The equation for a Gaussian kernel of size $(2k + 1) \times (2k + 1)$ is given by:

$$h_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i-k)^2 + (j-k)^2}{2\sigma^2}\right), 0 \leq i, j < 2k + 1$$

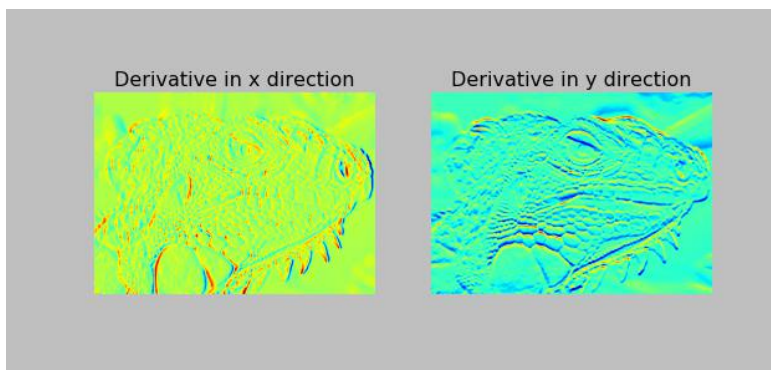
按题目要求得到以下结果：



按题目要求及给定公式：
对 x, y 方向分别求偏导并进行滤波

3.2 Finding gradients

梯度转换



The gradient of a 2D scalar function $I : \mathbb{R}^2 \rightarrow \mathbb{R}$ in Cartesian coordinate is defined by:

$$\nabla I(x, y) = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right],$$

where

$$\frac{\partial I(x, y)}{\partial x} = \lim_{\Delta x \rightarrow 0} \frac{I(x + \Delta x, y) - I(x, y)}{\Delta x}$$

$$\frac{\partial I(x, y)}{\partial y} = \lim_{\Delta y \rightarrow 0} \frac{I(x, y + \Delta y) - I(x, y)}{\Delta y}.$$

In case of images, we can approximate the partial derivatives by taking differences at one pixel intervals:

$$\frac{\partial I(x, y)}{\partial x} \approx \frac{I(x + 1, y) - I(x - 1, y)}{2}$$

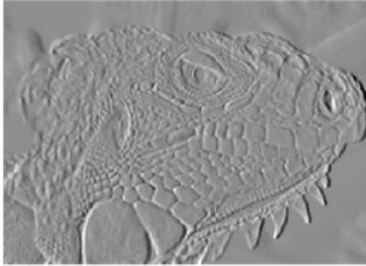
$$\frac{\partial I(x, y)}{\partial y} \approx \frac{I(x, y + 1) - I(x, y - 1)}{2}$$

Note that the partial derivatives can be computed by convolving the image I with some appropriate kernels D_x and D_y :

$$\frac{\partial I}{\partial x} \approx I * D_x = G_x$$

$$\frac{\partial I}{\partial y} \approx I * D_y = G_y$$

Derivative in x direction



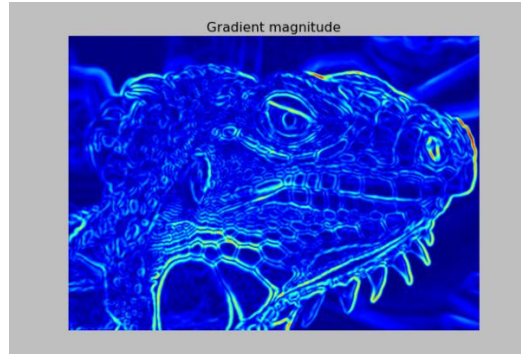
Derivative in y direction



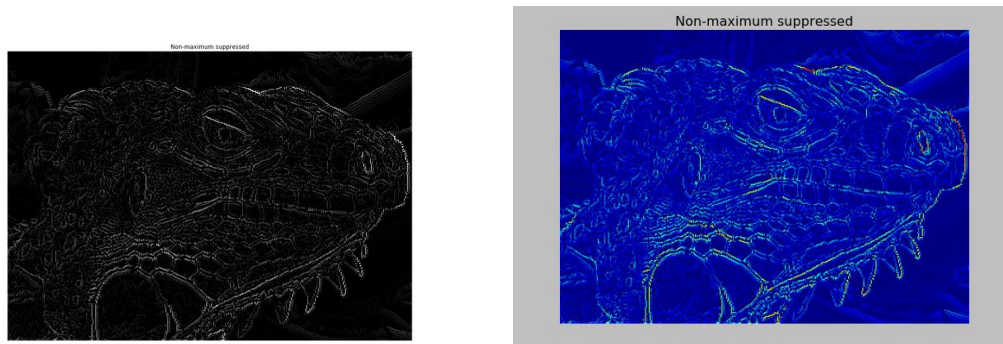
Gradient magnitude



Gradient magnitude



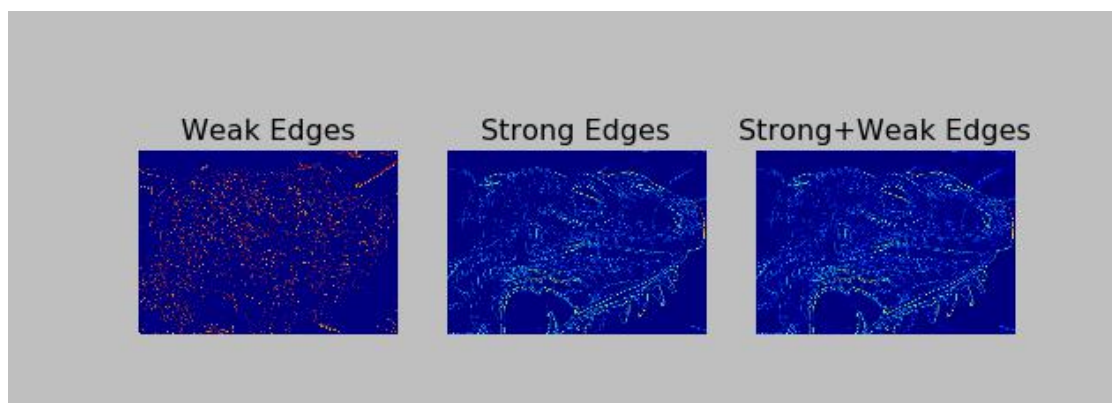
3.3 Non-maximum suppression



3.4 Double thresholding

在非最大抑制步骤之后剩余的边缘像素（仍然）以它们的像素逐个像素的强度标记。其中许多可能是图像中的真实边缘，但有些可能是由噪声或颜色变化引起的，例如由于粗糙的表面。辨别这两者之间最简单的方法是使用阈值，以便只保留一定值的边缘。Canny 边缘检测算法使用双重阈值。比高阈值更强的边缘像素被标记为强；弱于低阈值的边缘像素被抑制，并且两个阈值之间的边缘像素被标记为弱。

强边缘被解释为“某些边缘”，并可以立即包含在最终的边缘图像中。包含弱边缘的情况当且仅当它们连接到强边缘时。逻辑当然是噪声和其他小的变化不太可能导致强大的边缘（通过适当调整阈值水平）。因此，强边将（几乎）只是由于原始图像中的真实边缘。弱边缘可能是由于真实的边缘或噪声/颜色变化。后一种类型可能在整个图像上依赖于边缘分布，因此只有少量将位于强边缘附近。由于真正的边缘而导致的弱边缘更可能直接连接到强边缘。



Tips:用到的一些函数

numpy 中 `np.nonzero()` 函数
返回数组 a 中非零元素的索引值数组。

- (1) 只有 a 中非零元素才会有索引值，那些零值元素没有索引值；
- (2) 返回的索引值数组是一个 2 维 tuple 数组，该 tuple 数组中包含一维的 array 数组。其中，一维 array 向量的个数与 a 的维数是一致的。

(3) 索引值数组的每一个 array 均是从一个维度上来描述其索引值。比如，如果 a 是一个二维数组，则索引值数组有两个 array，第一个 array 从行维度来描述索引值；第二个 array 从列维度来描述索引值。

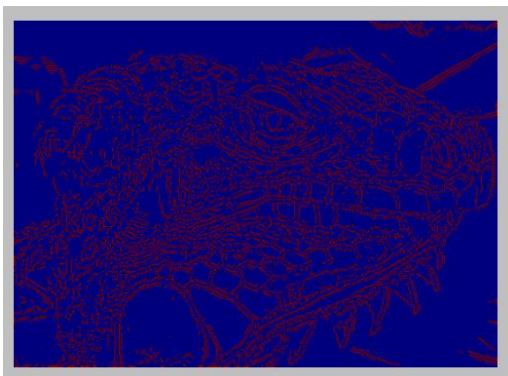
(4) 该 `np.transpose(np.nonzero(x))`

函数能够描述出每一个非零元素在不同维度的索引值。

通过 `a[np.nonzero(a)]` 得到所有 a 中的非零值

Numpy 中 `stack()` 函数详解

```
1 import numpy as np
2 a=[[1,2,3],
3    [4,5,6]]
4 print("列表a如下:")
5 print(a)
6
7 print("增加一维,新维度的下标为0")
8 c=np.stack(a,axis=0)
9 print(c)
10
11 print("增加一维,新维度的下标为1")
12 c=np.stack(a,axis=1)
13 print(c)
14
15 输出:
16 列表a如下:
17 [[1, 2, 3], [4, 5, 6]]
18 增加一维,新维度下标为0
19 [[1 2 3]
20  [4 5 6]]
21 增加一维,新维度下标为1
22 [[1 4]
23  [2 5]
24  [3 6]]
```



3.5 Edge tracking by hysteresis

按作业要求：

评估边缘检测器的一种方法是将检测到的边缘与手动指定的地面真实边缘进行比较。在

这里，我们使用精确度，召回率和 F1 分数作为评估指标。我们为您提供 40 个具有地面实况边缘注释的物体图像。运行下面的代码来计算整个图像集的精确度，召回率和 F1 分数。然后，调整 Canny 边缘检测器的参数以获得尽可能高的 F1 分数。通过仔细设置参数，您应该能够达到高于 0.31 的 F1 分数。

因运行速度较慢，只显示了前几个参数设置的情况：

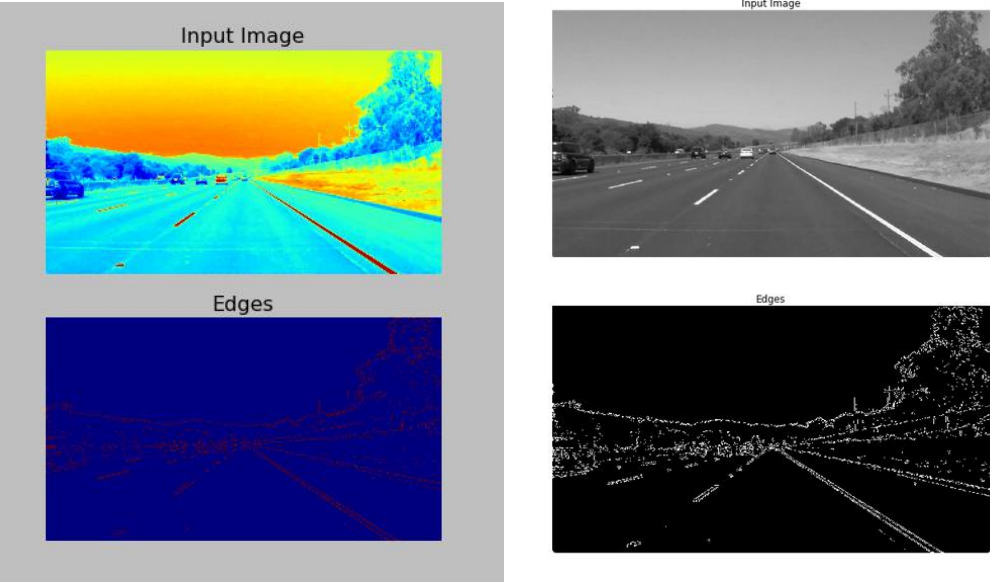
```
Total precision=0.1953, Total recall=9.3704
F1 score=0.3826
sigma=1, high=0.03, low=0.022
Total precision=0.1953, Total recall=9.3704
F1 score=0.3826
sigma=1, high=0.03, low=0.024
Total precision=0.1953, Total recall=9.3704
F1 score=0.3826
sigma=1, high=0.03, low=0.026
Total precision=0.1953, Total recall=9.3704
F1 score=0.3826
sigma=1, high=0.03, low=0.028
```

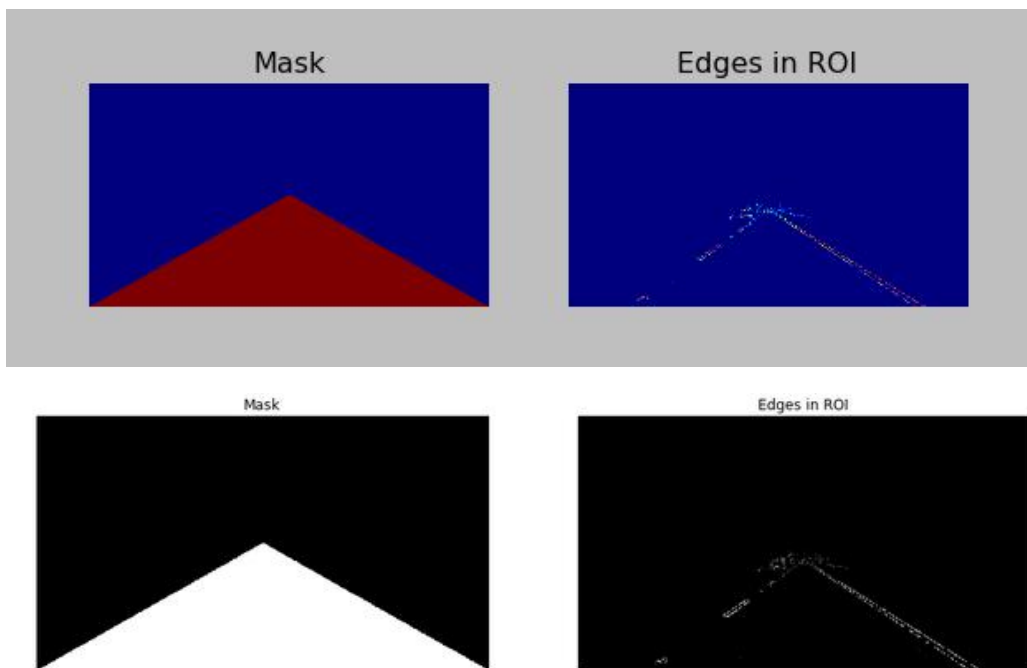
3.6 车道检测应用程序

使用 Canny 边缘检测器和 Hough 变换实现一个简单的车道检测应用程序。 以下是一些关于最终车道检测器外观的示例图像。

该算法可以分解为以下几个步骤：

- 1. 使用 Canny 边缘检测器检测边缘。
- 2. 提取感兴趣区域的边缘（覆盖图像底角和中心的三角形）。
- 3. 运行 Hough 变换来检测车道





Tip:用到的一些函数

①`np.ceil(a)` `np.floor(a)`: 计算各元素的 ceiling 值, floor 值 (ceiling 向上取整, floor 向下取整)

②`np.linspace`: 在默认情况下, `linspace` 函数可以生成元素为 50 的等间隔数列。而前两个参数分别是数列的开头与结尾。如果写入第三个参数, 可以制定数列的元素个数

```
x2 = np.linspace(1,10,10) ----[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

③`np.deg2rad(x)` is $x * \pi / 180$.

Examples:

```
np.deg2rad(180)
3.1415926535897931
```

④`min/max` 与 `np.argmin/np.argmax`:

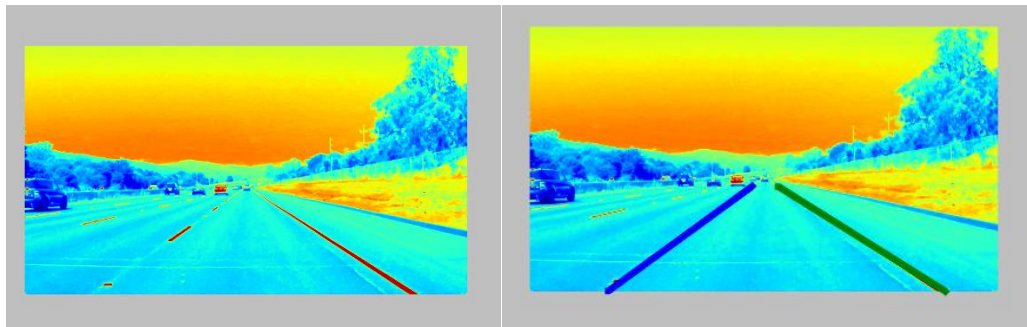
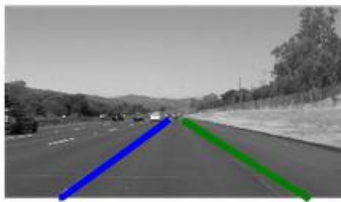
(1)`min/max` 与 `np.argmin/np.argmax` 函数的功能不同:前者返回值,后者返回最值所在的索引 (下标)

(2)处理的对象不同:前者跟适合处理 list 等可迭代对象,而后者自然是 numpy 里的核心数据结构 ndarray (多维数组)

(3)`min/max` 是 python 内置的函数,`np.argmin/np.argmax` 是 numpy 库中的成员函数



0 1 2 3 4 5 6 7 8 9 10



Lecture #6-7: Features and Fitting/Feature Descriptors

Lecture #8: Feature Descriptors and Resizing

1. Panorama Stitching(全景拼接)

实现过程：

In this assignment, we will detect and match keypoints from multiple images to build a single panoramic image. This will involve several tasks:

1. Use Harris corner detector to find keypoints.
2. Build a descriptor to describe each point in an image.
Compare two sets of descriptors coming from two different images and find matching keypoints.
3. Given a list of matching keypoints, use least-squares method to find the affine transformation matrix that maps points in one image to another.
4. Use RANSAC to give a more robust estimate of affine transformation matrix.

Given the transformation matrix, use it to transform the second image and overlay it on the first image, forming a panorama.

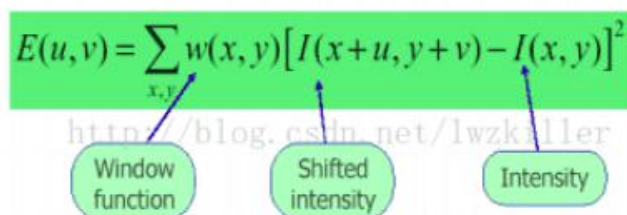
5. Implement a different descriptor (HOG descriptor) and get another stitching result.

2. Harris 角点检测

2.1 基本原理及思想

算法基本思想是使用一个固定窗口在图像上进行任意方向上的滑动，比较滑动前与滑动后两种情况，窗口中的像素灰度变化程度，如果存在任意方向上的滑动，都有着较大灰度变化，那么我们可以认为该窗口中存在角点

当窗口发生 $[u, v]$ 移动时，那么滑动前与滑动后对应的窗口中的像素点灰度变化描述如下

$$E(u, v) = \sum_{x, y} w(x, y) [I(x+u, y+v) - I(x, y)]^2$$


公式解释：

$>[u, v]$ 是窗口的偏移量

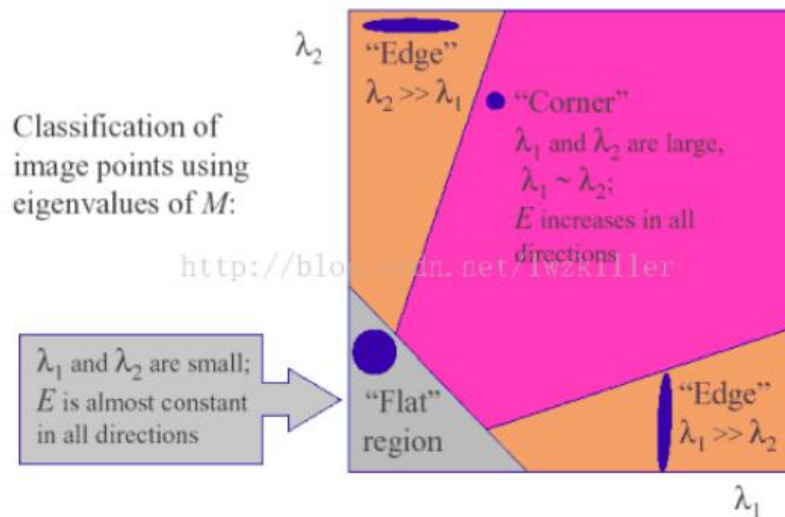
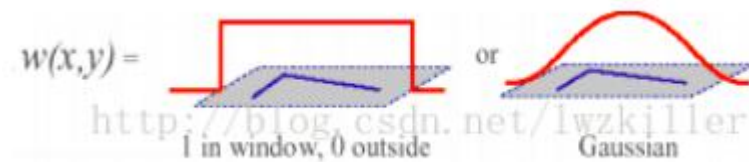
$>(x, y)$ 是窗口内所对应的像素坐标位置，窗口有多大，就有多少个位置

$>w(x, y)$ 是窗口函数，最简单情形就是窗口内的所有像素所对应的 w 权重系数均为1。

但有时候，我们会将 $w(x, y)$ 函数设定为以窗口中心为原点的二元正态分布。如果窗口中心点是角点时，移动前与移动后，该点的灰度变化应该最为剧烈，所以该点权重系数可以设定

大些，表示窗口移动时，该点在灰度变化贡献较大；而离窗口中心(角点)较远的点，这些点的灰度变化几近平缓，这些点的权重系数，可以设定小点，以示该点对灰度变化贡献较小，那么我们自然想到使用二元高斯函数来表示窗口函数，这里仅是个人理解，大家可以参考下。

所以通常窗口函数有如下两种形式：



如果存在两个主分量所对应的特征值都比较大，说明像素点的梯度分布比较散，梯度变化程度比较大，符合角点在窗口区域的特点；

如果是平坦区域，那么像素点的梯度所构成的点集比较集中在原点附近，因为窗口区域内的像素点的梯度幅值非常小，此时矩阵 M 的对角化的两个特征值比较小；

如果是边缘区域，在计算像素点的 x 、 y 方向上的梯度时，边缘上的像素点的某个方向的梯度幅值变化比较明显，另一个方向上的梯度幅值变化较弱，其余部分的点都还是集中原点附近

这样 M 对角化后的两个特征值理论应该是一个比较大，一个比较小，当然对于边缘这种情况，可能是呈 45° 的边缘，致使计算出的特征值并不是都特别的大，总之跟含有角点的窗口的分布情况还是不同的。

可以得出下列结论：

- >特征值都比较大时，即窗口中含有角点
- >特征值一个较大，一个较小，窗口中含有边缘
- >特征值都比较小，窗口处在平坦区域

$$R = \det M - k(\text{trace } M)^2$$

$$\begin{aligned}\det M &= \lambda_1 \lambda_2 \\ \text{trace } M &= \lambda_1 + \lambda_2\end{aligned}$$

2.2 基本步骤

1. Compute x and y derivatives (I_x, I_y) of an image
2. Compute products of derivatives (I_x^2, I_y^2, I_{xy}) at each pixel
3. Compute matrix M at each pixel, where

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

4. Compute corner response $R = \text{Det}(M) - k(\text{Trace}(M))^2$ at each pixel
5. Output corner response map $R(x, y)$

2.3 A simple descriptor

2.3.1 对图像归一化

这种方法给予原始数据的均值 (mean) 和标准差 (standard deviation) 进行数据的标准化。经过处理的数据符合标准正态分布，即均值为 0，标准差为 1，转化函数为：

$$\mathbf{x}^* = \frac{\mathbf{x} - \mu}{\sigma}$$

其中 μ 为所有样本数据的均值， σ 为所有样本数据的标准差。

2.3.2 match_descriptors

通过查找它们之间的距离来匹配特征描述符。当到最近的矢量的距离远小于到第二个最近的矢量的距离时，形成匹配，即距离的比率应该小于阈值。将匹配返回为矢量索引对。

2.4 特征匹配中的欧氏距离

欧式距离算法的核心是：设图像矩阵有 n 个元素 (n 个像素点)，用 n 个元素值 (x_1, \dots, x_n) 表示。

x_2, \dots, x_n) 组成该图像的特征组 (像素点矩阵中所有的像素点), 特征组形成了 n 维空间 (欧式距离就是针对多维空间的), 特征组中的特征码 (每一个像素点) 构成了每一维的数值, 就是 x_1 (第一个像素点) 对应一维, x_2 (第二个像素点) 对应二维, \dots , x_n (第 n 个像素点) 对应 n 维。在 n 维空间下, 两个图像矩阵各形成了一个点, 然后利用数学上的欧式距离公式计算这两个点之间的距离, 距离最小者就是最匹配的图像。

欧式距离公式:

点 $A = (x_1, x_2, \dots, x_n)$

点 $B = (y_1, y_2, \dots, y_n)$

$AB^2 = (x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2$

AB 就是所求的 A, B 两个多维空间中的点之间的距离

Tip: `Scipy.spatial.distance.cdist`

`scipy.spatial.distance.cdist(XA, XB, metric='euclidean', *args, **kwargs)`
[\[source\]](#)

Computes distance between each pair of the two collections of inputs.

See Notes for common calling conventions.

Parameters: **`XA`** : *ndarray*

An m_A by n array of m_A original observations in an n -dimensional space. Inputs are converted to float type.

`XB` : *ndarray*

An m_B by n array of m_B original observations in an n -dimensional space. Inputs are converted to float type.

`metric` : *str or callable, optional*

The distance metric to use. If a string, the distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule'.

Returns:

`Y` : *ndarray*

A m_A by m_B distance matrix is returned. For each i and j , the metric `dist(u=XA[i], v=XB[j])` is computed and stored in the ij th entry.

2.5 最小二乘法转换矩阵

我们现在有两个图像中匹配的关键点列表。我们将使用它来找到一个变换矩阵，将第二个图像中的点映射到第一个图像中的对应坐标。换句话说，如果图像 1 中的点 $p_1 = [y_1, x_1]$ 与图像 2 中的 $p_2 = [y_2, x_2]$ 匹配，我们需要找到一个仿射变换矩阵 H

$$\tilde{p}_2 H = \tilde{p}_1,$$

其中 \tilde{p}_1 和 \tilde{p}_2 是 p_1 和 p_2 的同质坐标。

请注意，不可能找到将图像 2 中的每个点完全映射到图像 1 中对应点的变换 H 。但是，我们可以用最小二乘估计变换矩阵。给定 N 匹配关键点对，令 X_1 和 X_2 为 $N \times 3$ 个矩阵，其行分别为图像 1 和图像 2 中对应关键点的同质坐标。那么，我们可以通过求解最小二乘问题来估计 H

$$X_2 H = X_1$$

Tip1: `numpy.linalg.lstsq`

```
numpy.linalg.lstsq(a, b, rcond='warn')[source]
```

Return the least-squares solution to a linear matrix equation.

Solves the equation $a x = b$ by computing a vector x that minimizes the Euclidean 2-norm $\|b - a x\|^2$. The equation may be under-, well-, or over-determined (i.e., the number of linearly independent rows of a can be less than, equal to, or greater than its number of linearly independent columns). If a is square and of full rank, then x (but for round-off error) is the “exact” solution of the equation.

返回元组，元组中四个元素，第一个元素表示所求的最小二乘解，第二个元素表示残差总和，第三个元素表示 X_1 矩阵秩，第四个元素表示 X_1 的奇异值

Tip2: `np.pad()`

对数组的填充

解释：

第一个参数是待填充数组

第二个参数是填充的形状，(2,3) 表示前面两个，后面三个

第三个参数是填充的方法

填充方法：

constant 连续一样的值填充，有关于其填充值的参数。constant_values= (x, y) 时前面用 x 填充，后面用 y 填充。缺参数是为 0000 。。。

edge 用边缘值填充

linear_ramp 边缘递减的填充方式

maximum, mean, median, minimum 分别用最大值、均值、中位数和最小值填充

reflect, symmetric 都是对称填充。前一个是关于边缘对称，后一个是关于边缘外的空气对称

wrap 用原数组后面的值填充前面，前面的值填充后面

也可以有其他自定义的填充方法

Parameters:

- a** : (M, N) array_like
"Coefficient" matrix.
- b** : $\{(M,), (M, K)\}$ array_like
Ordinate or "dependent variable" values. If *b* is two-dimensional, the least-squares solution is calculated for each of the *K* columns of *b*.
- rcond** : float, optional
Cut-off ratio for small singular values of *a*. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value of *a*.
Changed in version 1.14.0: If not set, a FutureWarning is given. The previous default of `-1` will use the machine precision as *rcond* parameter, the new default will use the machine precision times $\max(M, N)$. To silence the warning and use the new default, use `rcond=None`, to keep using the old behavior, use `rcond=-1`.

Returns:

- x** : $\{(N,), (N, K)\}$ ndarray
Least-squares solution. If *b* is two-dimensional, the solutions are in the *K* columns of *x*.
- residuals** : $\{(1,), (K,), (0,)\}$ ndarray
Sums of residuals; squared Euclidean 2-norm for each column in `b - a*x`. If the rank of *a* is $< N$ or $M \leq N$, this is an empty array. If *b* is 1-dimensional, this is a $(1,)$ shape array. Otherwise the shape is $(K,)$.
- rank** : int
Rank of matrix *a*.
- s** : $(\min(M, N),)$ ndarray
Singular values of *a*.

Raises:

- LinAlgError**
If computation does not converge.

3. 图像仿射变换及图像扭曲(Image Warping)

空间图像几何变换包括两个主要步骤：

(1) 空间坐标变换

(2) 变换坐标的赋值、插值运算

空间坐标变换一般可以表达为如下式子：

$$(x, y) = T\{(u, w)\}$$

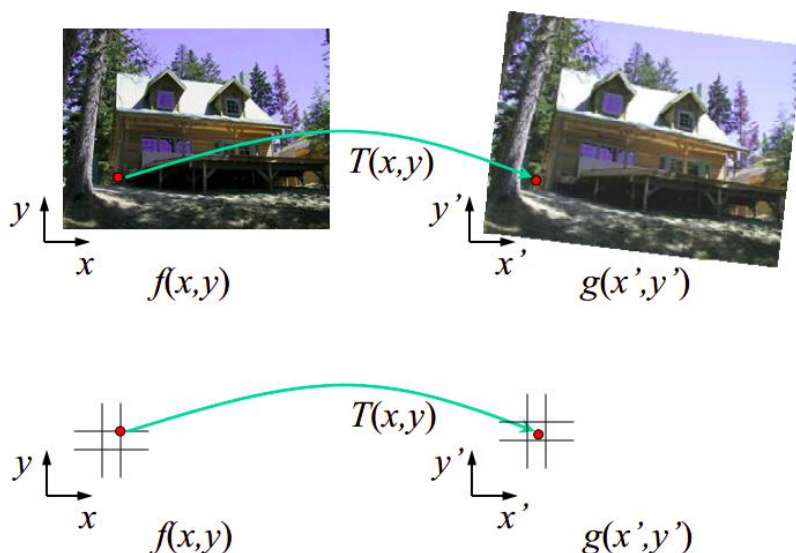
对于用得普遍的仿射变换，可以表达为如下式子：

$$\begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} v & w & 1 \end{bmatrix} T^T = \begin{bmatrix} v & w & 1 \end{bmatrix} \begin{bmatrix} t_{11} & t_{12} & 0 \\ t_{21} & t_{22} & 0 \\ t_{31} & t_{32} & 1 \end{bmatrix}$$

(x, y) 为变换后的坐标， (v, w) 为变换前的坐标。通过变换矩阵 T ，可以进行图像的缩放，旋转，平移等。有了坐标的变换，下面一步就是进行像素灰度级的赋值了。从原始图像映射到变换图像，赋值的时候需要进行插值运算

仿射变换的原始坐标中，首先将原坐标变换为齐次坐标(齐次坐标的理解)。

Image Warping 同时也分为 Forward Warping 和 Backward Warping。下面一一介绍：



仿射变换需要变换矩阵 T ，那么我们这里需要的是变换矩阵 H ，英文叫 Homography，单应矩阵。如果已经有幅图像，只需要找到原始图像中的任意四个点坐标(其中至少三个点不在同一条直线上)，并且指定目标图像中的四个点，这样通过这八个点，就能求出变换矩阵 H 。由于楼主只实验了 Backward Warping，所以下面以 Backward Warping 为例子进行说明，Forward Warping 与其类似，但是变换方向不一样，自然 H 的方向就不一样。具体过程如下：

$$\begin{pmatrix} x & y & 1 & 0 & 0 & 0 & -xx' & -yx' \\ 0 & 0 & 0 & x & y & 1 & -xy' & -yy' \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ \vdots \end{pmatrix} \Leftrightarrow \mathbf{A}\mathbf{h} = \mathbf{b}$$

变换矩阵 H 是 3X3，根据对其次坐标的理解，H 的最后一个元素始终为 1，又由于只需要各四个点，所以可以看到最后只取到了 h32。并且 x,y 全部已知，可以通过最小二乘方法求取 H：

最后，再在得到的 H 中，根据齐次坐标的概念，求得最终映射的坐标点：

$$\mathbf{A}\mathbf{h} = \mathbf{b} \Leftrightarrow \mathbf{A}^T \mathbf{A}\mathbf{h} = \mathbf{A}^T \mathbf{b} \Leftrightarrow \mathbf{h} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}$$

最后，再在得到的 H 中，根据齐次坐标的概念，求得最终映射的坐标点：

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \Leftrightarrow \mathbf{x}' = \mathbf{H}\mathbf{x}$$

$$x' = \frac{x'_1}{x'_3} = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}, \quad y' = \frac{x'_2}{x'_3} = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}$$

注意，h33 = 1。

4. RANSAC(Random Sample Consensus)去噪

4.1. 算法原理

RANSAC 算法的具体描述是：给定 N 个数据点组成的集合 P，假设集合中大多数的点都是可以通过一个模型来产生的，且最少通过 n 个点 (n < N) 可以拟合出模型的参数，则可以通过以下的迭代方式拟合该参数。

对下面的操作执行 k 次：

- (1) 从 P 中随机选择 n 个数据点；
- (2) 用这 n 个数据点拟合出一个模型 M；
- (3) 对 P 中剩余的数据点，计算每个点与模型 M 的距离，距离超过阈值的则认定为局外点，不超过阈值的认定为局内点，并记录该模型 M 所对应的局内点的值 m；

迭代 k 次以后，选择 m 最大的模型 M 作为拟合的结果。

因为在实际应用中 N 的值通常会很大，那么从其中任选 n 个数据点的组合就会很大，如

果对所有组合都进行上面的操作运算量就会很大,因此对于 k 的选择就很重要。通常情况下,只要保证模型估计需要的 n 个点都是内点的概率足够高即可。因此设 w 为 N 个数据中内点的比例, z 为进行 k 次选取后,至少有一次选取的 n 个点都是内点的概率。则有

$$z=1-(1-wn)^k$$

其中 $1-wn$ 表示一次选取不都是内点的概率, $(1-wn)^k$ 表示 k 次选取中没有一次都是内点的概率。

则有

$$k=\frac{\log(1-z)}{\log(1-wn)}$$

这里 z 一般要求满足大于 95%即可

4.2 实现步骤

The steps of RANSAC are:

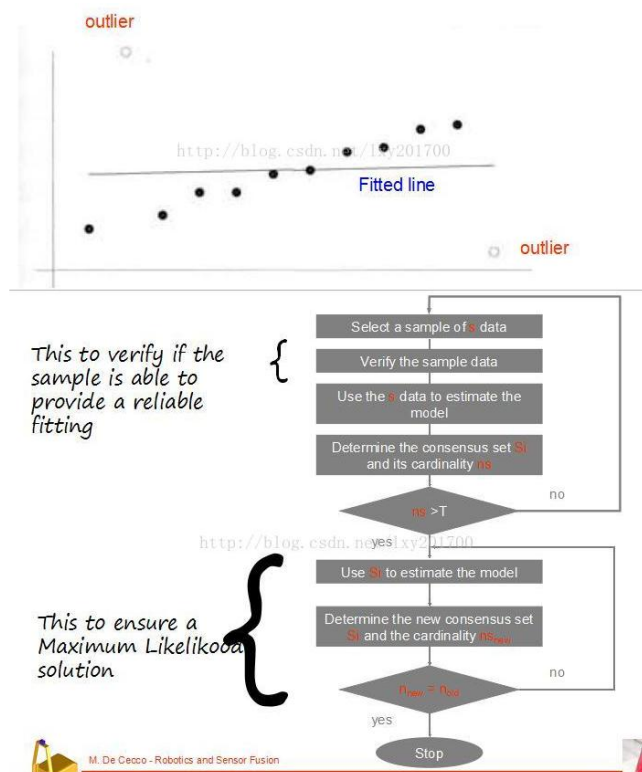
1. Select random set of matches
2. Compute affine transformation matrix
3. Find inliers using the given threshold
4. Repeat and keep the largest set of inliers
5. Re-compute least-squares estimate on all of the inliers

1. Randomly select a seed group from data.
2. Perform parameter estimation using the selected seed group.
3. Identify the inliers (points close to the estimated model).
4. (If there exists a sufficiently large number of inliers,)re-estimate the model using all inliers.
5. repeat steps 1-4 and finally keep the estimate with most inliers and best fit

即随机采样一致性,是一种简单且有效的去除噪声影响,估计模型的一种方法。与普通的去噪算法不同,RANSAC 算法是使用尽可能少的点来估计模型参数,然后尽可能的扩大得到的模型参数的影响范围。

4.3 代码思路

首先,从已求得的配准点对中抽取几对配准点,计算变换矩阵,并将这几对点记录为”内点”。继续寻找配准点对中的非内点,若这些配准点对符合矩阵,则将其添加到内点。当内点中的点对数大于设定阈值时,则判定此矩阵为精确的变换矩阵。依照以上方法,随机采样 N 次,选取内点数最大集合,剔除非内点等误配点对。



5. Histogram of Oriented Gradients(HOG)

5.1 基本原理

HOG stands for Histogram of Oriented Gradients. In HOG descriptor, the distribution (histograms) of directions of gradients (oriented gradients) are used as features. Gradients (x and y derivatives) of an image are useful because the magnitude of gradients is large around edges and corners (regions of abrupt intensity changes) and we know that edges and corners pack in a lot more information about object shape than flat regions.

5.2 实现步骤

1. compute the gradient image in x and y
Use the sobel filter provided by skimage.filters
2. compute gradient histograms
Divide image into cells, and calculate histogram of gradient in each cell.
3. normalize across block
Normalize the histogram so that they
4. flattening block into a feature vector

具体每一步的详细过程如下：

5.2.1 计算图像梯度

计算图像横坐标和纵坐标方向的梯度，并据此计算每个像素位置的梯度方向值；求导操作不仅能够捕获轮廓，人影和一些纹理信息，还能进一步弱化光照的影响。

图像中像素点(x,y)的梯度为：

$$\begin{aligned}G_x(x,y) &= H(x+1,y) - H(x-1,y) \\G_y(x,y) &= H(x,y+1) - H(x,y-1)\end{aligned}$$

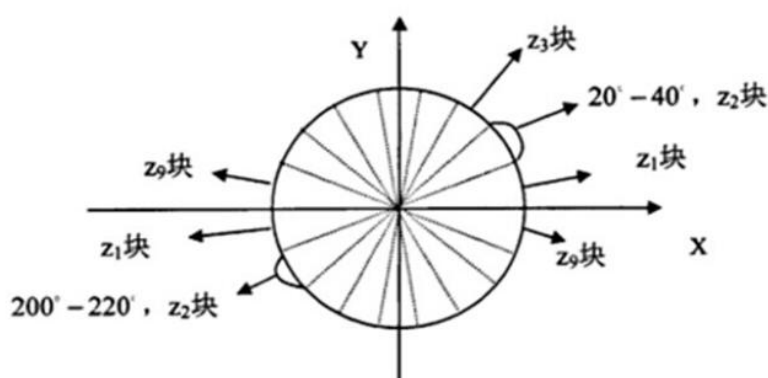
式中 $G_x(x,y)$, $G_y(x,y)$, $H(x,y)$ 分别表示输入图像中像素点(x,y)处的水平方向梯度、垂直方向梯度和像素值。像素点(x,y)处的梯度幅值和梯度方向分别为：

$$\begin{aligned}G(x,y) &= \sqrt{G_x(x,y)^2 + G_y(x,y)^2} \\ \alpha(x,y) &= \tan^{-1}\left(\frac{G_y(x,y)}{G_x(x,y)}\right)\end{aligned}$$

5.2.2 为每个细胞单元构建梯度方向直方图

这一步骤的目的是为局部图像区域提供一个指示函数量化梯度方向的同时能够保持对图像中人体对象的姿势和外观的弱敏感性。

将图像分成若干个“单元格 cell”，例如每个 cell 为 8×8 的像素大小。假设采用 9 个 bin 的直方图来统计这 8×8 个像素的梯度信息，即将 cell 的梯度方向 $0 \sim 180$ 度（或 $0 \sim 360$ 度，考虑了正负，signed）分成 9 个方向块。如下图所示：如果这个像素的梯度方向是 $20 \sim 40$ 度，直方图第 2 个 bin 即的计数就加 1，这样，对 cell 内每个像素用梯度方向在直方图进行加权投影，将其映射到对应的角度范围块内，就可以得到这个 cell 的梯度方向直方图了，就是该 cell 对应的 9 维特征向量（因为有 9 个 bin）。这边的加权投影所用的权值为当前点的梯度幅值。例如说：某个像素的梯度方向是在，其梯度幅值是 4，那么直方图第 2 个 bin 的计数就不是加 1 了，而是加 4。这样就得到关于梯度方向的一个加权直方图

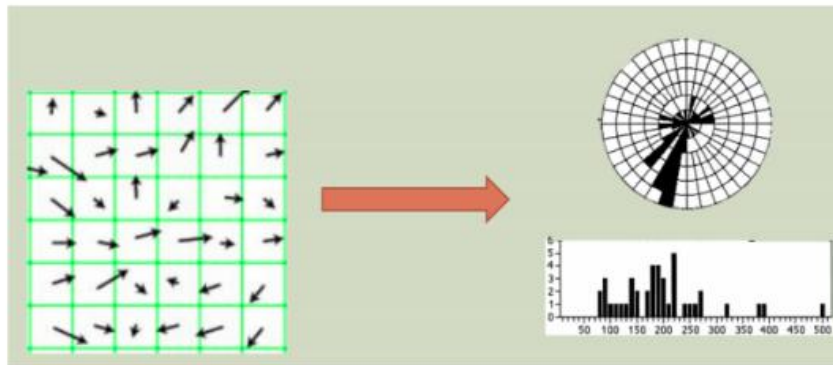


5.2.3 block 归一化

把细胞单元组合成大的块（block）并归一化梯度直方图

由于局部光照的变化以及前景-背景对比度的变化，使得梯度强度的变化范围非常大。这就需要对梯度强度做归一化。归一化能够进一步地对光照、阴影和边缘进行压缩。

R-HOG 区间大体上是一些方形的格子，它可以有三个参数来表征：每个区间中细胞单元的数目、每个细胞单元中像素点的数目、每个细胞的直方图通道数目。



Tips:一些函数

`np.max` 与 `np.maximum`

1. 参数

首先比较二者的参数部分：

`np.max(a, axis=None, out=None, keepdims=False)`

求序列的最值

最少接收一个参数

`axis`：默认为列向（也即 `axis=0`），`axis = 1` 时为行方向的最值；

`np.maximum(X, Y, out=None)`

`X` 与 `Y` 逐位比较取其大者；

最少接收两个参数

6. 作业题(hw3)

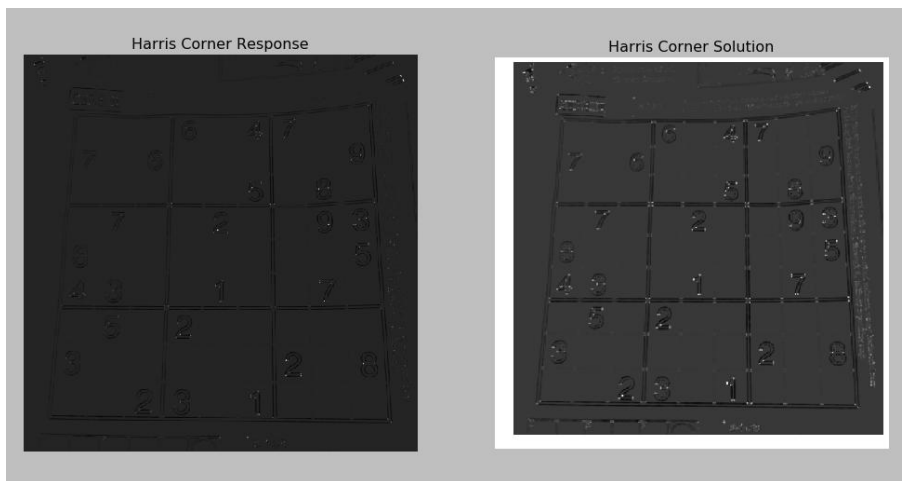
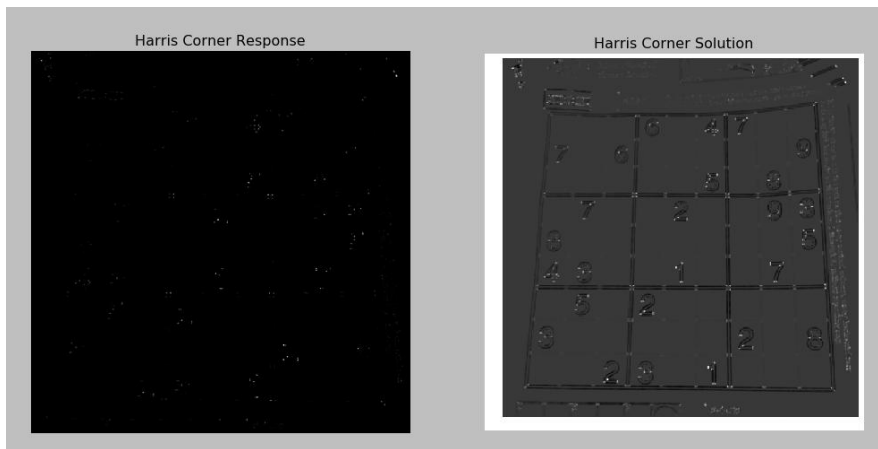
6.1 Harris Corner Detector

高斯滤波窗函数：

$$M = g(\sigma) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

Two dimension:

$$G(x, y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



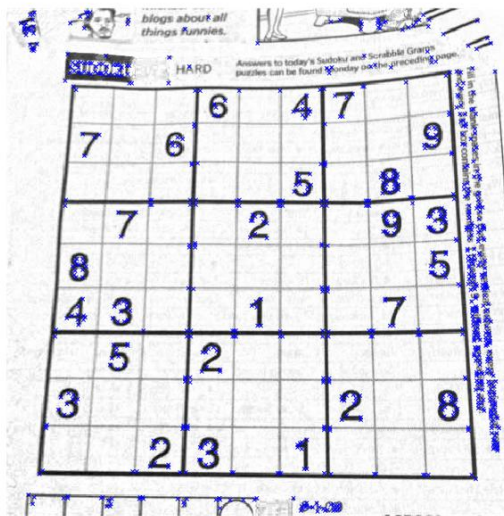
```
H, W = img.shape
window = np.ones((window_size, window_size))
response = np.zeros((H, W))
max = 0
for i in range(0, H):
    for j in range(0, W):
        if (img[i][j] > max):
            max = img[i][j]

for i in range(0, H):
    for j in range(0, W):
        img[i][j] = img[i][j] / max * 255

dx = filters.sobel_v(img)
dy = filters.sobel_h(img)
# YOUR CODE HERE
window = gaussian_kernel_2(window_size, 0.21)
# window = ([[4, 12, 4], [12, 36, 12], [4, 12, 4]])
c = convolve(dx * dy, window)
a = convolve(pow(dx, 2), window)
b = convolve(pow(dy, 2), window)

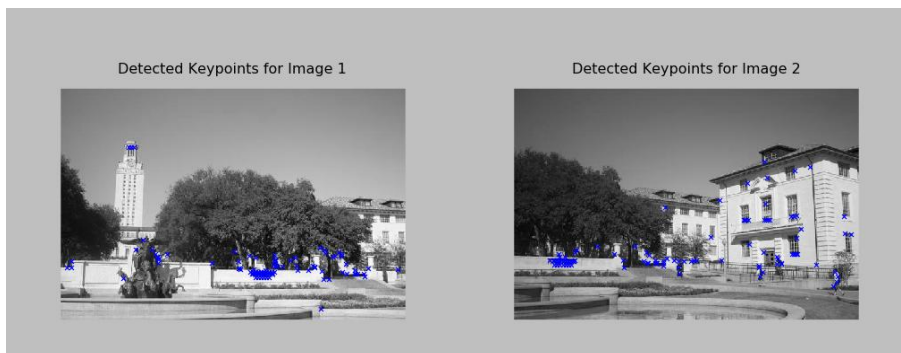
for i in range(0, H):
    for j in range(0, W):
        response[i][j] = pow((a[i][j] * b[i][j]) - (c[i][j] * c[i][j]), 2) - k * pow(a[i][j] + b[i][j], 2)
```

Detected Corners



6.2 Describing and Matching Keypoints

按作业要求得出结果



6.2.1 Creating Descriptors

按作业要求得出结果

```
def simple_descriptor(patch):
    """
    Describe the patch by normalizing the image values into a standard
    normal distribution (having mean of 0 and standard deviation of 1)
    and then flattening into a 1D array.

    The normalization will make the descriptor more robust to change
    in lighting condition.

    Hint:
    |   If a denominator is zero, divide by 1 instead.

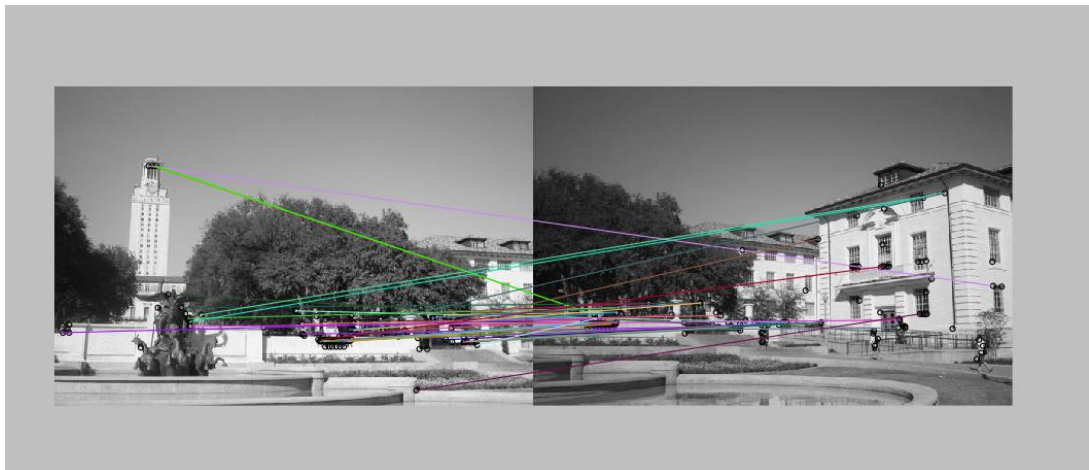
    Args:
    |   patch: grayscale image patch of shape (h, w)

    Returns:
    |   feature: 1D array of shape (h * w)
    """
    feature = []
    # YOUR CODE HERE

    h, w = patch.shape
    feature = np.zeros(h * w)
    x = 0
    sum = np.sum(patch)
    ave = sum / (h * w)
    d = np.sum(pow(patch - ave, 2))
    s = pow(d, 0.5)
    patch = (patch - ave) / s
    for i in range(0, h):
        for j in range(0, w):
            feature[x] = patch[i][j]
            x += 1
        pass
    # END YOUR CODE
    return feature
```

6.2.2 Matching Descriptors

按作业要求得出结果





6.3 Transformation Estimation

最小二乘法求解矩阵 H :

```

369 sol = np.array(
370     [[1.25, 2.5, 0.0],
371      [-5.75, -4.5, 0.0],
372      [0.25, -1.0, 1.0]]
373 )
374 print(H)
375 error = np.sum((H - sol) ** 2)
376
377 if error < 1e-20:
378     print('Implementation correct!')
379 else:
380     print('There is something wrong.')
381
382 def ransac(keypoints1, keypoints2, matches,
383           ...

```

```

(97, 2)
(103, 2)
[[ 1.25  2.5   0. ]
 [-5.75 -4.5   0. ]
 [ 0.25 -1.    1. ]]
Implementation correct!
[Finished in 2.5s]

```

Ransac 去噪

```

N = matches.shape[0]
n_samples = int(N * 0.2)

matched1 = pad(keypoints1[matches[:, 0].astype(int)])
matched2 = pad(keypoints2[matches[:, 1].astype(int)])
max_inliers = np.zeros(N)
n_inliers = 0

# RANSAC iteration start
# YOUR CODE HERE
p1 = []
p2 = []
p = random.sample(matches, n_samples)
for k in enumerate(p):
    p1.append(matched1[k.astype(int)])
    p2.append(matched2[k.astype(int)])
    pass
H = np.linalg.lstsq(p1, p2)
for kp in enumerate(matches):
    d = cdist(keypoints1[kp[0]] * H, keypoints2[kp[1]])
    if d <= threshold:
        n_inliers += 1
        matches.append(kp)
# dists = cdist(desc1, desc2)
pass

```


6.4 全景拼接:

根据作业要求及步骤
进行仿射变换及拼接
得到全景拼接图:



代码思路及部分过程

Handwritten notes detailing the RANSAC process for feature matching and affine transformation:

- keyframe 1:** $\{x_1, y_1, z_1, \dots, x_n, y_n, z_n\}$ (keypoints)
- keyframe 2:** $\{x_2, y_2, z_2, \dots, x_m, y_m, z_m\}$ (keypoints)
- Simple description:**
 - Input: patch: $n \times n$
 - Output: feature: $(1 \times n)$ (feature vector)
- describe keypoints:**
 - Input: patch size of patch, output: feature.
 - Input: $(x, y) \rightarrow 1 \times n$ feature vector.
- match description:**
 - Input: describe keypoints $n \times p$ (feature vectors).
 - Output: key $n \times p$ (keypoints).
- fit affine matrix:**
 - Input: $(p_1, p_2) \rightarrow n \times 2$ (keypoints).
 - Output: $H = 3 \times 3$ (affine transformation matrix).
- RANSAC:**
 - Input: keyframe 1, keyframe 2, matches: $n \times 2$ (feature vectors), threshold: value.
 - Process: Randomly select n_{sample} keypoints \rightarrow fit affine matrix H \rightarrow match keypoints with H \rightarrow find inliers \rightarrow find best H \rightarrow find inliers \rightarrow find best H .

6.5 Histogram of Oriented Gradients (HOG)

按作业要求:

6.5.1 特征点匹配



HOG descriptor Solution



6.5.2 全景拼接

HOG Descriptor Panorama Solution




```

def hog_descriptor(patch, pixels_per_cell=(8, 8)):
    """
    Generating hog descriptor by the following steps:
    1. compute the gradient image in x and y (already done for you)
    2. compute gradient histograms
    3. normalize across block
    4. flattening block into a feature vector

    Args:
        patch: grayscale image patch of shape (h, w)
        pixels_per_cell: size of a cell with shape (m, n)

    Returns:
        block: 1D array of shape ((h*w*n_bins)/(m*n))
    """
    assert (patch.shape[0] % pixels_per_cell[0] == 0), \
        'Heights of patch and cell do not match'
    assert (patch.shape[1] % pixels_per_cell[1] == 0), \
        'Widths of patch and cell do not match'

    n_bins = 9
    degrees_per_bin = 180 // n_bins

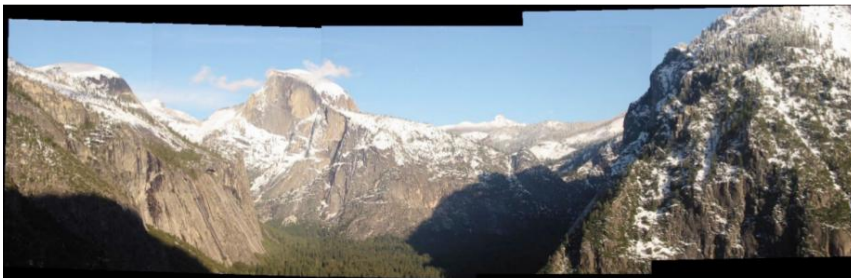
    Gx = filters.sobel_v(patch)
    Gy = filters.sobel_h(patch)

    # Unsigned gradients
    G = np.sqrt(Gx**2 + Gy**2)
    theta = (np.arctan2(Gy, Gx) * 180 / np.pi) % 180

    G_cells = view_as_blocks(G, block_shape=pixels_per_cell)
    theta_cells = view_as_blocks(theta, block_shape=pixels_per_cell)
    rows = G_cells.shape[0]
    cols = G_cells.shape[1]
    cells = np.zeros((rows, cols, n_bins))
    o = pixels_per_cell[0]
    p = pixels_per_cell[1]
    b = np.zeros((n_bins))
    x_theta = np.zeros((o, p))
    y_G = np.zeros((o, p))
    block = np.zeros((patch.shape[0] * patch.shape[1] * n_bins // o // p))
    block = []
    for i in range(0, rows):
        for j in range(0, cols):
            x_theta = flat(theta_cells[i][j])
            y_G = flat(G_cells[i][j])
            x_theta = (x_theta / 20).astype(int)
            for m in range(0, o * p):
                b[x_theta[m]] += 1 * y_G[m]
                pass
            block.append(b)

    block = np.array(block)
    block = simple_descriptor(block)

```



Lecture #9: Image Resizing and Segmentation

1. Seam Carving

1.1 算法原理

Retargeting means that we take an input and “retarget” to a different shape or size. Imagine input as being an image of size $n \times m$ and the desired output as an image of size $n' \times m'$. The idea behind retargeting is to

1. adhere to geometric constraints (e.g. aspect ratio),
2. preserve important content and structures, and
3. limit artifacts.

However, what is considered “important” is very subjective, for what may be important to one observer may not be important to another.

- Assume $m \times n \rightarrow m' \times n'$, $n' < n$ (summarization)
- Basic Idea: remove unimportant pixels from the image
 - Unimportant = pixels with less “energy”

$$E_1(\mathbf{I}) = \left| \frac{\partial}{\partial x} \mathbf{I} \right| + \left| \frac{\partial}{\partial y} \mathbf{I} \right|.$$

- Intuition for gradient-based energy:
 - Preserve strong contours
 - Human vision more sensitive to edges – so try remove content from smoother areas
 - Simple enough for producing some nice results

内容感知的图像缩放算法一般用于图像的裁剪,就是有的时候,你觉得一张照片有点大,你希望把它裁剪的小一些,但是你又想保留照片中的物体,这个时候就要用到内容感知的图片缩放算法了。

1.2 基本步骤

1.2.1. 计算图像能量图

The energy at each pixel is the sum of:

absolute value of the gradient in the xx direction

&&

absolute value of the gradient in the yy direction

能量图一般是图像像素的梯度模值,为了简化计算可先转换成灰度图像,然后直接采用如下公式(直接用 x、y 方向上的差分取绝对值,然后相加),其实这一步就是相当于边缘检测算法一样:

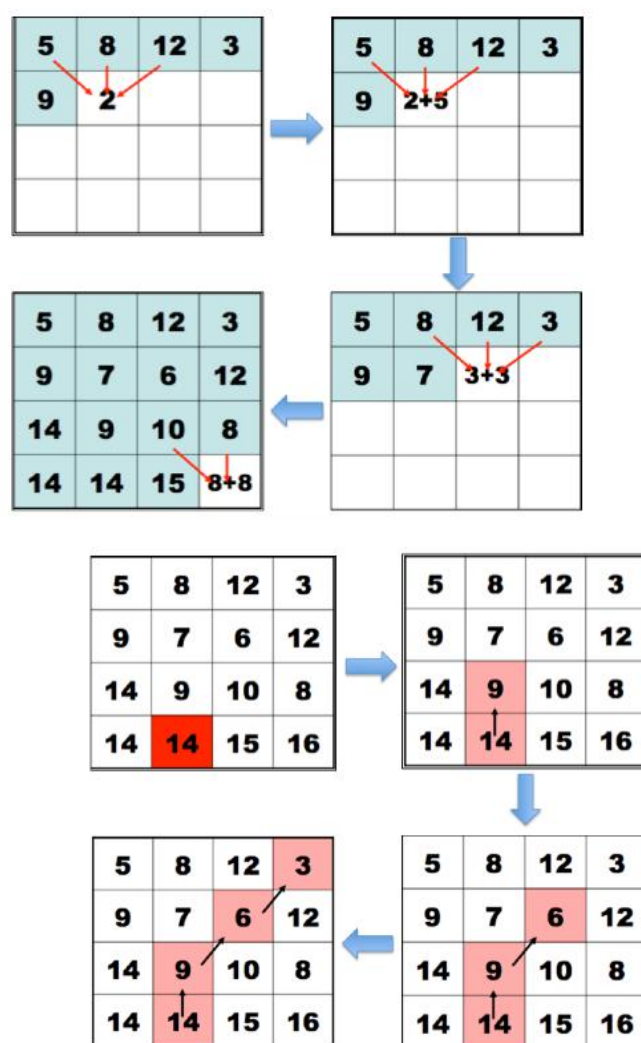
$$e(I) = \left| \frac{\partial}{\partial x} I \right| + \left| \frac{\partial}{\partial y} I \right|$$

1.2.2 寻找最小能量线

最小能量线指的是需要被移除的那一行:首先需要以图像第一行或最后一行为开始行进行迭代。下面实现的为从图像最后一行开始,往上迭代

找出最后一行需要被移除的像素点后,设其坐标为 $P(x,y)$,然后往上一行寻找,寻找的点为 P 点的在 $y-1$ 行中的三个相邻像素点中的能量最小值像素。也就是寻找的坐标为 $(x-1,y-1)$ 、 $(x,y-1)$ 、 $(x+1,y-1)$;

The recursion relation gives



1.2.3 移除得到的最小能量线

移除最小能量线,让图片的宽度缩小一个像素,同时所有位于最小能量线右边的像素点左移一个单位,从而实现图像缩小宽度缩小一个单位。

移除的时候 为了让图像看起来自然，需要在移除缝线的地方进行平均，假设移除坐标为 $P(x,y)$ ，那么移除后 $P(x-1,y)$ 的像素值为 $P(x-1,y)$ 与 $P(x,y)$ 的像素值的平均。 $P(x+1,y)$ 的像素值为 $P(x-1,y)$ 与 $P(x,y)$ 的像素值的平均，然后才能把 $P(x+1,y)$ 移动到 $P(x,y)$ 的位置。

2. 部分作业题(hw4)

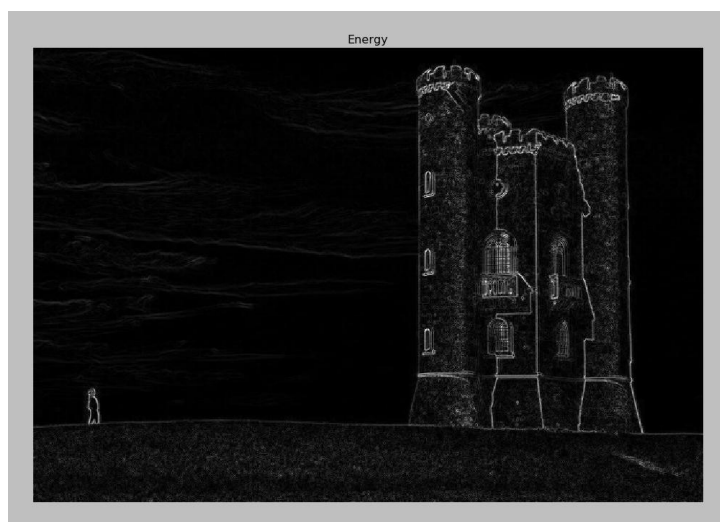


2.1 计算图像能量

The energy at each pixel is the sum of:

- absolute value of the gradient in the x direction
- absolute value of the gradient in the y direction

$$E(I) = \left| \frac{\partial}{\partial x} I \right| + \left| \frac{\partial}{\partial y} I \right|$$



Tip: 导入后首先要做灰度化处理:

采用下面的指令将图片进行灰度化处理:

```
img_gray=color.rgb2gray(img)
```

```
cost = np.zeros((H, W))
paths = np.zeros((H, W), dtype=np.int)

# Initialization
cost[0] = energy[0]
paths[0] = 0 # we don't care about the first row of paths

# YOUR CODE HERE
cost = energy
for i in range(1, H):
    for j in range(0, W):
        if(j == 0):
            cost[i][j] = cost[i][j] + \
                np.min([cost[i - 1][j], cost[i - 1][j + 1]])
            paths[i][j] = np.argmin(
                [cost[i - 1][j], cost[i - 1][j + 1]])
        elif(j == W - 1):
            cost[i][j] = cost[i][j] + \
                np.min([cost[i - 1][j], cost[i - 1][j - 1]])
            paths[i][j] = np.argmin(
                [cost[i - 1][j], cost[i - 1][j - 1]]) - 1
        else:
            cost[i][j] = cost[i][j] + \
                np.min([cost[i - 1][j - 1], cost[i - 1][j], cost[i - 1][j + 1]])
            paths[i][j] = np.argmin([cost[i - 1][j - 1], cost[i - 1][j], cost[i - 1][j + 1]]) - 1
        pass
    pass
pass
# END YOUR CODE

if axis == 0:
    cost = np.transpose(cost, (1, 0))
    paths = np.transpose(paths, (1, 0))
```

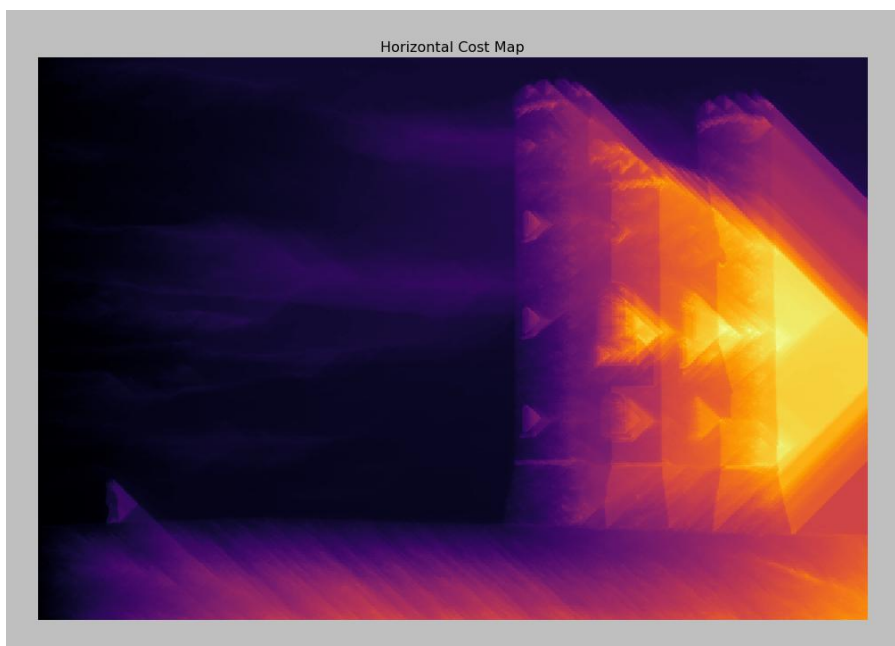
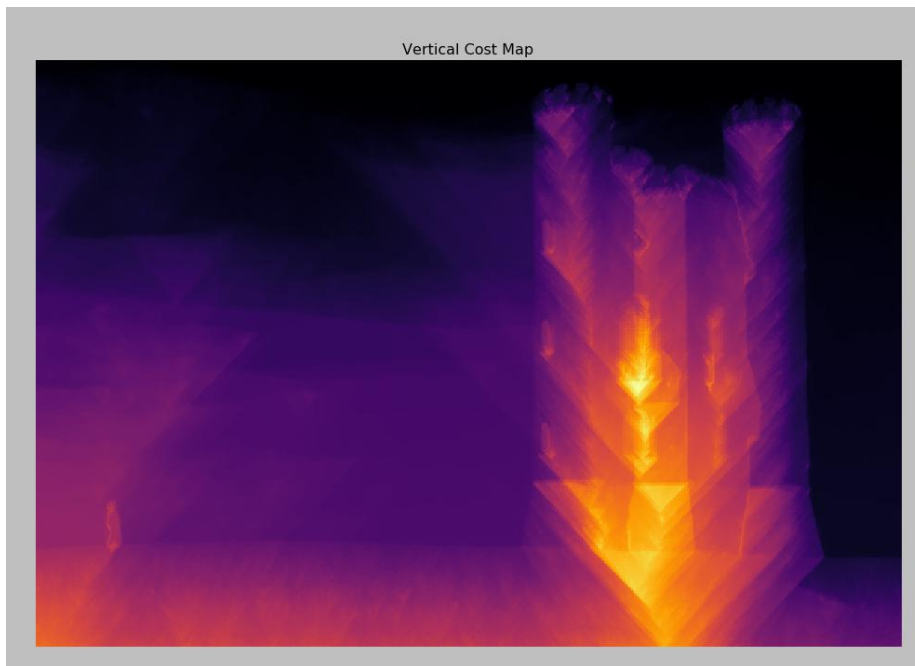
得到正确结果,与所给结果一致

```
Energy:
[[ 1.  2.  1.5]
 [ 3.  1.  2. ]
 [ 4.  0.5 3. ]]
Cost:
[[ 1.  2.  1.5]
 [ 4.  2.  3.5]
 [ 6.  2.5 5. ]]
Solution cost:
[[ 1.  2.  1.5]
 [ 4.  2.  3.5]
 [ 6.  2.5 5. ]]
Paths:
[[ 0  0  0]
 [ 0 -1  0]
 [ 1  0 -1]]
Solution paths:
[[ 0  0  0]
 [ 0 -1  0]
 [ 1  0 -1]]
[Finished in 0.7s]
```

2.2 Compute cost

Now implement the function `compute_cost`. Starting from the energy map, we'll go from the first row of the image to the bottom and compute the minimal cost at each pixel.

We'll use dynamic programming to compute the cost line by line starting from the first row.



2.3 Finding optimal seams

Using the cost maps we found above, we can determine the seam with the lowest energy in the image.

We can then remove this optimal seam, and repeat the process until we obtain a desired width.

按作业要求,循环删除能量最低的像素点,得到指定大小的图片:



```
def reduce(image, size, axis=1, efunc=energy function, cfunc=compute_cost):
    """Reduces the size of the image using the seam carving process.

    At each step, we remove the lowest energy seam from the image. We repeat the process
    until we obtain an output of desired size.
    Use functions:
    - efunc
    - cfunc
    - backtrack_seam
    - remove_seam

    Args:
    image: numpy array of shape (H, W, 3)
    size: size to reduce height or width to (depending on axis)
    axis: reduce in width (axis=1) or height (axis=0)
    efunc: energy function to use
    cfunc: cost function to use

    Returns:
    out: numpy array of shape (size, W, 3) if axis=0, or (H, size, 3) if axis=1
    """

    out = np.copy(image)
    if axis == 0:
        out = np.transpose(out, (1, 0, 2))

    H = out.shape[0]
    W = out.shape[1]

    assert W > size, "Size must be smaller than %d" % W
    assert size > 0, "Size must be greater than zero"

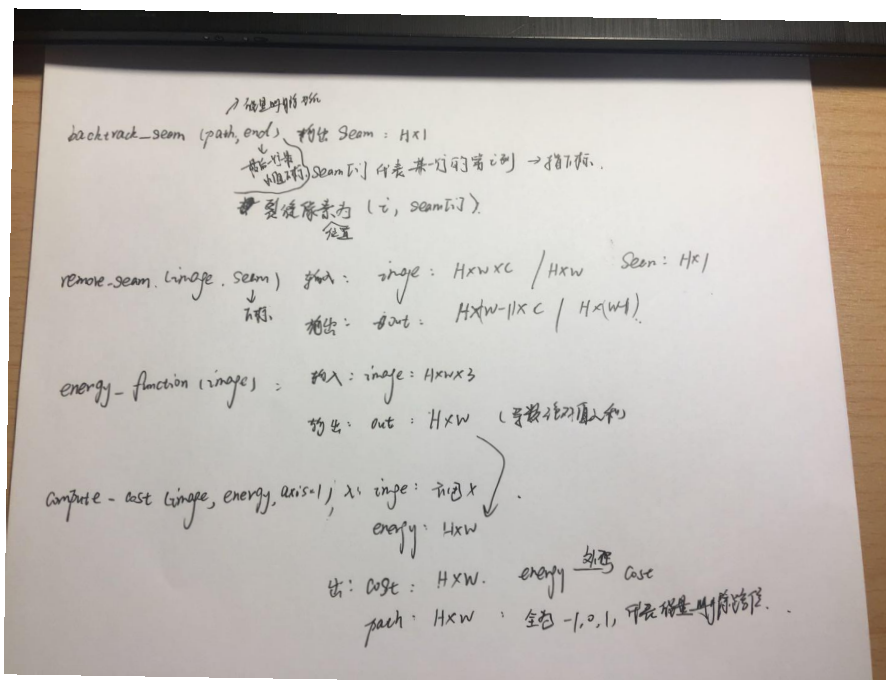
    # YOUR CODE HERE
    if (axis == 1):
        size_reduce = W - size
    elif (axis == 0):
        size_reduce = H - size
    else:
        pass
    for i in range(0, size_reduce):
        energy = efunc(image)
        vcost, vpaths = cfunc(image, energy)
        end = np.argmin(vcost[-1])
        seam = backtrack_seam(vpaths, end)
        out = remove_seam(image, seam)
        image = out
    # END YOUR CODE

    assert out.shape[1] == size, "Output doesn't have the right shape"

    if axis == 0:
        out = np.transpose(out, (1, 0, 2))

    return out
```

部分代码思路:



一些说明：

本周进度是 lecture 看到了第 10 章节,作业做到了第五次 hw4 的前半部分,hw4 还未做完,还有 hw3 的最后一道题关于多幅图像拼接还未完成,留到下一周再做汇报,这一周代码部分较多较长,所以有的题目只贴了部分代码和效果图,学习过程中遇到的主要问题在于有些实现算法的过程不太理解,有些公式不知道如何用代码实现,也经常会出现难以解决代码中的一些报错,涉及到多重循环和遍历的实现,代码运行速度较慢,耗时较长

后续学习过程中,会多推导多百度,希望能减少和避免这些问题