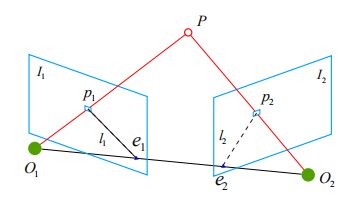
对极约束求解相机运动

# Ubuntu14.04+opencv3.4.6：对极约束求解相机运动

分类专栏： SLAM SLAM学习



以上图为例，我们希望求取两帧图像 I1, I2 之间的运动，设第一帧到第二帧的运动为 R, t。两个相机中心分别为 O1, O2。现在，考虑 I1 中有一个特征点 p1，它在 I2 中对应着特征点 p2。我们晓得这俩是通过特征匹配得到的。如果匹配正确，说明它们确实是同一个 空间点在两个成像平面上的投影。

连线O1p1 和连线 O2p2 在三维空间中会相交于点 P；

候点 O1, O2, P 三个点可以 确定一个平面，称为极平面（Epipolar plane）；

O1O2 连线与像平面 I1, I2 的交点分别 为 e1, e2。e1, e2，称为极点（Epipoles）；

O1O2 被称为基线（Baseline）；

极平面与 两个像平面 I1, I2 之间的相交线 l1, l2 为极线（Epipolar line）；

根据对及约束推导出本征矩阵和基础矩阵（参考：https://blog.csdn.net/zfjBIT/article/details/95965812）

基础矩阵（Fundamental Matrix）F 和本质矩阵（Essential Matrix）E，可以进一步简化对极约束：

对极约束简洁地给出了两个匹配点的空间位置关系。于是，相机位姿估计问题变为以下两步：

1. 根据配对点的像素位置，求出 E 或者 F；

2. 根据 E 或者 F，求出 R, t。

由于 E 和 F 只相差了相机内参，而内参在 SLAM 中通常是已知的，所以实践当中往往使用形式更简单的 E。

测试示例：

使用匹配好的特征点来计算 E,F 和 H，进而分解 E 得到 R, t。

#include <iostream>

#include <opencv2/core/core.hpp>

#include <opencv2/features2d/features2d.hpp>

#include <opencv2/highgui/highgui.hpp>

#include <opencv2/calib3d/calib3d.hpp>

// #include "extra.h" // use this if in OpenCV2

using namespace std;

using namespace cv;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 本程序演示了如何使用2D-2D的特征匹配估计相机运动

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void find\_feature\_matches (

const Mat& img\_1, const Mat& img\_2,

std::vector<KeyPoint>& keypoints\_1,

std::vector<KeyPoint>& keypoints\_2,

std::vector< DMatch >& matches );

void pose\_estimation\_2d2d (

std::vector<KeyPoint> keypoints\_1,

std::vector<KeyPoint> keypoints\_2,

std::vector< DMatch > matches,

Mat& R, Mat& t );

// 像素坐标转相机归一化坐标

Point2d pixel2cam ( const Point2d& p, const Mat& K );

int main ( int argc, char\*\* argv )

{

if ( argc != 3 )

{

cout<<"usage: pose\_estimation\_2d2d img1 img2"<<endl;

return 1;

}

//-- 读取图像

Mat img\_1 = imread ( argv[1], CV\_LOAD\_IMAGE\_COLOR );

Mat img\_2 = imread ( argv[2], CV\_LOAD\_IMAGE\_COLOR );

vector<KeyPoint> keypoints\_1, keypoints\_2;

vector<DMatch> matches;

find\_feature\_matches ( img\_1, img\_2, keypoints\_1, keypoints\_2, matches );

cout<<"一共找到了"<<matches.size() <<"组匹配点"<<endl;

//-- 估计两张图像间运动

Mat R,t;

pose\_estimation\_2d2d ( keypoints\_1, keypoints\_2, matches, R, t );

//-- 验证E=t^R\*scale

Mat t\_x = ( Mat\_<double> ( 3,3 ) <<

0, -t.at<double> ( 2,0 ), t.at<double> ( 1,0 ),

t.at<double> ( 2,0 ), 0, -t.at<double> ( 0,0 ),

-t.at<double> ( 1,0 ), t.at<double> ( 0,0 ), 0 );

cout<<"t^R="<<endl<<t\_x\*R<<endl;

//-- 验证对极约束

Mat K = ( Mat\_<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );

for ( DMatch m: matches )

{

Point2d pt1 = pixel2cam ( keypoints\_1[ m.queryIdx ].pt, K );

Mat y1 = ( Mat\_<double> ( 3,1 ) << pt1.x, pt1.y, 1 );

Point2d pt2 = pixel2cam ( keypoints\_2[ m.trainIdx ].pt, K );

Mat y2 = ( Mat\_<double> ( 3,1 ) << pt2.x, pt2.y, 1 );

Mat d = y2.t() \* t\_x \* R \* y1;

cout << "epipolar constraint = " << d << endl;

}

return 0;

}

void find\_feature\_matches ( const Mat& img\_1, const Mat& img\_2,

std::vector<KeyPoint>& keypoints\_1,

std::vector<KeyPoint>& keypoints\_2,

std::vector< DMatch >& matches )

{

//-- 初始化

Mat descriptors\_1, descriptors\_2;

// used in OpenCV3

Ptr<FeatureDetector> detector = ORB::create();

Ptr<DescriptorExtractor> descriptor = ORB::create();

// use this if you are in OpenCV2

// Ptr<FeatureDetector> detector = FeatureDetector::create ( "ORB" );

// Ptr<DescriptorExtractor> descriptor = DescriptorExtractor::create ( "ORB" );

Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create ( "BruteForce-Hamming" );

//-- 第一步:检测 Oriented FAST 角点位置

detector->detect ( img\_1,keypoints\_1 );

detector->detect ( img\_2,keypoints\_2 );

//-- 第二步:根据角点位置计算 BRIEF 描述子

descriptor->compute ( img\_1, keypoints\_1, descriptors\_1 );

descriptor->compute ( img\_2, keypoints\_2, descriptors\_2 );

//-- 第三步:对两幅图像中的BRIEF描述子进行匹配，使用 Hamming 距离

vector<DMatch> match;

//BFMatcher matcher ( NORM\_HAMMING );

matcher->match ( descriptors\_1, descriptors\_2, match );

//-- 第四步:匹配点对筛选

double min\_dist=10000, max\_dist=0;

//找出所有匹配之间的最小距离和最大距离, 即是最相似的和最不相似的两组点之间的距离

for ( int i = 0; i < descriptors\_1.rows; i++ )

{

double dist = match[i].distance;

if ( dist < min\_dist ) min\_dist = dist;

if ( dist > max\_dist ) max\_dist = dist;

}

printf ( "-- Max dist : %f \n", max\_dist );

printf ( "-- Min dist : %f \n", min\_dist );

//当描述子之间的距离大于两倍的最小距离时,即认为匹配有误.但有时候最小距离会非常小,设置一个经验值30作为下限.

for ( int i = 0; i < descriptors\_1.rows; i++ )

{

if ( match[i].distance <= max ( 2\*min\_dist, 30.0 ) )

{

matches.push\_back ( match[i] );

}

}

}

Point2d pixel2cam ( const Point2d& p, const Mat& K )

{

return Point2d

(

( p.x - K.at<double> ( 0,2 ) ) / K.at<double> ( 0,0 ),

( p.y - K.at<double> ( 1,2 ) ) / K.at<double> ( 1,1 )

);

}

void pose\_estimation\_2d2d ( std::vector<KeyPoint> keypoints\_1,

std::vector<KeyPoint> keypoints\_2,

std::vector< DMatch > matches,

Mat& R, Mat& t )

{

// 相机内参,TUM Freiburg2

Mat K = ( Mat\_<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );

//-- 把匹配点转换为vector<Point2f>的形式

vector<Point2f> points1;

vector<Point2f> points2;

for ( int i = 0; i < ( int ) matches.size(); i++ )

{

points1.push\_back ( keypoints\_1[matches[i].queryIdx].pt );

points2.push\_back ( keypoints\_2[matches[i].trainIdx].pt );

}

//-- 计算基础矩阵

Mat fundamental\_matrix;

fundamental\_matrix = findFundamentalMat ( points1, points2, CV\_FM\_8POINT );

cout<<"fundamental\_matrix is "<<endl<< fundamental\_matrix<<endl;

//-- 计算本质矩阵

Point2d principal\_point ( 325.1, 249.7 ); //相机光心, TUM dataset标定值

double focal\_length = 521; //相机焦距, TUM dataset标定值

Mat essential\_matrix;

essential\_matrix = findEssentialMat ( points1, points2, focal\_length, principal\_point );

cout<<"essential\_matrix is "<<endl<< essential\_matrix<<endl;

//-- 计算单应矩阵

Mat homography\_matrix;

homography\_matrix = findHomography ( points1, points2, RANSAC, 3 );

cout<<"homography\_matrix is "<<endl<<homography\_matrix<<endl;

//-- 从本质矩阵中恢复旋转和平移信息.

recoverPose ( essential\_matrix, points1, points2, R, t, focal\_length, principal\_point );

cout<<"R is "<<endl<<R<<endl;

cout<<"t is "<<endl<<t<<endl;

}

build$ ./pose\_estimation\_2d2d ../1.png ../2.png

-- Max dist : 95.000000

-- Min dist : 7.000000

一共找到了81组匹配点

fundamental\_matrix is

[5.43545306591195e-06, 0.0001366043242986687, -0.0214089008693648;

-0.0001321142229821413, 2.339475702772435e-05, -0.006332906454719608;

0.0210763035219264, -0.00366683395262244, 1]

essential\_matrix is

[0.007841371796261635, 0.2075176016541406, 0.1181719698273573;

-0.190270235105878, 0.03346203071185214, -0.6710910536624433;

-0.07512093110340777, 0.6702181417805779, 0.0197271235906156]

homography\_matrix is

[0.9131751806530853, -0.10924353082387, 29.95859981704123;

0.02223560379576808, 0.9826008029037508, 6.508910479581645;

-0.0001001560372702294, 0.0001037779474648939, 1]

R is

[0.9980704024700457, -0.05457644989749674, 0.02961220744670522;

0.05354481493884546, 0.9979665375218761, 0.03457951387400768;

-0.03143921924039243, -0.03292720916178488, 0.9989631496658776]

t is

[-0.9411804751897357;

-0.1569351957186449;

0.2992501586739325]

t^R=

[-0.01108937434188349, -0.293474206690423, -0.1671204024221927;

0.2690827470026501, -0.04732245765724633, 0.9490660696766777;

0.1062370395845343, -0.947831585854587, -0.02789836572845881]

————————————————

版权声明：本文为CSDN博主「JoannaJuanCV」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/zfjBIT/article/details/95964830

# OpenCV-Python系列五十对极几何

 本文是全系列中第54 / 63篇：[OpenCV-Python](https://panchuang.net/series/opencv-python/)

在本节中  
– 我们将学习多视图几何的基础知识  
– 我们将了解什么是极点，极线，极线约束等。

基础概念

当我们使用针孔相机拍摄图像时，我们失去了重要信息，即图像深度。 或者图像中的每个点距相机多远，因为它是3D到2D转换。 因此，是否能够使用这些摄像机找到深度信息是一个重要的问题。 答案是使用不止一台摄像机。 在使用两台摄像机（两只眼睛）的情况下，我们的眼睛工作方式相似，这称为立体视觉。 因此，让我们看看OpenCV在此字段中提供了什么。

（通过Gary Bradsky学习OpenCV在该领域有很多信息。）

在深入图像之前，让我们首先了解多视图几何中的一些基本概念。在本节中，我们将讨论对极几何。请参见下图，该图显示了使用两台摄像机拍摄同一场景的图像的基本设置。

如果仅使用左摄像机，则无法找到与图像中的点相对应的3D点，因为线上的每个点都投影到图像平面上的同一点。但也要考虑正确的形象。现在，直线$OX$上的不同点投射到右侧平面上的不同点($x’$)。因此，使用这两个图像，我们可以对正确的3D点进行三角剖分。这就是整个想法。

不同点的投影在右平面$OX$上形成一条线(line$l’$)。我们称其为对应于该点的Epiline。这意味着，要在正确的图像上找到该点，请沿着该轮廓线搜索。它应该在这条线上的某处（以这种方式考虑，可以在其他图像中找到匹配点，而无需搜索整个图像，只需沿着Epiline搜索即可。因此，它可以提供更好的性能和准确性）。这称为对极约束。类似地，所有点在另一幅图像中将具有其对应的Epiline。该平面称为对极面。

$O$和$O’$是相机中心。从上面给出的设置中，您可以看到在点处的左侧图像上可以看到右侧摄像机$O’$的投影。它称为极点。极点是穿过相机中心和图像平面的线的交点。左摄像机的极点也同理。在某些情况下，您将无法在图像中找到极点，它们可能位于图像外部（这意味着一个摄像机看不到另一个摄像机）。

所有的极线都通过其极点。因此，要找到中心线的位置，我们可以找到许多中心线并找到它们的交点。

因此，在节中，我们将重点放在寻找对极线和极线。但是要找到它们，我们需要另外两种成分，即基础矩阵(F)和基本矩阵(E)，基础矩阵包含有关平移和旋转的信息，这些信息在全局坐标中描述了第二个摄像头相对于第一个摄像头的位置。参见下图(图像由Gary Bradsky提供：Learning OpenCV):

[](https://panchuang.net/wp-content/uploads/2020/03/essential_matrix-1.jpg)

但是我们会更喜欢在像素坐标中进行测量，对吧？ 基本矩阵除包含有关两个摄像头的内在信息之外，还包含与基本矩阵相同的信息，因此我们可以将两个摄像头的像素坐标关联起来。（如果我们使用的是校正后的图像，并用焦距除以标准化该点，$F=E$）。简而言之，基本矩阵F将一个图像中的点映射到另一图像中的线（上）。这是从两个图像的匹配点计算得出的。 至少需要8个这样的点才能找到基本矩阵（使用8点算法时）。 选择更多点并使用RANSAC将获得更可靠的结果。

代码

因此，首先我们需要在两个图像之间找到尽可能多的匹配项，以找到基本矩阵。为此，我们将SIFT描述符与基于FLANN的匹配器和比率测试结合使用。

import numpy as np

import cv2 as cv

from matplotlib import pyplot as plt

img1 = cv.imread('myleft.jpg',0) #索引图像 # left image

img2 = cv.imread('myright.jpg',0) #训练图像 # right image

sift = cv.SIFT()

# 使用SIFT查找关键点和描述符

kp1, des1 = sift.detectAndCompute(img1,None)

kp2, des2 = sift.detectAndCompute(img2,None)

# FLANN 参数

FLANN\_INDEX\_KDTREE = 1

index\_params = dict(algorithm = FLANN\_INDEX\_KDTREE, trees = 5)

search\_params = dict(checks=50)

flann = cv.FlannBasedMatcher(index\_params,search\_params)

matches = flann.knnMatch(des1,des2,k=2)

good = []

pts1 = []

pts2 = []

# 根据Lowe的论文进行比率测试

for i,(m,n) in enumerate(matches):

if m.distance < 0.8\*n.distance:

good.append(m)

pts2.append(kp2[m.trainIdx].pt)

pts1.append(kp1[m.queryIdx].pt)

现在，我们获得了两张图片的最佳匹配列表。 让我们找到基本面矩阵。

pts1 = np.int32(pts1)

pts2 = np.int32(pts2)

F, mask = cv.findFundamentalMat(pts1,pts2,cv.FM\_LMEDS)

# 我们只选择内点

pts1 = pts1[mask.ravel()==1]

pts2 = pts2[mask.ravel()==1]

接下来，我们找到Epilines。在第二张图像上绘制与第一张图像中的点相对应的Epilines。因此，在这里提到正确的图像很重要。我们得到了一行线。因此，我们定义了一个新功能来在图像上绘制这些线条。

def drawlines(img1,img2,lines,pts1,pts2):

''' img1 - 我们在img2相应位置绘制极点生成的图像

lines - 对应的极点 '''

r,c = img1.shape

img1 = cv.cvtColor(img1,cv.COLOR\_GRAY2BGR)

img2 = cv.cvtColor(img2,cv.COLOR\_GRAY2BGR)

for r,pt1,pt2 in zip(lines,pts1,pts2):

color = tuple(np.random.randint(0,255,3).tolist())

x0,y0 = map(int, [0, -r[2]/r[1] ])

x1,y1 = map(int, [c, -(r[2]+r[0]\*c)/r[1] ])

img1 = cv.line(img1, (x0,y0), (x1,y1), color,1)

img1 = cv.circle(img1,tuple(pt1),5,color,-1)

img2 = cv.circle(img2,tuple(pt2),5,color,-1)

return img1,img2

现在，我们在两个图像中都找到了Epiline并将其绘制。

# 在右图（第二张图）中找到与点相对应的极点，然后在左图绘制极线

lines1 = cv.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2,F)

lines1 = lines1.reshape(-1,3)

img5,img6 = drawlines(img1,img2,lines1,pts1,pts2)

# 在左图（第一张图）中找到与点相对应的Epilines，然后在正确的图像上绘制极线

lines2 = cv.computeCorrespondEpilines(pts1.reshape(-1,1,2), 1,F)

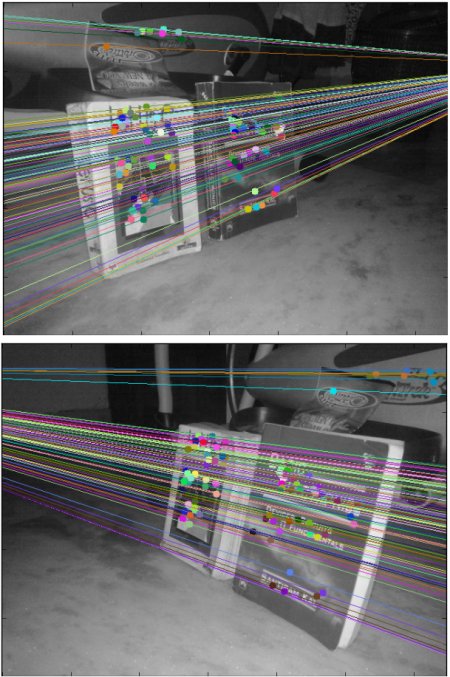
lines2 = lines2.reshape(-1,3)

img3,img4 = drawlines(img2,img1,lines2,pts2,pts1)

plt.subplot(121),plt.imshow(img5)

plt.subplot(122),plt.imshow(img3)

plt.show()

以下是我们得到的结果：  
[](https://panchuang.net/wp-content/uploads/2020/03/epiresult-1.jpg)

您可以在左侧图像中看到所有极点都收敛在右侧图像的外部。那个汇合点就是极点。  
为了获得更好的结果，应使用具有良好分辨率和许多非平面点的图像。