Sift 图像特征提取

1.原理简述

SIFT，即尺度不变特征变换（Scale-invariant feature transform，SIFT），是用于图像处理领域的一种描述。这种描述具有尺度不变性和旋转不变性，可在图像中检测出关键点，是一种局部特征描述子。

我将整个SIFT的实现分为两个部分。第一个部分是满足尺度不变性，也就是通过构建高斯差分金字塔，搭叠不同尺度的图像，然后对各个尺度下的关键点进行搜索和定位。第二个部分是为了满足旋转不变性，也就是对第一部分找到的关键点，通过梯度直方图计算出主方向，然后在该主方向下构建物体坐标系，对物体坐标系内的邻域进行计算，得出该关键点的特征向量。这个向量基于关键点的主方向，因此具有旋转不变性。

Figure 12: The training images for two objects are shown on the left. These can be recognized in a cluttered image with extensive occlusion, shown in the middle. The results of recognition are shown on the right. A parallelogram is drawn around each recognized object showing the boundaries of the original training image under the affine transformation solved for during recognition. Smaller squares indicate the keypoints that were used for recognition.

Lowe论文中的结果展示

根据助教的要求，第一部分寻找尺度不变的角点可以通过调用内置的 `cv2.goodFeaturesToTrack` 函数求得哈里斯角点。因此我先完成了第二部分（旋转不变性），再完成第一部分自己求角点。

实验环境：MacOS X, Python 3

2.求特征向量

- **梯度**

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y)=\arctan((L(x,y+1)-L(x,y-1))/(L(x+1,y)-L(x-1,y)))$$

这是Lowe论文中给出的求梯度的算子，也就是Roberts算子。根据上次实验Canny边缘检测的经验，我选择了使用Canny算子。考虑到边缘的处理以及后续会多次调用梯度幅值和方向，在程序的开头我就计算出这张图的梯度幅值与方向。

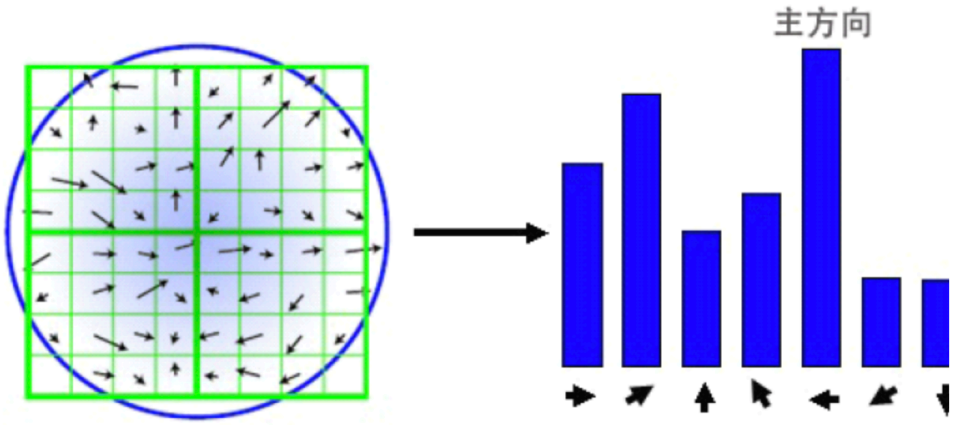
```
def getGrad(img):
    img=cv2.GaussianBlur(img,(7,7),1)*1.0
    cannyX = np.array([[-3,-10,-3],
                       [0,0,0],
                       [3,10,3]])
    cannyY = np.array([[-3,0,3],
                       [-10,0,10],
                       [-3,0,3]])
    grad_x=cv2.filter2D(img,ddepth=-1,kernel=cannyX,anchor=(-1,-1))
    grad_y=cv2.filter2D(img,ddepth=-1,kernel=cannyY,anchor=(-1,-1))
    gradMag=np.hypot(grad_x,grad_y)
    gradAng=np.arctan2(grad_y,np.add(grad_x,0.0000001))*180/np.pi
    for x in range(gradAng.shape[0]):
        for y in range(gradAng.shape[1]):
            if gradAng[x,y]<0:gradAng[x,y]+=360
    return gradMag,gradAng
```

需要注意的是，这里只是单层图片，在选择高斯模糊的正态分布标准差σ时，刚开始我选择0或论文推荐的1.6，但效果都不好，和同学讨论后选定经验值1，普适值有待探索。另外，高斯模糊后的结果应*1.0以float化，否则误差很大。

- **主方向确定**

对于每一个角点的坐标，对其周围16*16的区域进行梯度方向统计。这里16*16的选定只是PPT教程里的经验参考，并不是论文中的普适区域选法（由尺度值σ确定半径）。邻域内每个点的方向分到36份，权重为梯度幅值，进行直方图统计。最终总权重值最大的梯度角作为该关键点的主方向。为了简便，这里没有考虑80%辅方向的情况。

```python
def findOri(gradMag,gradAng,corners):
    oriList=[]
    for corner in corners:
        #注意：角点的表示是先列再行，与np中表示相反
        x=int(corner[0][1])
        y=int(corner[0][0])
        hist=np.zeros(36,dtype=float)
        try:
            subMag=np.array(gradMag[x-8:x+8,y-8:y+8])
            subAng=np.array(gradAng[x-8:x+8,y-8:y+8])
            for i in range(16):
                for j in range(16):
                    mag=subMag[i,j]
                    ang=subAng[i,j]
                    hist[int(ang/10)]+=mag
                    #分成36份,且排除掉边界
        except:continue
        mainOri=np.where(hist==np.max(hist))[0][0]*10
        oriList.append([x,y,mainOri])
    return oriList
```

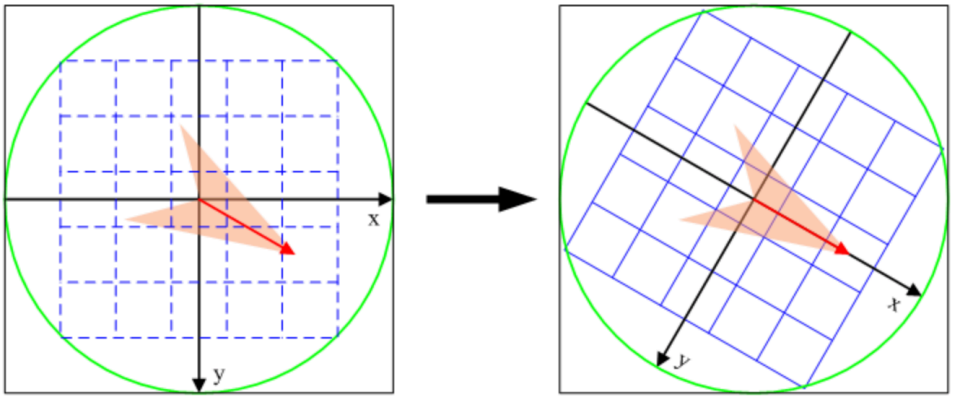


获取主方向

- **物体坐标系**

现在我们有了角点的坐标、主方向，就可以根据该主方向在角点周围建立物体坐标系。物体坐标系中，中心点(x,y)偏移(detx,dety)后的点的坐标，在图像坐标系内的坐标可通过以下旋转函数实现：

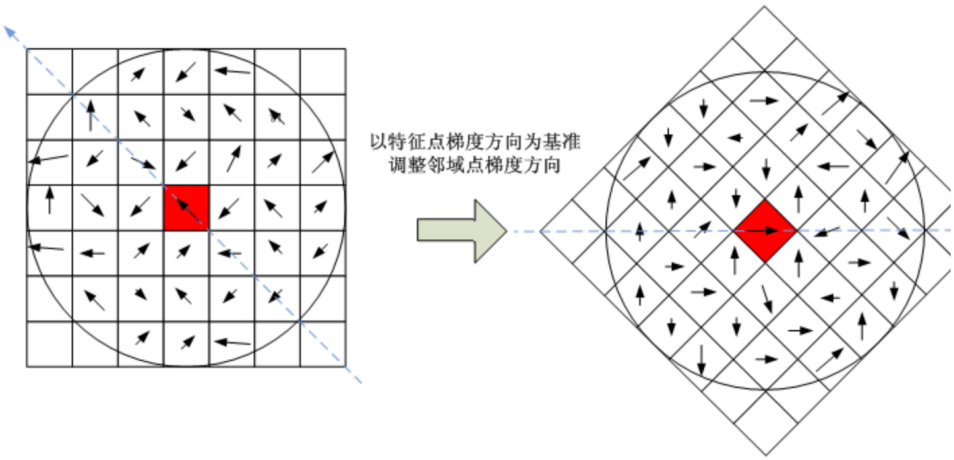
```python
def rotateCoor(x,y,mainOri,detx,dety):
#物体坐标系内经过detx,dety后的点的图像坐标系坐标,可能越界,后面要检查
    r=math.hypot(detx,dety)
    detAng=mainOri+math.atan2(dety,detx+0.0000001)*180.0/math.pi
    if detAng<0:detAng+=360.0
    if detAng>360:detAng-=360.0
    newx=x*1.0+r*math.cos(detAng/180.0*math.pi)
    newy=y*1.0+r*math.sin(detAng/180.0*math.pi)
    return newx,newy
```



坐标系变换

- **SIFT特征向量**

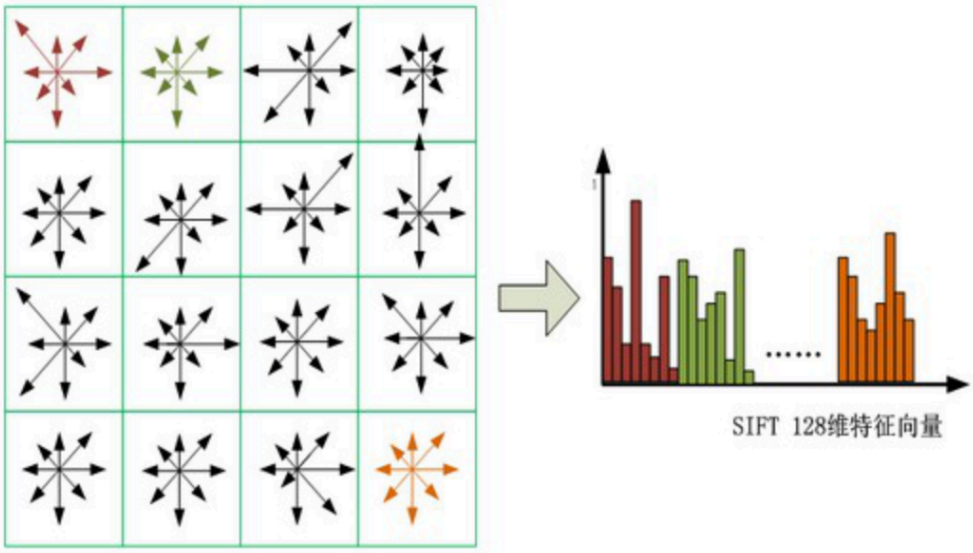
通过上一步的坐标旋转函数，我们可以构建角点主方向系内的邻域，如下图所示:



将物体坐标系内的16*16邻域分为4*4个块，每个块4*4个像素，对每个像素点采用和上一步类似的方法求梯度方向直方图。有两点需要注意，一是现在像素点坐标不是整数，因此采用双线性插值的方法获得近似的幅值和梯度角，二是求出的梯度角需要减去主方向。

```python
def findAllSift(img,gradMag,gradAng,oriList):
    kps,des=[],[]
    for key in oriList:
        x,y,mainOri=key[0],key[1],key[2]
        oneSift=[]
        xstarts=[-8,-4,0,4]
        ystarts=[-8,-4,0,4]
        for xstart in xstarts:
            for ystart in ystarts:
                hist=[0.0]*8
                for i in range(4):
                    for j in range(4):
                        detx,dety=xstart+i,ystart+j
                        newx,newy=rotateCoor(x,y,mainOri,detx,dety)
                        if newx<0 or newy<0 or newx>img.shape[0]-2 or newy>img.shape[1
                            continue
                        intx,inty=int(newx),int(newy)
                        dx1,dx2,dy1,dy2=newx-intx,1+intx-newx,newy-inty,inty+1-newy
                        ang=gradAng[intx,inty]*dx2*dy2+gradAng[intx+1,inty]*dx1*dy2+gr
                        ang-=mainOri
                        if ang<0:ang+=360.0
                        pixel=img[intx,inty]*dx2*dy2+img[intx+1,inty]*dx1*dy2+img[intx
                        #pixel=gradMag[intx,inty]*dx2*dy2+gradMag[intx+1,inty]*dx1*dy2
                        hist[int(ang/45)]+=pixel
                oneSift+=hist
        sum=0.0
        for value in oneSift:
            sum+=value**2
        if sum!=0:oneSift=np.array(oneSift)/math.sqrt(sum)
        des.append(oneSift)
        kps.append(cv2.KeyPoint(y,x,_size=gradMag[x,y]))
    return kps,np.array(des,dtype=np.float32)
```

按照我的理解，在邻域内求梯度角8个方向的直方图时，应该是梯度幅值作为权值投票，但是效果并不好。仔细读论文后发现，权重应该是原图像素的灰度值，而不是梯度幅值。对此我的理解是已经确定了角点和主方向，这时用原图像素为相对梯度角投票可以更好地模拟该角点的全面特征（包括0阶的明暗和1阶的轮廓方向），而不是只有轮廓。另外，既然使用了原图像素投票，最后应该对128维sift特征向量进行归一化，消除光照不均匀的影响。



128维向量的构建

- **特征点匹配**

通过上面的方法，我们可以对一张图的每个角点建立其sift128维特征向量。根据CNN可解释性相关研究，我认为sift向量主要描述的是texture特征。两个角点可能在不同的图片中，方向不同、尺度不同，如何判定这两个角点相似？用sift向量的内积表示。

这是我最初的配对方法，复杂度为$O(N^2)$，并且判断相似需要用到经验阈值：

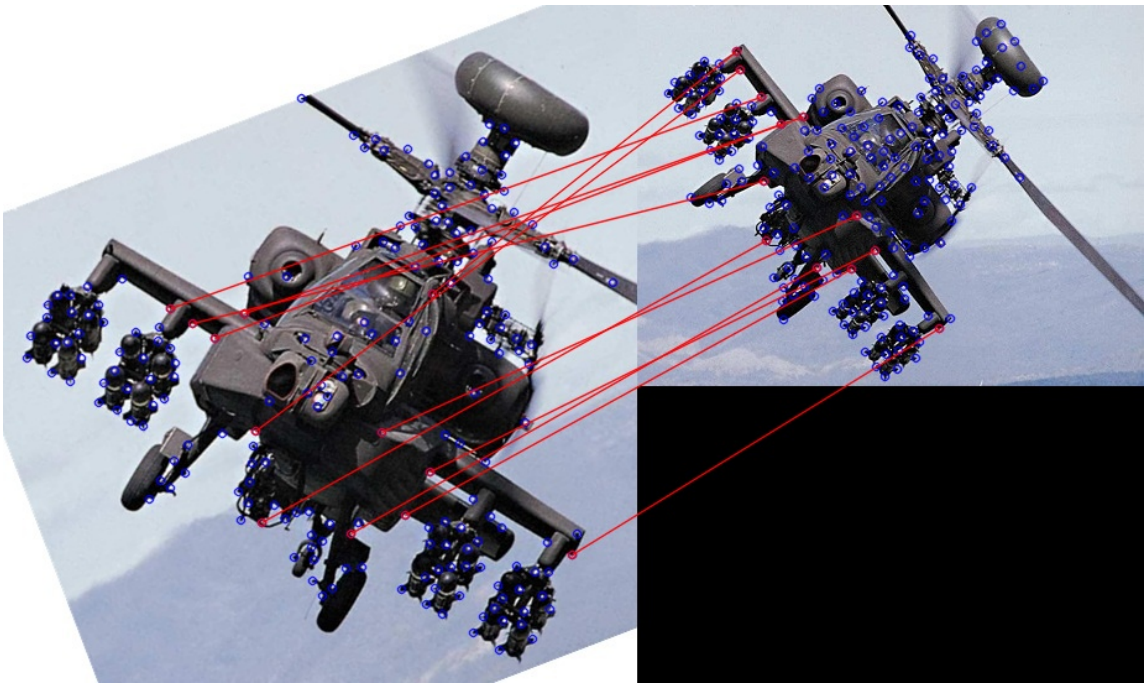
```
...
for sift1 in allSift1:
    max=0
    x1,y1,mainOri1,sift1Vec=sift1[0],sift1[1],sift1[2],sift1[3]
    x2tmp,y2tmp=0,0
    for sift2 in allSift2:
        x2,y2,mainOri2,sift2Vec=sift2[0],sift2[1],sift2[2],sift2[3]
        tmp=sum(sift1Vec*sift2Vec)
        if tmp>max:
            max=tmp
            x2tmp,y2tmp=x2,y2
    if max>0.55:
        matches.append([(x1,y1),(x2tmp,y2tmp)])
    ...
out = np.zeros((rows1,cols1+cols2,3), dtype='uint8')
out[:rows1,:cols1,:] = np.dstack([img1])
out[:rows2,cols1:cols1+cols2,:] = np.dstack([img2])
for match in matches:
    (c1,r1)=match[0]
    (c2,r2)=match[1]
    #color=...
    cv2.circle(out,(c1,r1),2,color,1)
    cv2.circle(out,(c2+cols1,r2),2,color,1)
    cv2.line(out,(c1,r1),(c2+cols1,r2),color,1)

cv2.imshow('Matched KeyPoints', out)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

后来发现可以调用opencv自带的配对方法 `cv2.BFMatcher()` ，该方法需要按照格式打包KeyPoint类的角点list和对应的储存特征向量的des矩阵。实际使用了PCA降维聚类的方法。实例内同样使用经验阈值0.85进行了初筛：

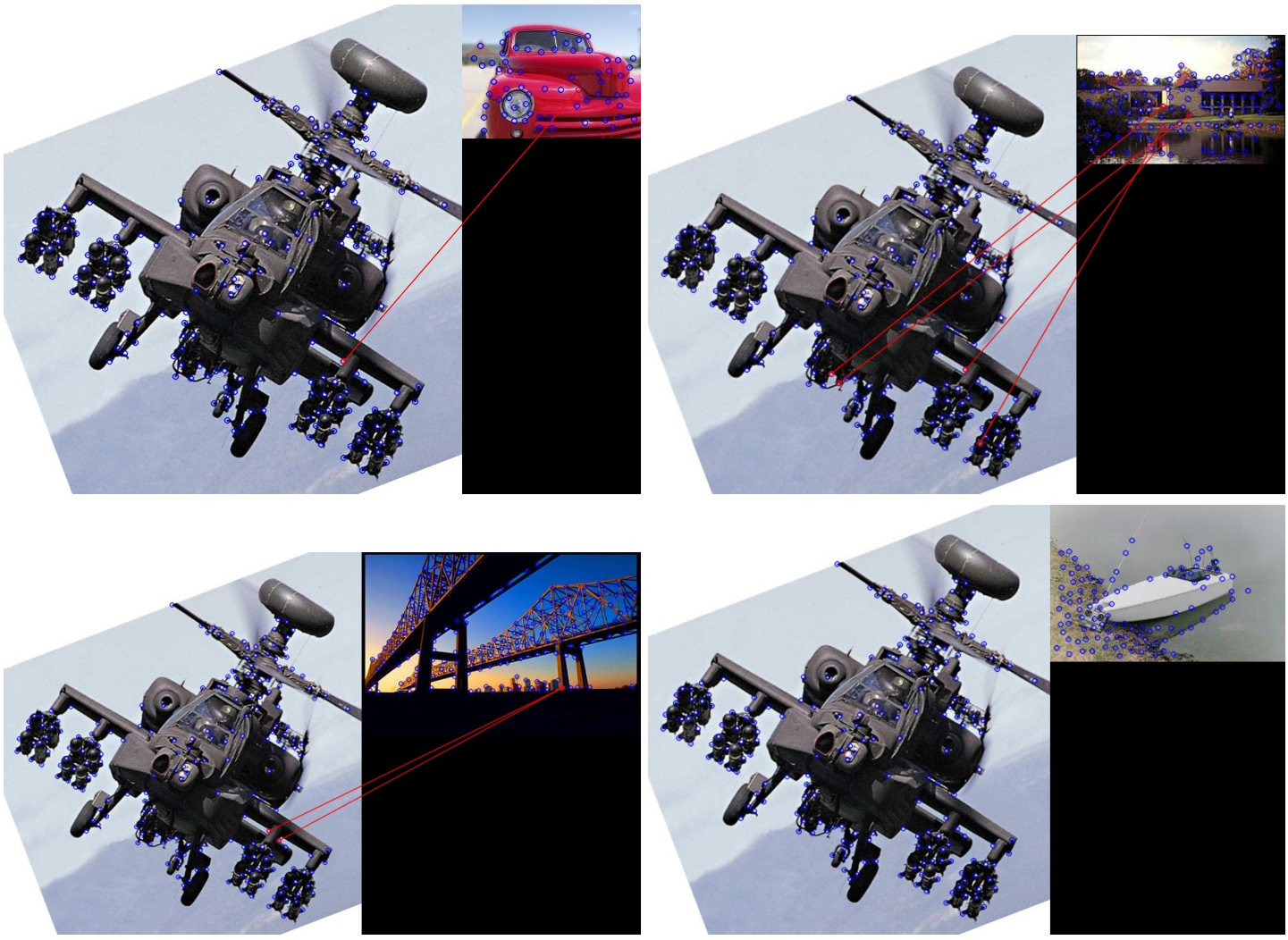
```python
def drawMatches(img1_dir,img2_dir,maxMatches=50):
    img1=cv2.imread(img1_dir,cv2.IMREAD_GRAYSCALE)
    img2=cv2.imread(img2_dir,cv2.IMREAD_GRAYSCALE)

    gradMag1,gradAng1=getGrad(img1)
    gradMag2,gradAng2=getGrad(img2)

    colorimg1=cv2.imread(img1_dir,cv2.IMREAD_COLOR)
    colorimg2=cv2.imread(img2_dir,cv2.IMREAD_COLOR)

    corners1=cv2.goodFeaturesToTrack(img1,maxCorners=maxMatches,qualityLevel=0.01,minD
    corners2=cv2.goodFeaturesToTrack(img2,maxCorners=maxMatches,qualityLevel=0.01,minD

    kps1,des1=findAllSift(img1,gradMag1,gradAng1,findOri(gradMag1,gradAng1,corners1))
    kps2,des2=findAllSift(img2,gradMag2,gradAng2,findOri(gradMag2,gradAng2,corners2))

    bf=cv2.BFMatcher()
    matches=bf.knnMatch(des1,des2,k=2)
    nice_match=[]
    for m,n in matches:
        if m.distance<0.8*n.distance:
            nice_match.append([m])
    M=max(colorimg1.shape[0],colorimg2.shape[0])
    N=colorimg1.shape[1]+colorimg2.shape[1]
    img_match=np.zeros((M, N))
    img_match=cv2.drawMatchesKnn(colorimg1,kps1,colorimg2,kps2,
        nice_match, img_match, matchColor=[0,0,255], singlePointColor=[255,0,0])
    cv2.imshow("MyMatch", img_match)
    cv2.waitKey(0)
```

至此整个第二部分，也就是实现旋转不变性已经完成。实际coding过程中，有一个很烦的问题就是x和y到底是哪个。比如画图、返回角点的坐标均是(col,row)，但是这个坐标在numpy读图像矩阵时却应该表示为[row,col]，需要细心。

- **结果展示**



为观察清晰，只算出较少角点且设较高的比对阈值，图中仅1个点配对错误，准确率90%以上



对比图

3.图像金字塔

- **尺度空间表达**

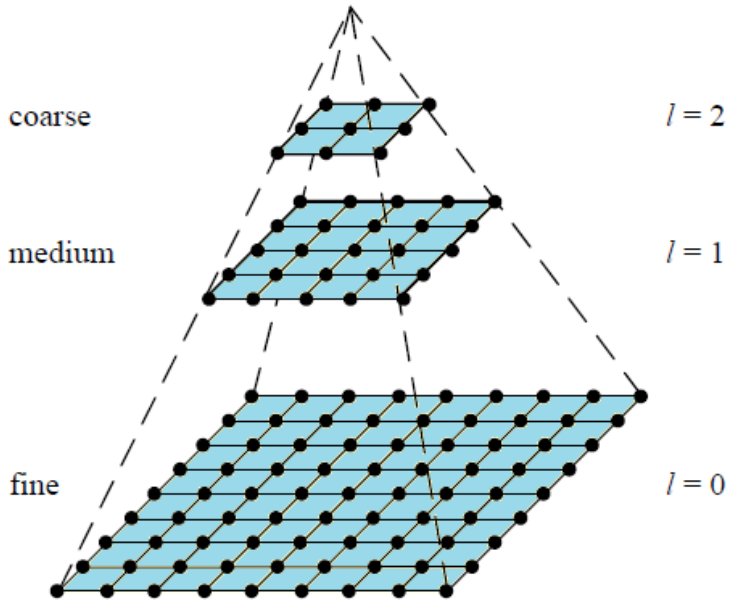
一个图像的尺度空间L(x,y,σ),定义为原始图像I(x,y)与一个可变尺度的2维高斯函数G(x,y,σ)卷积运算。

> It has been shown by Koenderink (1984) and Lindeberg (1994) that under a variety of reasonable assumptions the only possible scale-space kernel is the Gaussian function. There- fore, the scale space of an image is defined as a function, L(x, y, σ), that is produced from the convolution of a variable-scale Gaussian, G(x, y, σ), with an input image, I(x, y):
>
> $$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y),$$
>
> where $*$ is the convolution operation in x and y, and
>
> $$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

“尺度空间表达”指的是不同高斯核所平滑后的图片的不同表达。经过不同高斯核平滑后的图像分辨率相同，但是模糊程度有所区别。

- **图像金字塔化**

若对一张图片进行降采样，其像素点就会减少，图片尺寸也会随之变小。所谓图像金字塔化：就是先进行图像平滑，再进行降采样，根据降采样率不同，所得到一系列尺寸逐渐减小的图像。

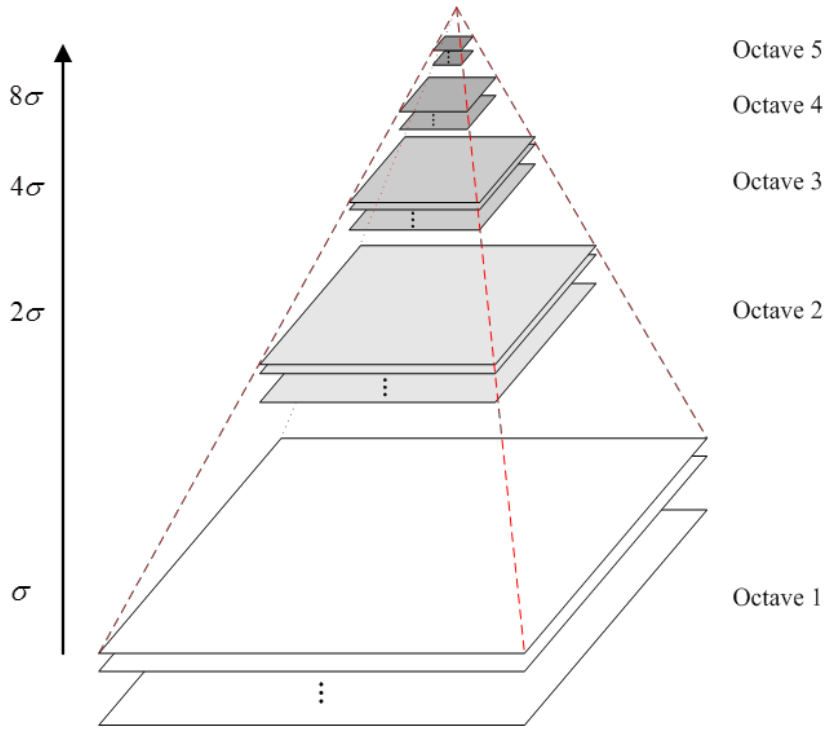


- **LOG与DOG**

以上的两种表达方法各有优劣：

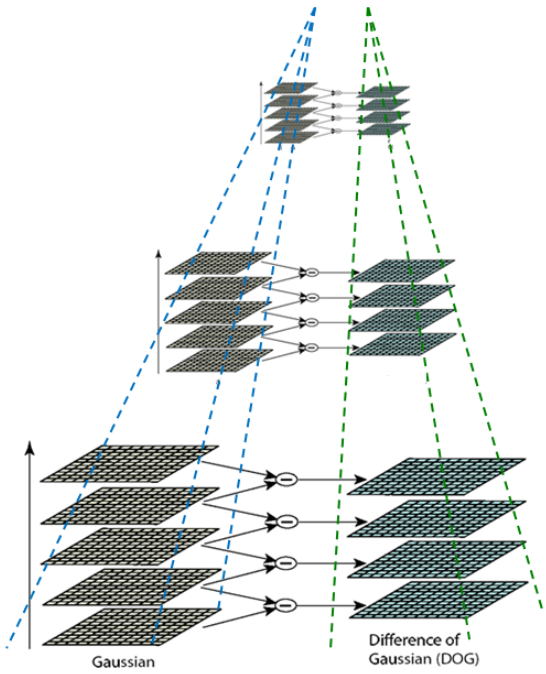
> 尺度空间表达”在所有尺度上具有相同分辨率，而“图像金字塔化”在每层的表达上分辨率都会减少固定比率。
>
> 图像金字塔化”处理速度快，占用存储空间小，而“尺度空间表达”刚好相反。

那么将两者融合起来的话，就得到了LOG图像，高斯拉普拉斯变换图像(Laplassian of Gaussian）。先将照片降采样，得到了不同分辨率下的图像金字塔。再对每层图像进行高斯卷积。这样一来，原本的图像金字塔每层只有一张图像，而卷积后，每层又增加了多张不同模糊程度下的照片。



而对LOG进一步优化以便更好地获取特征点，我们得到DOG图像，高斯差分（Difference of Gaussian）。在获得LOG图像后，用其相邻的图像进行相减，得到所有图像重新构造的金字塔就是DOG金字塔。数学表达为：

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma).$$



首先获取一张图的多个尺度下的金字塔。根据论文中BBF段，相邻层间图片尺寸相差0.8倍。且将图片尺寸限制在315-960的范围内，这样可以使后续运算耗时不过长，并且金字塔的octave层数在5左右：

```
def getIMGpyramid(img):
    res=[img]
    M=img.shape[0]
    N=img.shape[1]
    minMN=min(M,N)
    maxMN=max(M,N)
    while minMN>=315:
        nxt=cv2.resize(res[-1],(0,0),fx=0.8,fy=0.8)
        res.append(nxt)
        M=res[-1].shape[0]
        N=res[-1].shape[1]
        minMN=min(M,N)
    while maxMN<=960:
        nxt=cv2.resize(res[0],(0,0),fx=1.25,fy=1.25)
        res.insert(0,nxt)
        M=res[0].shape[0]
        N=res[0].shape[1]
        maxMN=max(M, N)
    return res
```

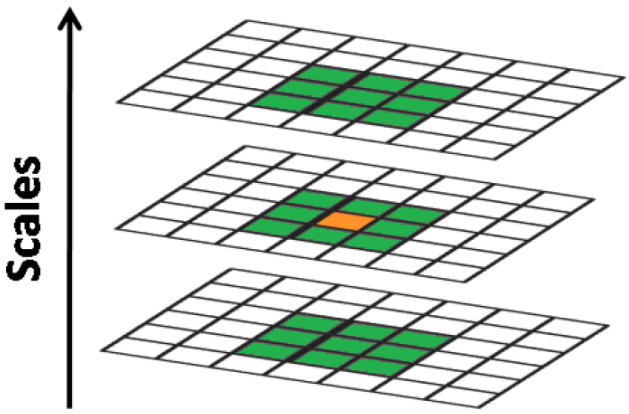
然后对一层（金字塔中一个octave）图像获取DOG。也就是由一张原图得到5个不同高斯平滑LOG，再相减得到4个DOG：

```
def getLaps(img,s=1.6,k=sqrt(2)):
    L5 = cv2.GaussianBlur(img,(5,5),s*(k**4))#Laplace of Gaussian Layers
    L4 = cv2.GaussianBlur(img,(5,5),s*(k**3))
    L3 = cv2.GaussianBlur(img,(5,5),s*(k**2))
    L2 = cv2.GaussianBlur(img,(5,5),s*k)
    L1 = cv2.GaussianBlur(img,(5,5),s)
    return L5,L4,L3,L2,L1

def getDogs(L5,L4,L3,L2,L1):
    DOG4 = array(L5-L4)#Difference of Gaussian layers
    DOG3 = array(L4-L3)
    DOG2 = array(L3-L2)
    DOG1 = array(L2-L1)
    return DOG1,DOG2,DOG3,DOG4
```

其中高斯核的正态分布标准差σ为$s, sk, ..., sk^4$，按照论文给出的值，s取1.6，k取$\sqrt{2}$。

- **寻找DOG极值点**

对于某一层DOG图像的某个像素，将其与其周围的像素点比较，当其大于或者小于所有相邻点时，即为极值点。由于需要找三个方向的极值点，一个octave内的4层dog，只能取dog2，dog3这两层：




```
def findExt(up,src,low):#respectively upper, itself, lower dog layers
    corExts=[]
    for i in range(1,src.shape[0]):
        for j in range(1,src.shape[1]):
            cmp=(src[i-1][j+1],src[i][j+1],src[i+1][j+1],src[i-1][j],src[i+1][j],src[i
                    up[i-1][j+1],up[i][j+1],up[i+1][j+1],up[i-1][j],up[i][j],up[i+1][j
                    low[i-1][j+1],low[i][j+1],low[i+1][j+1],low[i-1][j],low[i][j],low[
            if max(cmp)<src[i][j] or min(cmp)>src[i][j]:
                corExts.append([i,j])
    return corExts
```

- **极值点精确定位和筛选**

通过以上方法找到的极值点是离散的具体点。但实际上需要精确定位，转为连续，解泰勒展开：

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}} + \frac{1}{2}\mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}$$

得

$$\hat{\mathbf{x}} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1} \frac{\partial D}{\partial \mathbf{x}}$$

由于精确定位实现较为繁琐，实际求了某像素三个方向的梯度和二阶梯度，并且后面需要用三个方向的线性插值，在这里我没有代码实现，而是只用离散求得的粗略极值点。

但是目前求得的极值点可能是图像边缘区域产生的，要去除这些边缘点。使用了Hessian矩阵，也是Harris角点检测算法使用的方法。海塞矩阵是用来求曲率的，以函数的二阶偏导为元素:

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

$$\mathrm{Tr}(\mathbf{H}) = D_{xx} + D_{yy} = \alpha + \beta,$$
$$\mathrm{Det}(\mathbf{H}) = D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta.$$

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta^2} = \frac{(r+1)^2}{r},$$

若满足

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} < \frac{(r+1)^2}{r}.$$

则认为该点是边缘点。论文中r取10：

```python
def EliminateEdge(corExts,src):#src is the middle dog layer
    keyExts=[]
    for i in range(len(corExts)):
        x=corExts[i][0]
        y=corExts[i][1]
        rx,ry=x+1,y+1
        if src[rx][ry]==0:
            continue
        if rx+1<src.shape[0] and ry+1<src.shape[1]:
            fxx=src[rx-1][ry]+src[rx+1,ry]-2*src[rx,ry]
            fyy=src[rx][ry-1]+src[rx,ry+1]-2*src[rx,ry]
            fxy=src[rx-1][ry-1]+src[rx+1][ry+1]-src[rx-1][ry+1]-src[rx+1][ry-1]
            trace=fxx+fyy #for Hessian matrix
            det=fxx*fyy-fxy*fxy
            if trace*trace/det>=12.1: #let ro=10
                keyExts.append(corExts[i])
    return keyExts
```

通过以上的部分就初步实现了图像金字塔。若有5个octave，则可以得到5*2层dog图层内的角点。

看论文和网上的实践，有些是比较两张图时，分别比较10个dog层之间相互的match，取nice_match最多的一层，也就是要比较100次，这也从侧面反映出了sift算法在特征点检测时具有不能实时性的缺点。

而另外一些实践方法，算出了10层的角点，却只取其中一层（一般是dog1_2层）去算。这也是较主流的方法，比较快。但是这就让人思考——那为何要算出金字塔的多个octave呢？只算一个octave，然后在求特征向量时不都用16*16，而是用和尺度有关的值来缩放邻域大小，这样不就既可以满足尺度不变性，又可以使计算精简吗？

当然，图像特征提取不只有sift，目前较常用的还有HOG, LBP, Haar等，这些或许会更好地满足实时性（计算精简）的要求。比如就我所知，目前行人动态识别算法大都是基于HOG+SVM实现的。