**目标：**

* 了解光流的概念及 Lucas-Kanade 光流法。
* 使用cv.calcOpticalFlowPyrLK()函数来跟踪视频中的特征点。

## api：

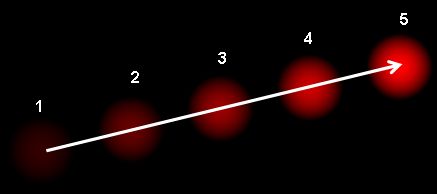
**稀疏光流Lucas-Kanade Optical Flow**

nextPts, status, err = cv.calcOpticalFlowPyrLK( prevImg, nextImg, prevPts, nextPts[, status[, err[, winSize[, maxLevel[, criteria[, flags[, minEigThreshold]]]]]]] )

**稠密光流Dense Optical Flow**

flow = cv.calcOpticalFlowFarneback( prev, next, flow, pyr\_scale, levels, winsize, iterations, poly\_n, poly\_sigma, flags)

**光流**

由于目标对象或者摄像机的移动造成的图像对象在连续两帧图像中的移动被称为光流。它是一个 2D 向量场，可以用来表示一个点从第一帧图像到第二帧图像之间的移动。如下图所示。 [](https://camo.githubusercontent.com/699b9a29b4c0bbdb5e6b371d626e239fd50a17616e7f5317eb95802ff3e031d6/68747470733a2f2f646f63732e6f70656e63762e6f72672f342e302e302f6f70746963616c5f666c6f775f6261736963312e6a7067)

上图显示了一个球在连续的五帧图像间的移动。箭头表示其位移向量。光流在很多领域中都很有用：

* 由运动重建结构
* 视频压缩
* 视频防抖

光流是基于以下假设下工作的：

1. 在连续的两帧图像之间，目标对象的像素的灰度值不改变。
2. 相邻像素具有相似的运动。

考虑第一帧中的像素 $$ I(x,y,t) ，它在时间之后移动距离，它在𝑑𝑡时间之后移动距离 (dx,dy) $$。根据第一条假设：灰度值不变。所以我们可以得到：

𝐼(𝑥,𝑦,𝑡)=𝐼(𝑥+𝑑𝑥,𝑦+𝑑𝑦,𝑡+𝑑𝑡)

然后对等号右侧采用泰勒级数展开，删除相同项并两边除以dt得到以下等式：

𝑓𝑥𝑢+𝑓𝑦𝑣+𝑓𝑡=0;

其中：

𝑓𝑥=𝜕𝑓𝜕𝑥;;;𝑓𝑦=𝜕𝑓𝜕𝑦

𝑢=𝑑𝑥𝑑𝑡;;;𝑣=𝑑𝑦𝑑𝑡

上边的公式叫做光流方程。其中 $$ f\_x $$ 和 $$ f\_y $$ 是图像梯度，同样 $$ f\_t $$ 是时间方向的梯度。但（u，v）是不知道的。我们不能在一个等式中求解两个未知数，有几个方法可以帮我们解决这个问题，其中的一个是 Lucas-Kanade 法。

**Lucas-Kanade方法**

现在我们要使用第二条假设，邻域内的所有点都有相似的运动。LucasKanade 法就是利用一个 3x3 邻域中的 9 个点具有相同运动的这一点。这样我们就可以找到$$ (f\_x, f\_y, f\_t) $$这 9 个点的光流方程，用它们组成一个具有两个未知数 9 个等式的方程组，这是一个约束条件过多的方程组。一个好的解决方法就是使用最小二乘拟合。下面就是求解结果：

[𝑢 𝑣]=[∑𝑖𝑓𝑥𝑖2∑𝑖𝑓𝑥𝑖𝑓𝑦𝑖 ∑𝑖𝑓𝑥𝑖𝑓𝑦𝑖∑𝑖𝑓𝑦𝑖2]−1[−∑𝑖𝑓𝑥𝑖𝑓𝑡𝑖 −∑𝑖𝑓𝑦𝑖𝑓𝑡𝑖]

（你会发现上边的逆矩阵与 Harris 角点检测器非常相似，这说明角点很适合被用来做跟踪） 从使用者的角度来看，想法很简单，我们去跟踪一些点，然后我们就会获得这些点的光流向量。但是还有一些问题。直到现在我们处理的都是很小的运动。如果有大的运动怎么办呢？图像金字塔。当我们进入金字塔时，小运动被移除，大运动变成小运动。因此，通过在那里应用Lucas-Kanade，我们就会得到尺度空间上的光流。

**Lucas-KanadeOpenCV中的Lucas-Kanade光流**

上述所有过程都被 OpenCV 打包成了一个函数cv2.calcOpticalFlowPyrLK()。现在我们使用这个函数创建一个小程序来跟踪视频中的一些点。我们使用函数 cv2.goodFeatureToTrack() 来确定要跟踪的点。我们首先在视频的第一帧图像中检测一些 Shi-Tomasi 角点，然后我们使用 LucasKanade 算法迭代跟踪这些角点。我们要给函数 cv2.calcOpticlaFlowPyrLK()传入前一帧图像和其中的点，以及下一帧图像。函数将返回带有状态数的点，如果状态数是 1，那说明在下一帧图像中找到了这个点（上一帧中角点），如果状态数是 0，就说明没有在下一帧图像中找到这个点。我们再把这些点作为参数传给函数，如此迭代下去实现跟踪。

**稀疏光流Lucas-Kanade Optical Flow**

* 函数原型：  
  nextPts, status, err = cv.calcOpticalFlowPyrLK( prevImg, nextImg, prevPts, nextPts[, status[, err[, winSize[, maxLevel[, criteria[, flags[, minEigThreshold]]]]]]] )
* prevImg：第一帧8bit灰度图像，或者是buildOpticalFlowPyramid创建的金字塔图像
* nextImg：第二帧8bit灰度图像，或者是与prevImg相同尺寸与类型的金字塔图像
* prevPts：光流法需要找到的二维点的vector。点坐标必须是单精度浮点数
* nextPts：包含输入特征在第二幅图像中计算出的新位置的二维点（单精度浮点坐标）的输出vector。当使用OPTFLOW\_USE\_INITIAL\_FLOW 标志时，nextPts的vector必须与input的大小相同
* status：输出状态vector，类型：unsigned chars；如果找到了对应特征的流，则将向量的每个元素设置为1，否则置0
* err：误差输出vector。vector的每个元素被设置为对应特征的误差，可以在flags参数中设置误差度量的类型；如果没有找到流，则未定义误差（使用status参数来查找此类情况）
* winSize：每级金字塔的搜索窗口大小
* maxLevel：最大金字塔层次数，如果设置为0，则不使用金字塔（单级）；如果设置为1，则使用两个级别，等等。如果金字塔被传递到input，那么算法使用的级别与金字塔同级别但不大于MaxLevel。
* criteria：指定迭代搜索算法的终止准则；在指定的最大迭代次数标准值criteria.maxCount之后，或者当搜索窗口移动小于criteria.epsilon
* flags：操作标志，可选参数：  
  OPTFLOW\_USE\_INITIAL\_FLOW：使用初始估计，存储在nextPts中；如果未设置标志，则将prevPts复制到nextPts并被视为初始估计。  
  OPTFLOW\_LK\_GET\_MIN\_EIGENVALS：使用最小本征值作为误差度量（见minEigThreshold描述）；如果未设置标志，则将原始周围的一小部分和移动的点之间的 L1 距离除以窗口中的像素数，作为误差度量
* minEigThreshold：算法所计算的光流方程的2x2标准矩阵的最小本征值（该矩阵称为[Bouguet00]中的空间梯度矩阵）÷ 窗口中的像素数。如果该值小于MinEigThreshold，则过滤掉相应的特征，相应的流也不进行处理。因此可以移除不好的点并提升性能。

## 实例1

|  |  |
| --- | --- |
|  | **import cv2 import numpy as np  cap = cv2.VideoCapture('../datas/slow.mp4') *# feature params* feature\_params = dict(  maxCorners=100,  qualityLevel=0.3,  minDistance=7,  blockSize=7 ) *# Parameters for lucas kanade optical flow* Ik\_params = dict(  winSize=(15,15),  maxLevel=2,  criteria=(cv2.TERM\_CRITERIA\_EPS|cv2.TERM\_CRITERIA\_COUNT,10,0.03) ) *# Create some random colors* color = np.random.randint(0,255,(100,3)) ret,old\_frame = cap.read() *# 读取一帧* old\_gray = cv2.cvtColor(old\_frame,cv2.COLOR\_BGR2GRAY) *# 转化为灰度图 # 使用Shi-tomas角点检测* p0 = cv2.goodFeaturesToTrack(old\_gray,mask=None,\*\*feature\_params) *# Create a mask image for drawing purposes* mask = np.zeros\_like(old\_frame)  *#* while True:  ret,frame = cap.read()  gray\_frame = cv2.cvtColor(frame,cv2.COLOR\_BGR2GRAY)  *# 计算光流* p1,st,err = cv2.calcOpticalFlowPyrLK(old\_gray,gray\_frame,p0,None,\*\*Ik\_params)  *# Select good points* good\_new = p1[st==1]  good\_old = p0[st==1]  *# draw the tracks* for i,(new,old) in enumerate(zip(good\_new,good\_old)):  a,b = new.ravel()  c,d = old.ravel()  mask = cv2.line(mask,(a,b),(c,d), color[i].tolist(),2)  frame = cv2.circle(frame,(a,b),5,color[i].tolist(),-1)  img = cv2.add(frame,mask)  cv2.imshow("result", img)  k = cv2.waitKey(30) & 0xff  if k==27:  break  old\_gray = gray\_frame.copy()  *# p0 = good\_new.reshape(-1, 1, 2)* p0 = p1 cv2.destroyAllWindows() cap.release()** |

### 效果：

|  |
| --- |
| image7 |

## OpenCV中的密集光流

Lucas-Kanade 法是计算稀疏特征集的光流（上面的例子使用Shi-Tomasi 算法检测角点）。OpenCV 还提供了一种计算稠密光流的方法，它会计算图像中的所有点的光流。这是基于 Gunner\_Farneback 的算法，2003年Gunner Farneback在“Two-Frame Motion Estimation Based on Polynomial Expansion”中对该算法进行了解释。 下面的例子就是使用上面的算法计算稠密光流。结果是一个带有光流向量（u，v）的双通道数组。通过计算我们能得到光流的大小和方向。我们使用颜色对结果进行编码以便于更好的观察。方向对应于 H（Hue）通道，大小对应于 V（Value）通道。

**稠密光流Dense Optical Flow**

CalcOpticalFlowFarneback()函数是利用用Gunnar Farneback的算法计算全局性的稠密光流算法（即图像上所有像素点的光流都计算出来），由于要计算图像上所有点的光流，故计算耗时，速度慢。

* 函数原型：flow = cv.calcOpticalFlowFarneback( prev, next, flow, pyr\_scale, levels, winsize, iterations, poly\_n, poly\_sigma, flags)，参数含义与上一函数类似。
* prev：第一帧单通道8bit灰度图像，
* next：第二帧8bit灰度图像，或者是与prev相同尺寸与类型的金字塔图像
* flow：输出的光流图像，与prev图像尺寸相同，类型是CV\_32FC2.
* pyr\_scale：上下两层金字塔图像的尺寸比例， pyr\_scale=0.5，表示每层减小一半
* levels：金字塔层数，包括初始图像； levels=1 表示不产生金字塔图像，只有原始图像
* winsize：均值窗口大小，越大越能denoise并且能够检测快速移动目标，但会引起模糊运动区域
* iterations：每层金字塔的迭代次数
* poly\_n：像素邻域大小，一般为5，7等，较大的值意味着图像将以更平滑的表面进行近似处理产生更粗犷的算法和更模糊的运动场
* poly\_sigma：高斯标注差，一般为1-1.5，or poly\_n=5, you can set poly\_sigma=1.1, for poly\_n=7, a good value would be poly\_sigma=1.5.
* flags：计算方法。主要包括OPTFLOW\_USE\_INITIAL\_FLOW和OPTFLOW\_FARNEBACK\_GAUSSIAN

## 实例2

|  |  |
| --- | --- |
|  | **import cv2 import numpy as np  cap = cv2.VideoCapture('../datas/vtest.avi') ret,frame1 = cap.read() prvs = cv2.cvtColor(frame1,cv2.COLOR\_BGR2GRAY) hsv = np.zeros\_like(frame1) hsv[...,1] = 255 while True:  ret,frame2 = cap.read()  next =cv2.cvtColor(frame2,cv2.COLOR\_BGR2GRAY)  *# 计算光流* flow = cv2.calcOpticalFlowFarneback(prvs,next,None,0.5,3,15,  3,5,1.2,0)  mag,ang = cv2.cartToPolar(flow[...,0],flow[...,1])  hsv[...,0] = ang\*180/np.pi/2  hsv[...,2] = cv2.normalize(mag,None,0,255,cv2.NORM\_MINMAX)  bgr = cv2.cvtColor(hsv,cv2.COLOR\_HSV2BGR)  cv2.imshow('frame2', bgr)  k = cv2.waitKey(30) & 0xff  if k==27:  break  elif k == ord('s'):  cv2.imwrite('opticalfb.png', frame2)  cv2.imwrite('opticalhsv.png', bgr)  prvs = next  cap.release() cv2.destroyAllWindows()** |

### 效果：上面的是某一个帧，下面的是光流效果

|  |
| --- |
| image8 |