

第 7 讲

视觉里程计 1

本节目标

1. 理解图像特征点的意义，并掌握在单幅图像中提取出特征点，及多幅图像中匹配特征点的方法。
2. 理解对极几何的原理，利用对极几何的约束，恢复出图像之间的摄像机的三维运动。
3. 理解 PNP 问题，及利用已知三维结构与图像的对应关系，求解摄像机的三维运动。
4. 理解 ICP 问题，及利用点云的匹配关系，求解摄像机的三维运动。
5. 理解如何通过三角化，获得二维图像上对应点的三维结构。

本书之前的内容，介绍了运动方程和观测方程的具体形式，并讲解了以非线性优化为主的求解方法。从本讲开始，我们结束了基础知识的铺垫，开始步入正题：按照第二讲的内容，分别介绍视觉里程计、优化后端、回环检测和地图构建四个模块。本讲和下一讲主要介绍作为视觉里程计的主要理论，然后在第九章中进行一次实践。本讲关注基于特征点方式的视觉里程计算法。我们将介绍什么是特征点，如何提取和匹配特征点，以及如何根据配对的特征点估计相机运动。

7.1 特征点法

回顾第二讲的内容，我们说过视觉 SLAM 主要分为视觉前端和优化后端。前端也称为视觉里程计（VO）。它根据相邻图像的信息，估计出粗略的相机运动，给后端提供较好的初始值。VO 的实现方法，按是否需要提取特征，分为特征点法的前端以及不提特征的直接法前端。基于特征点法的前端，长久以来（直到现在）被认为是视觉里程计的主流方法。它运行稳定，对光照、动态物体不敏感，是目前比较成熟的解决方案。在本讲中，我们将从特征点法入手，学习如何提取、匹配图像特征点，然后估计两帧之间的相机运动和场景结构，从而实现一个基本的两帧间视觉里程计。

7.1.1 特征点

VO 的主要问题是是如何根据图像来估计相机运动。然而，图像本身是一个由亮度和色彩组成的矩阵，如果直接从矩阵层面考虑运动估计，将会非常困难。所以，我们习惯于采用这样一种做法：首先，从图像中选取比较有代表性的点。这些点在相机视角发生少量变化后会保持不变，所以我们在各个图像中找到相同的点。然后，在这些点的基础上，讨论相机位姿估计问题，以及这些点的定位问题。在经典 SLAM 模型中，把它们称为路标。而在视觉 SLAM 中，路标则是指图像特征（Features）。

根据维基百科的定义，图像特征是一组与计算任务相关的信息，计算任务取决于具体的应用 [29]。简而言之，**特征是图像信息的另一种数字表达形式**。一组好的特征对于在指定任务上的最终表现至关重要，所以多年来研究者们花费了大量的精力对特征进行研究。数字图像在计算机中以灰度值矩阵的方式存储，所以最简单的，单个图像像素也是一种“特征”。但是，在视觉里程计中，我们希望**特征点在相机运动之后保持稳定**，而灰度值受光照、形变、物体材质的影响严重，在不同图像之间变化非常大，不够稳定。理想的情况是，当场景和相机视角发生少量改变时，我还能从图像中判断哪些地方是同一个点，因此仅凭灰度值是不够的，我们需要对图像提取特征点。

特征点是图像里一些特别的地方。以图 7-1 为例。我们可以把图像中的角点、边缘和区块都当成图像中有代表性的地方。不过，我们更容易精确地指出，某两幅图像当中出现了同一个角点；同一个边缘则稍微困难一些，因为沿着该边缘前进，图像局部是相似的；同一个区块则是最困难的。我们发现，图像中的角点、边缘相比于像素区块而言更加“特别”，它们不同图像之间的辨识度更强。所以，一种直观的提取特征的方式就是在不同图像间辨认角点，确定它们的对应关系。在这种做法中，角点就是所谓的特征。

然而，在大多数应用中，单纯的角度依然不能满足很多我们的需求。例如，从远处看上去是角点的地方，当相机走近之后，可能就不显示为角点了。或者，当我旋转相机时，角点的外观会发生变化，我们也就不容易辨认出那是同一个角点。为此，计算机视觉领域的研究者

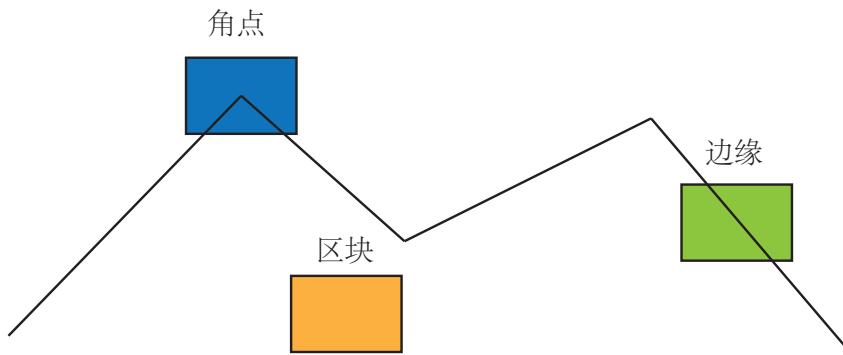


图 7-1 可以作为图像特征的部分：角点、边缘、区块

们在长年的研究中，设计了许多更加稳定的局部图像特征，如著名的 SIFT[30], SURF[31], ORB[32] 等等。相比于朴素的角点，这些人工设计的特征点能够拥有如下的性质：

1. 可重复性 (Repeatability): 相同的“区域”可以在不同的图像中被找到。
2. 可区别性 (Distinctiveness): 不同的“区域”有不同的表达。
3. 高效率 (Efficiency): 同一图像中，特征点的数量应远小于像素的数量。
4. 本地性 (Locality): 特征仅与一小片图像区域相关。

特征点由**关键点** (Key-point) 和**描述子** (Descriptor) 两部分组成。比方说，当我们谈论 SIFT 特征时，是指“提取 SIFT 关键点，并计算 SIFT 描述子”两件事情。关键点是指该特征点在图像里的位置，有些特征点还具有朝向、大小等信息。描述子通常是一个向量，按照某种人为设计的方式，描述了该关键点周围像素的信息。描述子是按照“**外观相似的特征应该有相似的描述子**”的原则设计的。因此，只要两个特征点的描述子在向量空间上的距离相近，就可以认为它们是同样的特征点。

历史上，研究者提出过许多图像特征。它们有些很精确，在相机的运动和光照变化下仍具有相似的表达，但相应地需要较大的计算量。其中，SIFT(尺度不变特征变换，Scale-Invariant Feature Transform) 当属最为经典的一种。它充分考虑了在图像变换过程中出现的光照，尺度，旋转等变化，但随之而来的是极大的计算量。由于整个 SLAM 过程中，图像特征的提取与匹配仅仅是诸多环节中的一个，到目前（2016 年）为止，普通 PC 的 CPU 还无法实时地计算 SIFT 特征，进行定位与建图。所以在 SLAM 中我们甚少使用这种“奢侈”的图像特征。

另一些特征，则考虑适当降低精度和鲁棒性，提升计算的速度。例如 FAST 关键点属于计算特别快的一种特征点（注意这里“关键点”的用词，说明它没有描述子）。而 ORB（Oriented FAST and Rotated BRIEF）特征则是目前看来非常具有代表性的实时图像特征。它改进了 FAST 检测子 [33] 不具有方向性的问题，并采用速度极快的二进制描述子 BRIEF[34]，使整个图像特征提取的环节大大加速。根据作者在论文中的测试，在同一幅图像中同时提取约 1000 个特征点的情况下，ORB 约要花费 15.3ms，SURF 约花费 217.3ms，SIFT 约花费 5228.7ms。由此可以看出 ORB 在保持了特征子具有旋转，尺度不变性的同时，速度方面提升明显，对于实时性要求很高的 SLAM 来说是一个很好的选择。

大部分特征提取都具有较好的并行性，可以通过 GPU 等设备来加速计算。经过 GPU 加速后的 SIFT，就可以满足实时计算要求。但是，引入 GPU 将带来整个 SLAM 成本的提升。由此带来的性能提升，是否足以抵去付出的计算成本，需要系统的设计人员仔细考量。在目前的 SLAM 方案中，ORB 是质量与性能之间较好的折中，因此我们以 ORB 为代表，介绍提取特征的整个过程。

7.1.2 ORB 特征

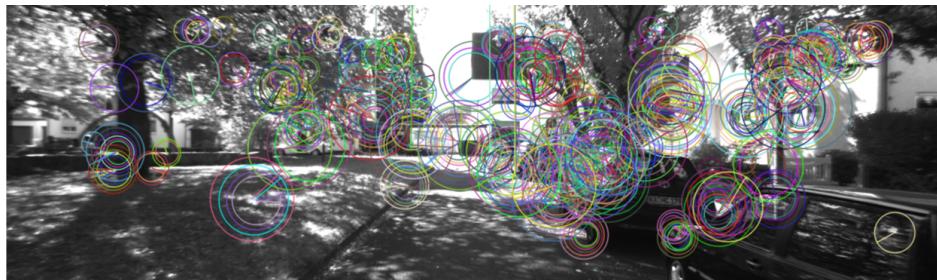


图 7-2 OpenCV 提供的 ORB 特征点检测结果。

ORB 特征亦由关键点和描述子两部分组成。它的关键点称为“Oriented FAST”，是一种改进的 FAST 角点，什么是 FAST 角点我们将在下文介绍。它的描述子称为 BRIEF (Binary Robust Independent Elementary Features)。因此，提取 ORB 特征分为两个步骤：

1. FAST 角点提取：找出图像中的“角点”。相较于原版的 FAST，ORB 中计算了特征点的主方向，为后续的 BRIEF 描述子增加了旋转不变特性。
2. BRIEF 描述子：对前一步提取出特征点的周围图像区域进行描述。

下面我们分别介绍 FAST 和 BRIEF。

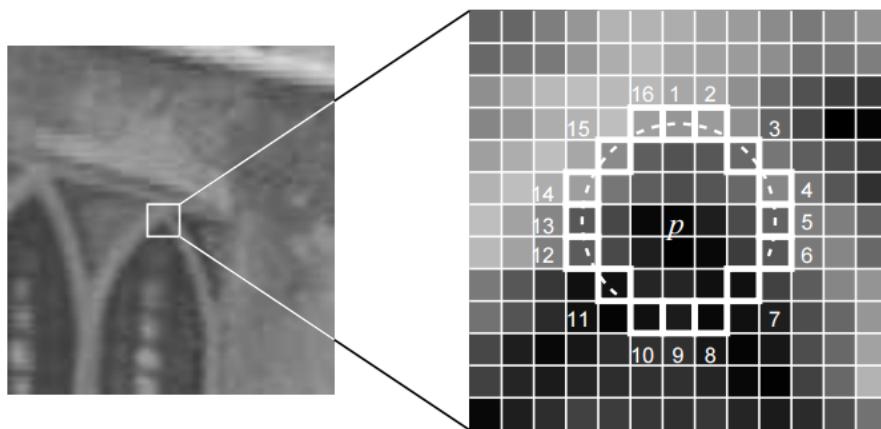
FAST 关键点

图 7-3 FAST 特征点 [33]。

FAST 是一种角点，主要检测局部像素灰度变化明显的地方，以速度快著称。它的思想是：如果一个像素与它邻域的像素差别较大（过亮或过暗），那它更可能是角点。相比于其他角点检测算法，FAST 只需比较像素亮度的大小，十分快捷。它的检测过程如下（见图 7-3）：

1. 在图像中选取像素 p ，假设它的亮度为 I_p 。
2. 设置一个阈值 T （比如 I_p 的 20%）。
3. 以像素 p 为中心，选取半径为 3 的圆上的 16 个像素点。
4. 假如选取的圆上，有连续的 N 个点的亮度大于 $I_p + T$ 或小于 $I_p - T$ ，那么像素 p 可以被认为是特征点（ N 通常取 12，即为 FAST-12。其它常用的 N 取值为 9 和 11，他们分别被称为 FAST-9，FAST-11）。
5. 循环以上四步，对每一个像素执行相同的操作。

在 FAST-12 算法中，为了更高效，可以添加一项预测试操作，以快速地排除绝大多数不是角点的像素。具体操作为，对于每个像素，直接检测邻域圆上的第 1, 5, 9, 13 个像素的亮度。只有当这四个像素中有三个同时大于 $I_p + T$ 或小于 $I_p - T$ 时，当前像素才有可能是一个角点，否则应该直接排除。这样的预测试操作大大加速了角点检测。此外，原始的 FAST 角点经常出现“扎堆”的现象。所以在第一遍检测之后，还需要用非极大值抑制。

制 (Non-maximal suppression)，在一定区域内仅保留响应极大值的角点，避免角点集中 的问题。

FAST 特征点的计算仅仅是比较像素间亮度的差异，速度非常快，但它也有一些问题。首先，FAST 特征点数量很大且不确定，而我们往往希望对图像提取固定数量的特征。因此，在 ORB 中，对原始的 FAST 算法进行了改进。我们可以指定最终要提取的角点数量 N ，对原始 FAST 角点分别计算 Harris 响应值，然后选取前 N 个具有最大响应值的角点，作为最终的角点集合。

其次，FAST 角点不具有方向信息。而且，由于它固定取半径为 3 的圆，存在尺度问题：远处看着像是角点的地方，接近后看可能就不是角点了。针对 FAST 角点不具有方向性和尺度的弱点，ORB 添加了尺度和旋转的描述。尺度不变性由构建图像金字塔^①，并在金字塔的每一层上检测角点来实现。而特征的旋转是由灰度质心法 (Intensity Centroid) 实现的。我们稍微介绍一下。

质心是指以图像块灰度值作为权重的中心。其具体操作步骤如下 [35]：

1. 在一个小的图像块 B 中，定义图像块的矩为：

$$m_{pq} = \sum_{x,y \in B} x^p y^q I(x,y), \quad p, q = \{0, 1\}.$$

2. 通过矩可以找到图像块的质心：

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right).$$

3. 连接图像块的几何中心 O 与质心 C ，得到一个方向向量 \overrightarrow{OC} ，于是特征点的方向可以定义为：

$$\theta = \arctan(m_{01}/m_{10}).$$

通过以上方法，FAST 角点便具有了尺度与旋转的描述，大大提升了它们在不同图像之间表述的鲁棒性。所以在 ORB 中，把这种改进后的 FAST 称为 Oriented FAST。

BRIEF 描述子

在提取 Oriented FAST 关键点后，我们对每个点计算其描述子。ORB 使用改进的 BRIEF 特征描述。我们先来讲 BRIEF 是什么。

^①金字塔是指对图像进行不同层次的降采样，以获得不同分辨率的图像。

BRIEF 是一种二进制描述子，它的描述向量由许多个 0 和 1 组成，这里的 0 和 1 编码了关键点附近两个像素（比如说 p 和 q ）的大小关系：如果 p 比 q 大，则取 1，反之就取 0。如果我们取了 128 个这样的 p, q ，最后就得到 128 维由 0, 1 组成的向量。那么， p 和 q 如何选取呢？在作者原始的论文中给出了若干种挑选方法，大体上都是按照某种概率分布，随机地挑选 p 和 q 的位置，读者可以阅读 BRIEF 论文或 OpenCV 源码以查看它的具体实现 [34]。BRIEF 使用了随机选点的比较，速度非常快，而且由于使用了二进制表达，存储起来也十分方便，适用于实时的图像匹配。原始的 BRIEF 描述子不具有旋转不变性的，因此在图像发生旋转时容易丢失。而 ORB 在 FAST 特征点提取阶段计算了关键点的方向，所以可以利用方向信息，计算了旋转之后的“Steer BRIEF”特征，使 ORB 的描述子具有较好的旋转不变性。

由于考虑到了旋转和缩放，使得 ORB 在平移、旋转、缩放的变换下仍有良好的表现。同时，FAST 和 BRIEF 的组合也非常的高效，使得 ORB 特征在实时 SLAM 中非常受欢迎。我们在图 7-2 中展示了一张图像提取 ORB 之后的结果，下面来介绍如何在不同的图像之间进行特征匹配。

7.1.3 特征匹配



图 7-4 两帧图像间的特征匹配。

特征匹配是视觉 SLAM 中极为关键的一步，宽泛地说，特征匹配解决了 SLAM 中的数据关联问题（data association），即确定当前看到的路标与之前看到的路标之间的对应

关系。通过对图像与图像，或者图像与地图之间的描述子进行准确的匹配，我们可以为后续的姿态估计，优化等操作减轻大量负担。然而，由于图像特征的局部特性，误匹配的情况广泛存在，而且长期以来一直没有得到有效解决，目前已经成为视觉 SLAM 中制约性能提升的一大瓶颈。部分原因是因为场景中经常存在大量的重复纹理，使得特征描述非常相似。在这种情况下，仅利用局部特征解决误匹配是非常困难的。

不过，让我们先来看正确匹配的情况，等做完实验再回头去讨论误匹配问题。考虑两个时刻的图像。如果在图像 I_t 中提取到特征点 x_t^m , $m = 1, 2, \dots, M$, 在图像 I_{t+1} 中提取到特征点 x_{t+1}^n , $n = 1, 2, \dots, N$, 如何寻找这两个集合元素的对应关系呢？最简单的特征匹配方法就是**暴力匹配（Brute-Force Matcher）**。即对每一个特征点 x_t^m , 与所有的 x_{t+1}^n 测量描述子的距离，然后排序，取最近的一个作为匹配点。描述子距离表示了两个特征之间的相似程度，不过在实际运用中还可以取不同的距离度量范数。对于浮点类型的描述子，使用欧氏距离进行度量即可。而对于二进制的描述子（比如 BRIEF 这样的），我们往往使用汉明距离（Hamming distance）做为度量——两个二进制串之间的汉明距离，指的是它们不同位数的个数。

然而，当特征点数量很大时，暴力匹配法的运算量将变得很大，特别是当我们想要匹配一个帧和一张地图的时候。这不符合我们在 SLAM 中的实时性需求。此时**快速近似最近邻（FLANN）**算法更加适合于匹配点数量极多的情况。由于这些匹配算法理论已经成熟，而且实现上也已集成到 OpenCV，所以我们这里就不再描述它的技术细节了。感兴趣的读者，可以阅读 [36] 作为参考。

7.2 实践：特征提取和匹配



图 7-5 实验使用的两帧图像。

目前主流的几种图像特征在 OpenCV 开源图像库中都已经集成完毕，我们可以很方便地进行调用。下面我们来实际练习一下 OpenCV 的图像特征提取、计算和匹配的过程。我们为此实验准备了两张图像，位于 `slambook/ch7/` 下的 `1.png` 和 `2.png`，如图 7-5 所示。它们是来自公开数据集 [37] 中的两张图像，我们看到相机发生了微小的运动。本节演示如

何提取 ORB 特征，并进行匹配。下个程序将演示如何估计相机运动。

特征提取与匹配代码：

slambook/ch7/feature_extraction.cpp

```
1 #include <iostream>
2 #include <opencv2/core/core.hpp>
3 #include <opencv2/features2d/features2d.hpp>
4 #include <opencv2/highgui/highgui.hpp>
5
6 using namespace std;
7 using namespace cv;
8
9 int main ( int argc, char** argv )
10 {
11     if ( argc != 3 )
12     {
13         cout<<"usage: feature_extraction img1 img2"<<endl;
14         return 1;
15     }
16     //-- 读取图像
17     Mat img_1 = imread ( argv[1], CV_LOAD_IMAGE_COLOR );
18     Mat img_2 = imread ( argv[2], CV_LOAD_IMAGE_COLOR );
19
20     //-- 初始化
21     std::vector<KeyPoint> keypoints_1, keypoints_2;
22     Mat descriptors_1, descriptors_2;
23     Ptr<ORB> orb = ORB::create ( 500, 1.2f, 8, 31, 0, 2, ORB::HARRIS_SCORE,31,20 );
24
25     //-- 第一步：检测 Oriented FAST 角点位置
26     orb->detect ( img_1,keypoints_1 );
27     orb->detect ( img_2,keypoints_2 );
28
29     //-- 第二步：根据角点位置计算 BRIEF 描述子
30     orb->compute ( img_1, keypoints_1, descriptors_1 );
31     orb->compute ( img_2, keypoints_2, descriptors_2 );
32
33     Mat outimg1;
34     drawKeypoints( img_1, keypoints_1, outimg1, Scalar::all(-1), DrawMatchesFlags::DEFAULT );
35     imshow("ORB特征点",outimg1);
36
37     //-- 第三步：对两幅图像中的BRIEF描述子进行匹配，使用 Hamming 距离
38     vector<DMatch> matches;
39     BFMatcher matcher ( NORM_HAMMING );
40     matcher.match ( descriptors_1, descriptors_2, matches );
41
42     //-- 第四步：匹配点对筛选
43     double min_dist=10000, max_dist=0;
```

```

45 // 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
46 for ( int i = 0; i < descriptors_1.rows; i++ )
47 {
48     double dist = matches[i].distance;
49     if ( dist < min_dist ) min_dist = dist;
50     if ( dist > max_dist ) max_dist = dist;
51 }
52
53 printf ( "-- Max dist : %f \n", max_dist );
54 printf ( "-- Min dist : %f \n", min_dist );
55
56 // 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。
57 // 但有时候最小距离会非常小，设置一个经验值作为下限。
58 std::vector< DMatch > good_matches;
59 for ( int i = 0; i < descriptors_1.rows; i++ )
60 {
61     if ( matches[i].distance <= max ( 2*min_dist, 30.0 ) )
62     {
63         good_matches.push_back ( matches[i] );
64     }
65 }
66
67 //-- 第五步：绘制匹配结果
68 Mat img_match;
69 Mat img_goodmatch;
70 drawMatches ( img_1, keypoints_1, img_2, keypoints_2, matches, img_match );
71 drawMatches ( img_1, keypoints_1, img_2, keypoints_2, good_matches, img_goodmatch );
72 imshow ( "所有匹配点对", img_match );
73 imshow ( "优化后匹配点对", img_goodmatch );
74 waitKey(0);
75
76 return 0;
77 }
```

运行此程序（需要输入两个图像位置），将输出运行结果：

```

1 % build/feature_extraction 1.png 2.png
2 -- Max dist : 95.000000
3 -- Min dist : 4.000000
```

图 7-6 显示了例程的运行结果。我们看到未筛选的匹配中带有大量的误匹配。经过一次筛选之后，匹配数量减少了许多，但大多数匹配都是正确的。这里，我们筛选的依据是汉明距离小于最小距离的两倍，这是一种工程上的经验方法，不一定有理论依据。不过，尽管在示例图像中能够筛选出正确的匹配，但我们仍然不能保证在所有其他图像中得到的匹配全是正确的。因此，在后面的运动估计中，还需要使用去除误匹配的算法。

接下来，我们希望根据匹配的点对，估计相机的运动。这里由于相机的原理不同，情况发生了变化：

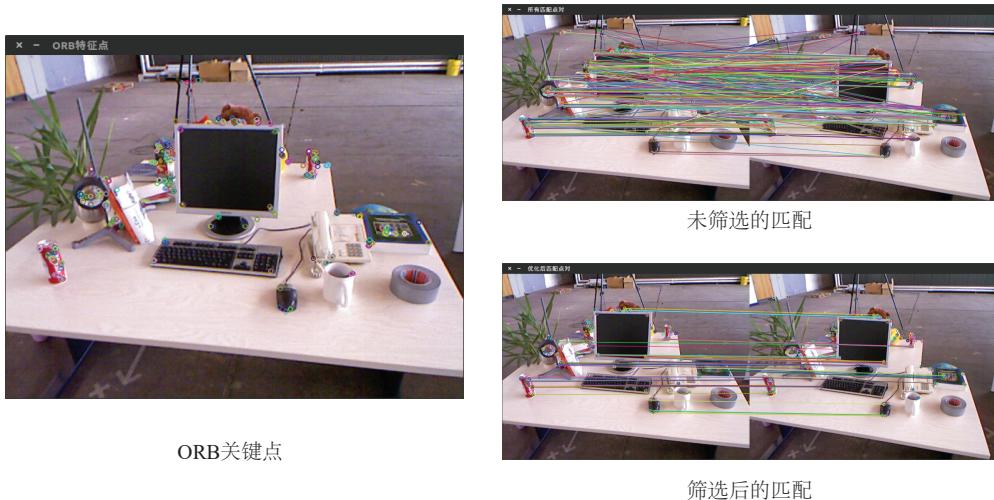


图 7-6 特征提取与匹配结果。

1. 当相机为单目时，我们只知道 2D 的像素坐标，因而问题是根据**两组 2D 点**估计运动。该问题用**对极几何**来解决。
2. 当相机为双目、RGB-D 时，或者我们通过某种方法得到了距离信息，那问题就是根据**两组 3D 点**估计运动。该问题通常用 ICP 来解决。
3. 如果我们有 3D 点和它们在相机的投影位置，也能估计相机的运动。该问题通过 **PnP** 求解。

因此，下面几节的内容，我们就来介绍这三种情形下的相机运动估计。我们将从最基本的 2D-2D 情形出发，看看它如何求解，求解过程又具有哪些麻烦的问题。

7.3 2D-2D: 对极几何

现在，假设我们从两张图像中，得到了一对配对好的特征点，像图 7-7 里显示的那样。如果我们有若干对这样的匹配点，就可以通过这些二维图像点的对应关系，恢复出在两帧之间摄像机的运动。这里“若干对”具体是多少对呢？我们会在下文介绍。先来看看两个图像当中的匹配点有什么几何关系吧。

以图 7-7 为例，我们希望求取两帧图像 I_1, I_2 之间的运动，设第一帧到第二帧的运动为 \mathbf{R}, \mathbf{t} 。两个相机中心分别为 O_1, O_2 。现在，考虑 I_1 中有一个特征点 p_1 ，它在 I_2 中对应着

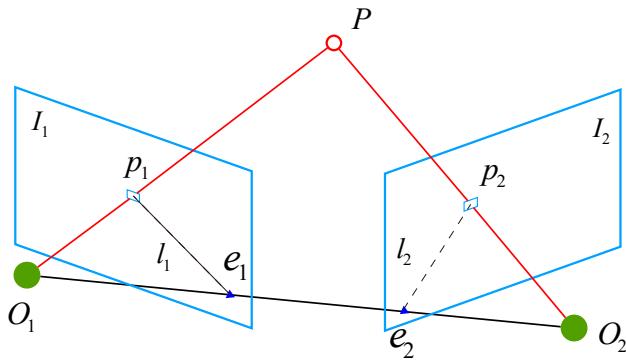


图 7-7 对极几何约束。

特征点 p_2 。我们晓得这两点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。这里我们需要一些术语来描述它们之间的几何关系。首先，连线 $\overrightarrow{O_1p_1}$ 和连线 $\overrightarrow{O_2p_2}$ 在三维空间中会相交于点 P 。这时候点 O_1, O_2, P 三个点可以确定一个平面，称为极平面（Epipolar plane）。 O_1O_2 连线与像平面 I_1, I_2 的交点分别为 e_1, e_2 。 e_1, e_2 ，称为极点（Epipoles）， O_1O_2 被称为基线（Baseline）。称极平面与两个像平面 I_1, I_2 之间的相交线 l_1, l_2 为极线（Epipolar line）。

直观上讲，从第一帧的角度上看，射线 $\overrightarrow{O_1p_1}$ 是某个像素可能出现的空间位置——因为该射线上的所有点都会投影到同一个像素点。同时，如果不知道 P 的位置，那么当我们在第二个图像上看时，连线 $\overrightarrow{e_2p_2}$ （也就是第二个图像中的极线）就是 P 可能出现的投影的位置，也就是射线 $\overrightarrow{O_1p_1}$ 在第二个相机中的投影。现在，由于我们通过特征点匹配，确定了 p_2 的像素位置，所以能够推断 P 的空间位置，以及相机的运动。要提醒读者的是，这都是多亏了正确的特征匹配。如果没有特征匹配，我们就没法确定 p_2 到底在极线的哪个位置了。那时，就必须在极线上搜索以获得正确的匹配，这将在第 13 讲中提到。

现在，我们从代数角度来看一下这里出现的几何关系。在第一帧的坐标系下，设 P 的空间位置为：

$$\mathbf{P} = [X, Y, Z]^T.$$

根据第 5 讲介绍的针孔相机模型，我们知道两个像素点 p_1, p_2 的像素位置为：

$$s_1 \mathbf{p}_1 = \mathbf{K} \mathbf{P}, \quad s_2 \mathbf{p}_2 = \mathbf{K} (\mathbf{R} \mathbf{P} + \mathbf{t}). \quad (7.1)$$

这里 \mathbf{K} 为相机内参矩阵， \mathbf{R}, \mathbf{t} 为两个坐标系的相机运动（如果我们愿意，也可以写成李代数形式）。如果使用齐次坐标，我们也可以把上式写成在乘以非零常数下成立的（up

to a scale) 等式^①:

$$\mathbf{p}_1 = \mathbf{K}\mathbf{P}, \quad \mathbf{p}_2 = \mathbf{K}(\mathbf{R}\mathbf{P} + \mathbf{t}). \quad (7.2)$$

现在, 取:

$$\mathbf{x}_1 = \mathbf{K}^{-1}\mathbf{p}_1, \quad \mathbf{x}_2 = \mathbf{K}^{-1}\mathbf{p}_2. \quad (7.3)$$

这里的 $\mathbf{x}_1, \mathbf{x}_2$ 是两个像素点的归一化平面上的坐标。代入上式, 得:

$$\mathbf{x}_2 = \mathbf{R}\mathbf{x}_1 + \mathbf{t}. \quad (7.4)$$

两边同时左乘 \mathbf{t}^\wedge 。回忆 \wedge 的定义, 这相当于两侧同时与 \mathbf{t} 做外积:

$$\mathbf{t}^\wedge\mathbf{x}_2 = \mathbf{t}^\wedge\mathbf{R}\mathbf{x}_1. \quad (7.5)$$

然后, 两侧同时左乘 \mathbf{x}_2^T :

$$\mathbf{x}_2^T\mathbf{t}^\wedge\mathbf{x}_2 = \mathbf{x}_2^T\mathbf{t}^\wedge\mathbf{R}\mathbf{x}_1. \quad (7.6)$$

观察等式左侧, $\mathbf{t}^\wedge\mathbf{x}_2$ 是一个与 \mathbf{t} 和 \mathbf{x}_2 都垂直的向量。把它再和 \mathbf{x}_2 做内积时, 将得到 0。因此, 我们就得到了一个简洁的式子:

$$\mathbf{x}_2^T\mathbf{t}^\wedge\mathbf{R}\mathbf{x}_1 = 0. \quad (7.7)$$

重新代入 $\mathbf{p}_1, \mathbf{p}_2$, 有:

$$\mathbf{p}_2^T\mathbf{K}^{-T}\mathbf{t}^\wedge\mathbf{R}\mathbf{K}^{-1}\mathbf{p}_1 = 0. \quad (7.8)$$

这两个式子都称为对极约束, 它以形式简洁著名。它的几何意义是 O_1, P, O_2 三者共面。对极约束中同时包含了平移和旋转。我们把中间部分记作两个矩阵: 基础矩阵 (Fundamental Matrix) \mathbf{F} 和本质矩阵 (Essential Matrix) \mathbf{E} , 可以进一步简化对极约束:

$$\mathbf{E} = \mathbf{t}^\wedge\mathbf{R}, \quad \mathbf{F} = \mathbf{K}^{-T}\mathbf{E}\mathbf{K}^{-1}, \quad \mathbf{x}_2^T\mathbf{E}\mathbf{x}_1 = \mathbf{p}_2^T\mathbf{F}\mathbf{p}_1 = 0. \quad (7.9)$$

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也就是说, 在等式一侧乘以任意非零常数时, 我们认为等式仍是成立的。

1. 根据配对点的像素位置, 求出 \mathbf{E} 或者 \mathbf{F} ;
2. 根据 \mathbf{E} 或者 \mathbf{F} , 求出 \mathbf{R}, \mathbf{t} .

由于 \mathbf{E} 和 \mathbf{F} 只相差了相机内参, 而内参在 SLAM 中通常是已知的^①, 所以实践当中往往使用形式更简单的 \mathbf{E} 。我们以 \mathbf{E} 为例, 介绍上面两个问题如何求解。

7.3.2 本质矩阵

根据定义, 本质矩阵 $\mathbf{E} = \mathbf{t}^\wedge \mathbf{R}$ 。它是一个 3×3 的矩阵, 内有 9 个未知数。那么, 是不是任意一个 3×3 的矩阵都可以被当成本质矩阵呢? 从 \mathbf{E} 的构造方式上看, 有以下值得注意的地方:

- 本质矩阵是由对极约束定义的。由于对极约束是等式为零的约束, 所以对 \mathbf{E} 乘以任意非零常数后, 对极约束依然满足。我们把这件事情称为 \mathbf{E} 在不同尺度下是等价的。
- 根据 $\mathbf{E} = \mathbf{t}^\wedge \mathbf{R}$, 可以证明 [3], 本质矩阵 \mathbf{E} 的奇异值必定是 $[\sigma, \sigma, 0]^T$ 的形式。这称为本质矩阵的内在性质。
- 另一方面, 由于平移和旋转各有三个自由度, 故 $\mathbf{t}^\wedge \mathbf{R}$ 共有六个自由度。但由于尺度等价性, 故 \mathbf{E} 实际上有五个自由度。

\mathbf{E} 具有五个自由度的事实, 表明我们最少可以用五对点来求解 \mathbf{E} 。但是, \mathbf{E} 的内在性质是一种非线性性质, 在求解线性方程时会带来麻烦, 因此, 也可以只考虑它的尺度等价性, 使用八对点来估计 \mathbf{E} ——这就是经典的八点法 (Eight-point-algorithm) [38, 39]。八点法只利用了 \mathbf{E} 的线性性质, 因此可以在线性代数框架下求解。下面我们来看八点法是如何工作的。

考虑一对匹配点, 它们的归一化坐标为: $\mathbf{x}_1 = [u_1, v_1, 1]^T$, $\mathbf{x}_2 = [u_2, v_2, 1]^T$ 。根据对极约束, 有:

$$(u_1, v_1, 1) \begin{pmatrix} e_1 & e_2 & e_3 \\ e_4 & e_5 & e_6 \\ e_7 & e_8 & e_9 \end{pmatrix} \begin{pmatrix} u_2 \\ v_2 \\ 1 \end{pmatrix} = 0. \quad (7.10)$$

我们把矩阵 \mathbf{E} 展开, 写成向量的形式:

$$\mathbf{e} = [e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9]^T,$$

^①在 SfM 研究中则有可能是未知, 有待估计的。

那么对极约束可以写成与 e 有关的线性形式:

$$[u_1 u_2, u_1 v_2, u_1, v_1 u_2, v_1 v_2, v_1, u_2, v_2, 1] \cdot e = 0. \quad (7.11)$$

同理, 对于其它点对也有相同的表示。我们把所有点都放到一个方程中, 变成线性方程组 (u^i, v^i 表示第 i 个特征点, 以此类推):

$$\begin{pmatrix} u_1^1 u_2^1 & u_1^1 v_2^1 & u_1^1 & v_1^1 u_2^1 & v_1^1 v_2^1 & v_1^1 & u_2^1 & v_2^1 & 1 \\ u_1^2 u_2^2 & u_1^2 v_2^2 & u_1^2 & v_1^2 u_2^2 & v_1^2 v_2^2 & v_1^2 & u_2^2 & v_2^2 & 1 \\ \vdots & \vdots \\ u_1^8 u_2^8 & u_1^8 v_2^8 & u_1^8 & v_1^8 u_2^8 & v_1^8 v_2^8 & v_1^8 & u_2^8 & v_2^8 & 1 \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \\ e_9 \end{pmatrix} = 0. \quad (7.12)$$

这八个方程构成了一个线性方程组。它的系数矩阵由特征点位置构成, 大小为 8×9 。 e 位于该矩阵的零空间中。如果系数矩阵是满秩的 (即秩为 8), 那么它的零空间维数为 1, 也就是 e 构成一条线。这与 e 的尺度等价性是一致的。如果八对匹配点组成的矩阵满足秩为 8 的条件, 那么 E 的各元素就可由上述方程解得。

接下来的问题是如何根据已经估得的本质矩阵 E , 恢复出相机的运动 R, t 。这个过程是由奇异值分解 (SVD) 得到的。设 E 的 SVD 分解为:

$$E = U \Sigma V^T, \quad (7.13)$$

其中 U, V 为正交阵, Σ 为奇异值矩阵。根据 E 的内在性质, 我们知道 $\Sigma = \text{diag}(\sigma, \sigma, 0)$ 。在 SVD 分解中, 对于任意一个 E , 存在两个可能的 t, R 与它对应:

$$\begin{aligned} t_1^\wedge &= UR_Z\left(\frac{\pi}{2}\right)\Sigma U^T, & R_1 &= UR_Z^T\left(\frac{\pi}{2}\right)V^T \\ t_2^\wedge &= UR_Z\left(-\frac{\pi}{2}\right)\Sigma U^T, & R_2 &= UR_Z^T\left(-\frac{\pi}{2}\right)V^T. \end{aligned} \quad (7.14)$$

其中 $R_Z\left(\frac{\pi}{2}\right)$ 表示沿 Z 轴旋转 90 度得到的旋转矩阵。同时, 由于 $-E$ 和 E 等价, 所以对任意一个 t 取负号, 也会得到同样的结果。因此, 从 E 分解到 t, R 时, 一共存在四

个可能的解。

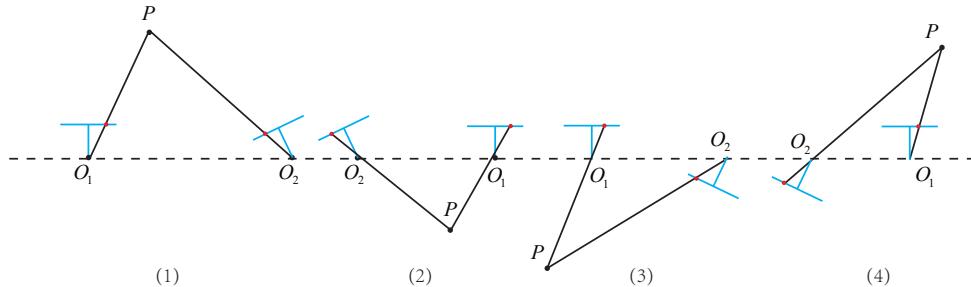


图 7-8 分解本质矩阵得到的四个解。在保持投影点（红点）不变的情况下，两个相机以及空间点一共有四种可能的情况。

图 7-8 形象地显示了分解本质矩阵得到的四个解。我们已知空间点在相机（蓝色线）上的投影（红点），想要求解相机的运动。在保持红点不变的情况下，可以画出四种可能的情况，不过幸运的是，只有第一种解中， P 在两个相机中都具有正的深度。因此，只要把任意一点代入四种解中，检测该点在两个相机下的深度，就可以确定哪个解是正确的了。

如果利用 \mathbf{E} 的内在性质，那么它只有五个自由度。所以最小可以通过五对点来求解相机运动 [40, 41]。然而这种做法形式复杂，从工程实现角度考虑，由于平时通常会有几十对乃至上百对的匹配点，从八对减至五对意义并不明显。为保持简单，我们这里就只介绍基本的八点法了。

剩下的问题还有一个：根据线性方程解出的 \mathbf{E} ，可能不满足 \mathbf{E} 的内在性质——它的奇异值不一定为 $\sigma, \sigma, 0$ 的形式。这时，在做 SVD 时，我们会刻意地把 Σ 矩阵调整成上面的样子。通常的做法是，对八点法求得的 \mathbf{E} 进行 SVD 分解后，会得到奇异值矩阵 $\Sigma = \text{diag}(\sigma_1, \sigma_2, \sigma_3)$ ，不妨设 $\sigma_1 \geq \sigma_2 \geq \sigma_3$ 。取：

$$\mathbf{E} = \mathbf{U} \text{diag}\left(\frac{\sigma_1 + \sigma_2}{2}, \frac{\sigma_1 + \sigma_2}{2}, 0\right) \mathbf{V}^T. \quad (7.15)$$

这相当于是把求出来的矩阵投影到了 \mathbf{E} 所在的流形上。当然，更简单的做法是将奇异值矩阵取成 $\text{diag}(1, 1, 0)$ ，因为 \mathbf{E} 具有尺度等价性，这样做也是合理的。

7.3.3 单应矩阵

除了基本矩阵和本质矩阵，我们还有一种称为单应矩阵（Homography） \mathbf{H} 的东西，它描述了两个平面之间的映射关系。若场景中的特征点都落在同一平面上（比如墙，地面等），

则可以通过单应性来进行运动估计。这种情况在无人机携带的俯视相机，或扫地机携带的顶视相机中比较常见。由于之前没提到过单应，我们稍微介绍一下。

单应矩阵通常描述处于共同平面上的一些点，在两张图像之间的变换关系。考虑在图像 I_1 和 I_2 有一对匹配好的特征点 p_1 和 p_2 。这些特征点落在某平面上。设这个平面满足方程：

$$\mathbf{n}^T \mathbf{P} + d = 0. \quad (7.16)$$

稍加整理，得：

$$-\frac{\mathbf{n}^T \mathbf{P}}{d} = 1. \quad (7.17)$$

然后，回顾本开头的式 (7.1)，得：

$$\begin{aligned} \mathbf{p}_2 &= \mathbf{K}(\mathbf{R}\mathbf{P} + \mathbf{t}) \\ &= \mathbf{K} \left(\mathbf{R}\mathbf{P} + \mathbf{t} \cdot \left(-\frac{\mathbf{n}^T \mathbf{P}}{d} \right) \right) \\ &= \mathbf{K} \left(\mathbf{R} - \frac{\mathbf{t}\mathbf{n}^T}{d} \right) \mathbf{P} \\ &= \mathbf{K} \left(\mathbf{R} - \frac{\mathbf{t}\mathbf{n}^T}{d} \right) \mathbf{K}^{-1} \mathbf{p}_1. \end{aligned}$$

于是，我们得到了一个直接描述图像坐标 \mathbf{p}_1 和 \mathbf{p}_2 之间的变换，把中间这部分记为 \mathbf{H} ，于是

$$\mathbf{p}_2 = \mathbf{H}\mathbf{p}_1. \quad (7.18)$$

它的定义与旋转、平移以及平面的参数有关。与基础矩阵 \mathbf{F} 类似，单应矩阵 \mathbf{H} 也是一个 3×3 的矩阵，求解时的思路也和 \mathbf{F} 类似，同样地可以先根据匹配点计算 \mathbf{H} ，然后将它分解以计算旋转和平移。把上式展开，得：

$$\begin{pmatrix} u_2 \\ v_2 \\ 1 \end{pmatrix} = \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix}. \quad (7.19)$$

请注意这里的等号是在非零因子下成立的。我们在实际处理中，通常乘以一个非零因

子使得 $h_9 = 1$ (在它取非零值时)。然后根据第三行, 去掉这个非零因子, 于是有:

$$u_2 = \frac{h_1 u_1 + h_2 v_1 + h_3}{h_7 u_1 + h_8 v_1 + h_9}$$

$$v_2 = \frac{h_4 u_1 + h_5 v_1 + h_6}{h_7 u_1 + h_8 v_1 + h_9}.$$

整理得:

$$h_1 u_1 + h_2 v_1 + h_3 - h_7 u_1 u_2 - h_8 v_1 u_2 = u_2$$

$$h_4 u_1 + h_5 v_1 + h_6 - h_7 u_1 v_2 - h_8 v_1 v_2 = v_2.$$

这样一组匹配点对就可以构造出两项约束(事实上有三个约束, 但是因为线性相关, 只取前两个), 于是自由度为 8 的单应矩阵可以通过 4 对匹配特征点算出(注意: 这些特征点不能有三点共线的情况), 即求解以下的线性方程组(当 $h_9 = 0$ 时, 右侧为零):

$$\begin{pmatrix} u_1^1 & v_1^1 & 1 & 0 & 0 & 0 & -u_1^1 u_2^1 & -v_1^1 u_2^1 \\ 0 & 0 & 0 & u_1^1 & v_1^1 & 1 & -u_1^1 v_2^1 & -v_1^1 v_2^1 \\ u_1^2 & v_1^2 & 1 & 0 & 0 & 0 & -u_1^2 u_2^2 & -v_1^2 u_2^2 \\ 0 & 0 & 0 & u_1^2 & v_1^2 & 1 & -u_1^2 v_2^2 & -v_1^2 v_2^2 \\ u_1^3 & v_1^3 & 1 & 0 & 0 & 0 & -u_1^3 u_2^3 & -v_1^3 u_2^3 \\ 0 & 0 & 0 & u_1^3 & v_1^3 & 1 & -u_1^3 v_2^3 & -v_1^3 v_2^3 \\ u_1^4 & v_1^4 & 1 & 0 & 0 & 0 & -u_1^4 u_2^4 & -v_1^4 u_2^4 \\ 0 & 0 & 0 & u_1^4 & v_1^4 & 1 & -u_1^4 v_2^4 & -v_1^4 v_2^4 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \end{pmatrix} = \begin{pmatrix} u_2^1 \\ v_2^1 \\ u_2^2 \\ v_2^2 \\ u_2^3 \\ v_2^3 \\ u_2^4 \\ v_2^4 \end{pmatrix}. \quad (7.20)$$

这种做法把 \mathbf{H} 矩阵看成了向量, 通过解该向量的线性方程来恢复 \mathbf{H} , 又称直接线性变换法(Direct Linear Transform)。与本质矩阵相似, 求出单应矩阵以后需要对其进行分解, 才可以得到相应的旋转矩阵 \mathbf{R} 和平移向量 \mathbf{t} 。分解的方法包括数值法[42, 43]与解析法[44]。与本质矩阵的分解类似, 单应矩阵的分解同样会返回四组旋转矩阵与平移向量, 并且同时可以计算出它们分别对应的场景点所在平面的法向量。如果已知成像的地图点的深度全为正值(即在相机前方), 则又可以排除两组解。最后仅剩两组解, 这时需要通过更多的先验信息进行判断。通常我们可以通过假设已知场景平面的法向量来解决, 如场景平面与相机平面平行, 那么法向量 \mathbf{n} 的理论值为 $\mathbf{1}^T$ 。

单应性在 SLAM 中具重要意义。当特征点共面, 或者相机发生纯旋转的时候, 基础矩阵的自由度下降, 这就出现了所谓的退化(degenerate)。现实中的数据总包含一些噪声,

这时候如果我们继续使用八点法求解基础矩阵，基础矩阵多余出来的自由度将会主要由噪声决定。为了能够避免退化现象造成的影响，通常我们会同时估计基础矩阵 F 和单应矩阵 H ，选择重投影误差比较小的那个作为最终的运动估计矩阵。

7.4 实践：对极约束求解相机运动

下面，我们来练习一下如何通过 Essential 矩阵求解相机运动。上一节实践部分的程序提供了特征匹配，而这次我们就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 。整个程序使用 OpenCV 提供的算法进行求解。我们把上一节的特征提取封装成函数，以供后面使用。本节只展示位姿估计部分的代码。

slambook/ch7/pose_estimation_2d2d.cpp（片段）

```
1 void pose_estimation_2d2d (
2     std::vector<KeyPoint> keypoints_1,
3     std::vector<KeyPoint> keypoints_2,
4     std::vector< DMatch > matches,
5     Mat& R, Mat& t )
6 {
7     // 相机内参, TUM Freiburg2
8     Mat K = ( Mat<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );
9
10    //-- 把匹配点转换为 vector<Point2f> 的形式
11    vector<Point2f> points1;
12    vector<Point2f> points2;
13
14    for ( int i = 0; i < ( int ) matches.size(); i++ )
15    {
16        points1.push_back( keypoints_1[matches[i].queryIdx].pt );
17        points2.push_back( keypoints_2[matches[i].trainIdx].pt );
18    }
19
20    //-- 计算基础矩阵
21    Mat fundamental_matrix;
22    fundamental_matrix = findFundamentalMat ( points1, points2, CV_FM_8POINT );
23    cout<<"fundamental_matrix is "<< endl << fundamental_matrix << endl;
24
25    //-- 计算本质矩阵
26    Point2d principal_point ( 325.1, 249.7 );      // 光心, TUM dataset 标定值
27    int focal_length = 521;           // 焦距, TUM dataset 标定值
28    Mat essential_matrix;
29    essential_matrix = findEssentialMat ( points1, points2, focal_length, principal_point, RANSAC );
30    cout<<"essential_matrix is "<< endl << essential_matrix << endl;
31
32    //-- 计算单应矩阵
33    Mat homography_matrix;
34    homography_matrix = findHomography ( points1, points2, RANSAC, 3, noArray(), 2000, 0.99 );
```

```

35 cout<<"homography_matrix is "<<endl<<homography_matrix<<endl;
36
37 //-- 从本质矩阵中恢复旋转和平移信息.
38 recoverPose ( essential_matrix, points1, points2, R, t, focal_length, principal_point );
39 cout<<"R is "<<endl<<R<<endl;
40 cout<<"t is "<<endl<<t<<endl;
41 }

```

该函数提供了从特征点求解相机运动的部分，然后，我们在主函数中调用它，就能得到相机的运动：

slambook/ch7/pose_estimation_2d2d.cpp（片段）

```

1 int main( int argc, char** argv )
2 {
3     if ( argc != 3 )
4     {
5         cout<<"usage: feature_extraction img1 img2"<<endl;
6         return 1;
7     }
8     //-- 读取图像
9     Mat img_1 = imread ( argv[1], CV_LOAD_IMAGE_COLOR );
10    Mat img_2 = imread ( argv[2], CV_LOAD_IMAGE_COLOR );
11
12    vector<KeyPoint> keypoints_1, keypoints_2;
13    vector<DMatch> matches;
14    find_feature_matches( img_1, img_2, keypoints_1, keypoints_2, matches );
15    cout<<"一共找到了"<<matches.size()<<"组匹配点"<<endl;
16
17    //-- 估计两张图像间运动
18    Mat R,t;
19    pose_estimation_2d2d( keypoints_1, keypoints_2, matches, R, t );
20
21    //-- 验证  $E=t^T R * scale$ 
22    Mat t_x = (Mat_<double>(3,3) <<
23        0, -t.at<double>(2,0), t.at<double>(1,0),
24        t.at<double>(2,0), 0, -t.at<double>(0,0),
25        -t.at<double>(1,0), t.at<double>(0,0), 0);
26
27    cout<<"t^T R = "<<endl<<t_x*R<<endl;
28    //-- 验证对极约束
29    Mat K = ( Mat_<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );
30    for ( DMatch m: matches )
31    {
32        Point2d pt1 = pixel2cam( keypoints_1[ m.queryIdx ].pt, K );
33        Mat y1 = (Mat_<double>(3,1) << pt1.x, pt1.y, 1);
34        Point2d pt2 = pixel2cam( keypoints_2[ m.trainIdx ].pt, K );
35        Mat y2 = (Mat_<double>(3,1) << pt2.x, pt2.y, 1);
36        Mat d = y2.t() * t_x * R * y1;

```

```

37     cout << "epipolar constraint = " << d << endl;
38 }
39 return 0;
40 }
```

我们在函数中输出了 \mathbf{E} , \mathbf{F} 和 \mathbf{H} 的数值, 然后验证对极约束是否成立, 以及 $\mathbf{t}^\wedge \mathbf{R}$ 和 \mathbf{E} 在非零数乘下等价的事实。现在, 调用此程序即可看到输出结果:

```

1 % build/pose_estimation_2d2d 1.png 2.png
2 -- Max dist : 95.000000
3 -- Min dist : 4.000000
4 一共找到了 79 组匹配点
5 fundamental_matrix is
6 [4.84448438246611e-06, 0.0001222601840188731, -0.01786737827487386;
7 -0.0001174326832719333, 2.122888800459598e-05, -0.01775877156212593;
8 0.01799658210895528, 0.008143605989020664, 1]
9 essential_matrix is
10 [-0.0203618550523477, -0.4007110038118445, -0.03324074249824097;
11 0.3939270778216369, -0.03506401846698079, 0.5857110303721015;
12 -0.006788487241438284, -0.5815434272915686, -0.01438258684486258]
13 homography_matrix is
14 [0.9497129583105288, -0.143556453147626, 31.20121878625771;
15 0.04154536627445031, 0.9715568969832015, 5.306887618807696;
16 -2.81813676978796e-05, 4.353702039810921e-05, 1]
17 R is
18 [0.9985961798781875, -0.05169917220143662, 0.01152671359827873;
19 0.05139607508976055, 0.9983603445075083, 0.02520051547522442;
20 -0.01281065954813571, -0.02457271064688495, 0.9996159607036126]
21 t is
22 [-0.8220841067933337;
23 -0.03269742706405412;
24 0.5684264241053522]
25
26 t^R=
27 [0.02879601157010516, 0.5666909361828478, 0.04700950886436416;
28 -0.5570970160413605, 0.0495880104673049, -0.8283204827837456;
29 0.009600370724838804, 0.8224266019846683, 0.02034004937801349]
30 epipolar constraint = [0.002528128704106625]
31 epipolar constraint = [-0.001663727901710724]
32 epipolar constraint = [-0.0008009088410884102]
33 .....
```

在程序的输出结果中可以看出, 对极约束的满足精度约在 10^{-3} 量级。根据前面的讨论, 分解得到的 \mathbf{R}, \mathbf{t} 一共有四种可能性。不过 OpenCV 替我们使用三角化检测角点的深度是否为正, 从而选出正确的解。

需要注意的地方是, 我们要弄清程序求解出来的 \mathbf{R}, \mathbf{t} 是什么意义。按照例程的定义, 我们的对极约束是从

$$\mathbf{x}_2 = \mathbf{R}\mathbf{x}_1 + \mathbf{t}$$

得到的。这里的 \mathbf{R}, \mathbf{t} 组成的变换矩阵，是第一个图到第二个图的坐标变换矩阵：

$$\mathbf{x}_2 = \mathbf{T}_{21}\mathbf{x}_1. \quad (7.21)$$

请读者在实践中务必清楚这里使用的变换顺序（因为有时我们会用 \mathbf{T}_{12} ），它们非常容易搞反。

7.4.1 讨论

从演示程序中可以看到，输出的 \mathbf{E} 和 \mathbf{F} 之差相差了相机内参矩阵。虽然它们在数值上并不直观，但可以验证它们的数学关系。从 \mathbf{E}, \mathbf{F} 和 \mathbf{H} 都可以分解出运动，不过 \mathbf{H} 需要假设特征点位于平面上。对于本实验的数据，这个假设是不好的，所以我们这里主要用 \mathbf{E} 来分解运动。

值得一提的是，由于 \mathbf{E} 本身具有尺度等价性，它分解得到的 \mathbf{t}, \mathbf{R} 也有一个尺度等价性。而 $\mathbf{R} \in SO(3)$ 自身具有约束，所以我们认为 \mathbf{t} 具有一个尺度。换言之，在分解过程中，对 \mathbf{t} 乘以任意非零常数，分解都是成立的。因此，我们通常把 \mathbf{t} 进行归一化，让它的长度等于 1。

尺度不确定性

对 \mathbf{t} 长度的归一化，直接导致了单目视觉的尺度不确定性（Scale Ambiguity）。例如，程序中输出的 \mathbf{t} 第一维约 0.822。这个 0.822 究竟是指 0.822 米呢，还是 0.822 厘米呢，我们是没法确定的。因为对 \mathbf{t} 乘以任意比例常数后，对极约束依然是成立的。换言之，在单目 SLAM 中，对轨迹和地图同时缩放任意倍数，我们得到的图像依然是一样的。这在第二讲中就已经给读者介绍过了。

在单目视觉中，我们对两张图像的 \mathbf{t} 归一化，相当于固定了尺度。虽然我们不知道它的实际长度为多少，但我们以这时的 \mathbf{t} 为单位 1，计算相机运动和特征点的 3D 位置。这被称为单目 SLAM 的初始化。在初始化之后，就可以用 3D-2D 来计算相机运动了。初始化之后的轨迹和地图的单位，就是初始化时固定的尺度。因此，单目 SLAM 有一步不可避免的初始化。初始化的两张图像必须有一定程度的平移，而后的轨迹和地图都将以此次的平移为单位。

除了对 \mathbf{t} 进行归一化之外，另一种方法是令初始化时所有的特征点平均深度为 1，也可以固定一个尺度。相比于令 \mathbf{t} 长度为 1 的做法，把特征点深度归一化可以控制场景的规模大小，使计算在数值上更稳定些。不过这并没有理论上的差别。

初始化的纯旋转问题

从 \mathbf{E} 分解到 \mathbf{R}, \mathbf{t} 的过程中，如果相机发生的是纯旋转，导致 \mathbf{t} 为零，那么，得到的 \mathbf{E} 也将为零，这将导致我们无从求解 \mathbf{R} 。不过，此时我们可以依靠 \mathbf{H} 求取旋转，但仅有旋转时，我们无法用三角测量估计特征点的空间位置（这将在下文提到），于是，另一个结论是，**单目初始化不能只有纯旋转，必须要有一定程度的平移**。如果没有平移，单目将无法初始化。在实践当中，如果初始化时平移太小，会使得位姿求解与三角化结果不稳定，从而导致失败。相对的，如果把相机左右移动而不是原地旋转，就容易让单目 SLAM 初始化。因而有经验的 SLAM 研究人员，在单目 SLAM 情况下，经常选择让相机进行左右平移以顺利地进行初始化。

多于八对点的情况

当给定的点数多于八对时（比如例程找到了 79 对匹配），我们可以计算一个最小二乘解。回忆式 (7.12) 中线性化后的对极约束，我们把左侧的系数矩阵记为 \mathbf{A} :

$$\mathbf{A}\mathbf{e} = \mathbf{0}. \quad (7.22)$$

对于八点法， \mathbf{A} 的大小为 8×9 。如果给定的匹配点多于 8，该方程构成一个超定方程，即不一定存在 \mathbf{e} 使得上式成立。因此，可以通过最小化一个二次型来求：

$$\min_{\mathbf{e}} \|\mathbf{A}\mathbf{e}\|_2^2 = \min_{\mathbf{e}} \mathbf{e}^T \mathbf{A}^T \mathbf{A} \mathbf{e}. \quad (7.23)$$

于是就求出了在最小二乘意义下的 \mathbf{E} 矩阵。不过，当可能存在误匹配的情况时，我们会更倾向于使用随机采样一致性（Random Sample Consensus, RANSAC）来求，而不是最小二乘。RANSAC 是一种通用的做法，适用于很多带错误数据的情况，可以处理带有错误匹配的数据。

7.5 三角测量

之前两节，我们使用对极几何约束估计了相机运动，也讨论这种方法的局限性。在得到运动之后，下一步我们需要用相机的运动估计特征点的空间位置。在单目 SLAM 中，仅通过单张图像无法获得像素的深度信息，我们需要通过**三角测量（Triangulation）**（或**三角化**）的方法来估计地图点的深度。

三角测量是指，通过在两处观察同一个点的夹角，确定该点的距离。三角测量最早由高斯提出并应用于测量学中，它在天文学、地理学的测量中都有应用。例如，我们可以通过不同季节观察到星星的角度，估计它离我们的距离。在 SLAM 中，我们主要用三角化来估计像素点的距离。

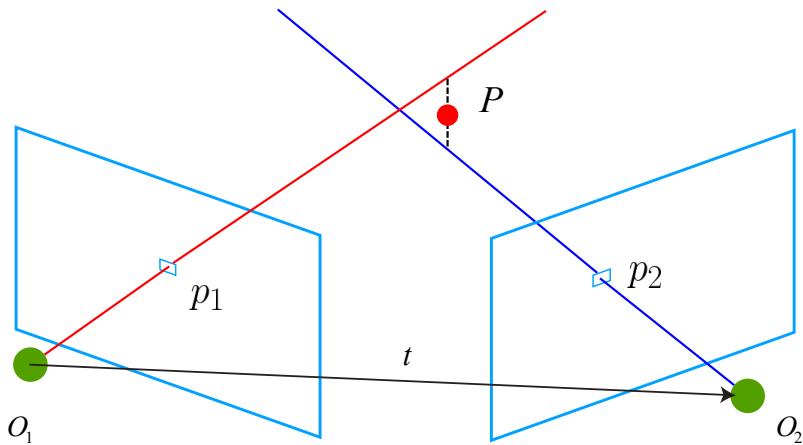


图 7-9 三角化获得地图点深度

和上一节类似，考虑图像 I_1 和 I_2 ，以左图为参考，右图的变换矩阵为 \mathbf{T} 。相机光心为 O_1 和 O_2 。在 I_1 中有特征点 p_1 ，对应 I_2 中有特征点 p_2 。理论上直线 O_1p_1 与 O_2p_2 在场景中会相交于一点 P ，该点即是两个特征点所对应的地图点在三维场景中的位置。然而由于噪声的影响，这两条直线往往无法相交。因此，（又）可以通过最二小乘去求解。

按照对极几何中的定义，设 $\mathbf{x}_1, \mathbf{x}_2$ 为两个特征点的归一化坐标，那么它们满足：

$$s_1 \mathbf{x}_1 = s_2 \mathbf{R} \mathbf{x}_2 + \mathbf{t}. \quad (7.24)$$

现在我们已经知道了 \mathbf{R}, \mathbf{t} ，想要求解的是两个特征点的深度 s_1, s_2 。当然这两个深度是可以分开求的，比方说先来看 s_2 。如果我要算 s_2 ，那么先对上式两侧左乘一个 \mathbf{x}_1^\wedge ，得：

$$s_1 \mathbf{x}_1^\wedge \mathbf{x}_1 = 0 = s_2 \mathbf{x}_1^\wedge \mathbf{R} \mathbf{x}_2 + \mathbf{x}_1^\wedge \mathbf{t}. \quad (7.25)$$

该式左侧为零，右侧可看成 s_2 的一个方程，可以根据它直接求得 s_2 。有了 s_2 ， s_1 也非常容易求出。于是，我们就得到了两个帧下的点的深度，确定了它们的空间坐标。当然，由于噪声的存在，我们估得的 \mathbf{R}, \mathbf{t} ，不一定精确使式 (7.24) 为零，所以更常见的做法求最小二乘解而不是零解。

7.6 实践：三角测量

7.6.1 三角测量代码

下面，我们演示如何根据之前根据对极几何求解的相机位姿，通过三角化求出上一节特征点的空间位置。我们调用 OpenCV 提供的 triangulation 函数进行三角化。

slambook/ch7/triangulation.cpp (片断)

```
1 void triangulation (
2     const vector<KeyPoint>& keypoint_1,
3     const vector<KeyPoint>& keypoint_2,
4     const std::vector< DMatch >& matches,
5     const Mat& R, const Mat& t,
6     vector<Point3d>& points
7 );
8
9 void triangulation (
10    const vector< KeyPoint >& keypoint_1,
11    const vector< KeyPoint >& keypoint_2,
12    const std::vector< DMatch >& matches,
13    const Mat& R, const Mat& t,
14    vector< Point3d >& points )
15 {
16     Mat T1 = (Mat_<double> (3,4) <<
17         1,0,0,0,
18         0,1,0,0,
19         0,0,1,0);
20     Mat T2 = (Mat_<double> (3,4) <<
21         R.at<double>(0,0), R.at<double>(0,1), R.at<double>(0,2), t.at<double>(0,0),
22         R.at<double>(1,0), R.at<double>(1,1), R.at<double>(1,2), t.at<double>(1,0),
23         R.at<double>(2,0), R.at<double>(2,1), R.at<double>(2,2), t.at<double>(2,0)
24 );
25
26     Mat K = ( Mat_<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );
27     vector<Point2d> pts_1, pts_2;
28     for ( DMatch m:matches )
29     {
30         // 将像素坐标转换至相机坐标
31         pts_1.push_back ( pixel2cam( keypoint_1[m.queryIdx].pt, K ) );
32         pts_2.push_back ( pixel2cam( keypoint_2[m.trainIdx].pt, K ) );
33     }
34
35     Mat pts_4d;
36     cv::triangulatePoints( T1, T2, pts_1, pts_2, pts_4d );
37
38     // 转换成非齐次坐标
39     for ( int i=0; i<pts_4d.cols; i++ )
```

```

40    {
41        Mat x = pts_4d.col(i);
42        x /= x.at<float>(3,0); // 归一化
43        Point3d p (
44            x.at<float>(0,0),
45            x.at<float>(1,0),
46            x.at<float>(2,0)
47        );
48        points.push_back( p );
49    }
50 }

```

同时，在 main 函数中增加三角测量部分，并验证重投影关系：

```

1 int main (int argc, char** argv)
2 {
3     // ....
4     //--- 三角化
5     vector<Point3d> points;
6     triangulation( keypoints_1, keypoints_2, matches, R, t, points );
7
8     //--- 验证三角化点与特征点的重投影关系
9     Mat K = ( Mat<double> ( 3,3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );
10    for ( int i=0; i<matches.size(); i++ )
11    {
12        Point2d pt1_cam = pixel2cam( keypoints_1[ matches[i].queryIdx ].pt, K );
13        Point2d pt1_cam_3d (
14            points[i].x/points[i].z,
15            points[i].y/points[i].z
16        );
17
18        cout<<"point in the first camera frame: "<<pt1_cam<<endl;
19        cout<<"point projected from 3D "<<pt1_cam_3d<<, d=><<points[i].z<<endl;
20
21        // 第二个图
22        Point2f pt2_cam = pixel2cam( keypoints_2[ matches[i].trainIdx ].pt, K );
23        Mat pt2_trans = R*( Mat<double>(3,1) << points[i].x, points[i].y, points[i].z ) + t;
24        pt2_trans /= pt2_trans.at<double>(2,0);
25        cout<<"point in the second camera frame: "<<pt2_cam<<endl;
26        cout<<"point reprojected from second frame: "<<pt2_trans.t()<<endl;
27        cout<<endl;
28    }
29    // ...
30 }

```

我们打印了每个空间点在两个相机坐标系下的投影坐标与像素坐标——相当于 P 的投影位置与看到的特征点位置。由于误差的存在，它们会有一些微小的差异。以下是某一特征点的信息：

```
1 point in the first camera frame: [0.0844072, -0.0734976]
```

```

2 point projected from 3D [0.0843702, -0.0743606], d=14.9895
3 point in the second camera frame: [0.0431343, -0.0459876]
4 point reprojected from second frame: [0.04312769812378599, -0.04515455276163744, 1]

```

可以看到，误差的量级大约在小数点后第三位。可以看到，三角化特征点的距离大约为 15。但由于尺度不确定性，我们并不知道这里的 15 究竟是多少米。

7.6.2 讨论

关于三角测量，还有一个必须注意的地方。

三角测量是由平移得到的，有平移才会有对极几何中的三角形，才谈得上三角测量。因此，纯旋转是无法使用三角测量的，因为对极约束将永远满足。在平移存在的情况下，我们还要关心三角测量的不确定性，这会引出一个**三角测量的矛盾**。

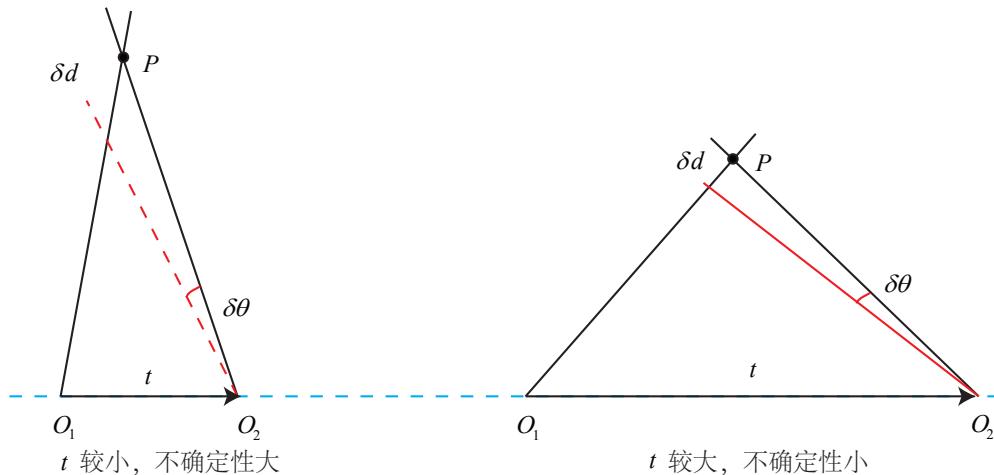


图 7-10 三角测量的矛盾。

如图 7-10 所示。当平移很小时，像素上的不确定性将导致较大的深度不确定性。也就是说，如果特征点运动一个像素 δx ，使得视线角变化了一个角度 $\delta\theta$ ，那么测量到深度值将有 δd 的变化。从几何关系可以看到，当 t 较大时， δd 将明显变小，这说明平移较大时，在同样的相机分辨率下，三角化测量将更精确。对该过程的定量分析可以使用正弦定理得到，但我们这里先考虑定性分析。

因此，要增加三角化的精度，其一是提高特征点的提取精度，也就是提高图像分辨率——但这会导致图像变大，提高计算成本。另一方式是使平移量增大。但是，平移量增大，

会导致图像的外观发生明显的变化，比如箱子原先被挡住的侧面显示出来了，比如反射光发生了变化了，等等。外观变化会使得特征提取与匹配变得困难。总而言之，在增大平移，会导致匹配失效；而平移太小，则三角化精度不够——这就是三角化的矛盾。

虽然本节只介绍了三角化的深度估计，但只要我们愿意，也能够定量地计算每个特征点的位置及不确定性。所以，如果假设特征点服从高斯分布，并且对它不断地进行观测，在信息正确的情况下，我们就能够期望它的方差会不断减小乃至收敛。这就得到了一个滤波器，称为深度滤波器（Depth Filter）。不过，由于它的原理较复杂，我们留到第 13 讲再详细讨论它。下面，我们来讨论从 3D-2D 的匹配点来估计相机运动，以及 3D-3D 的估计方法。

7.7 3D-2D: PnP

PnP (Perspective-n-Point) 是求解 3D 到 2D 点对运动的方法。它描述了当我们知道 n 个 3D 空间点以及它们的投影位置时，如何估计相机所在的位姿。前面已经说了，2D-2D 的对极几何方法需要八个或八个以上的点对（以八点法为例），且存在着初始化、纯旋转和尺度的问题。然而，如果两张图像中，其中一张特征点的 3D 位置已知，那么最少只需三个点对（需要至少一个额外点验证结果）就可以估计相机运动。特征点的 3D 位置可以由三角化，或者由 RGB-D 相机的深度图确定。因此，在双目或 RGB-D 的视觉里程计中，我们可以直接使用 PnP 估计相机运动。而在单目视觉里程计中，必须先进行初始化，然后才能使用 PnP。3D-2D 方法不需要使用对极约束，又可以在很少的匹配点中获得较好的运动估计，是最重要的一种姿态估计方法。

PnP 问题有很多种求解方法，例如用三对点估计位姿的 P3P[45]，直接线性变换 (DLT)，EPnP (Efficient PnP) [46]，UPnP[47] 等等。此外，还能用非线性优化的方式，构建最小二乘问题并迭代求解，也就是万金油式的 Bundle Adjustment。我们先来看 DLT，然后再讲 Bundle Adjustment。

7.7.1 直接线性变换

考虑某个空间点 P ，它的齐次坐标为 $\mathbf{P} = (X, Y, Z, 1)^T$ 。在图像 I_1 中，投影到特征点 $\mathbf{x}_1 = (u_1, v_1, 1)^T$ （以归一化平面齐次坐标表示）。此时相机的位姿 \mathbf{R}, \mathbf{t} 是未知的。与单应矩阵的求解类似，我们定义增广矩阵 $[\mathbf{R}|\mathbf{t}]$ 为一个 3×4 的矩阵，包含了旋转与平移信息^①。我们把它的展开形式列写如下：

^①请注意这和 $SE(3)$ 中的变换矩阵 \mathbf{T} 是不同的。

$$s \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix} = \begin{pmatrix} t_1 & t_2 & t_3 & t_4 \\ t_5 & t_6 & t_7 & t_8 \\ t_9 & t_{10} & t_{11} & t_{12} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}. \quad (7.26)$$

用最后一行把 s 消去, 得到两个约束:

$$u_1 = \frac{t_1X + t_2Y + t_3Z + t_4}{t_9X + t_{10}Y + t_{11}Z + t_{12}} \quad v_1 = \frac{t_5X + t_6Y + t_7Z + t_8}{t_9X + t_{10}Y + t_{11}Z + t_{12}}.$$

为了简化表示, 定义 \mathbf{T} 的行向量:

$$\mathbf{t}_1 = (t_1, t_2, t_3, t_4)^T, \mathbf{t}_2 = (t_5, t_6, t_7, t_8)^T, \mathbf{t}_3 = (t_9, t_{10}, t_{11}, t_{12})^T,$$

于是有:

$$\mathbf{t}_1^T \mathbf{P} - \mathbf{t}_3^T \mathbf{P} u_1 = 0,$$

和

$$\mathbf{t}_2^T \mathbf{P} - \mathbf{t}_3^T \mathbf{P} v_1 = 0.$$

请注意 \mathbf{t} 是待求的变量, 可以看到每个特征点提供了两个关于 \mathbf{t} 的线性约束。假设一共有 N 个特征点, 可以列出线性方程组:

$$\begin{pmatrix} \mathbf{P}_1^T & 0 & -u_1 \mathbf{P}_1^T \\ 0 & \mathbf{P}_1^T & -v_1 \mathbf{P}_1^T \\ \vdots & \vdots & \vdots \\ \mathbf{P}_N^T & 0 & -u_N \mathbf{P}_N^T \\ 0 & \mathbf{P}_N^T & -v_N \mathbf{P}_N^T \end{pmatrix} \begin{pmatrix} \mathbf{t}_1 \\ \mathbf{t}_2 \\ \mathbf{t}_3 \end{pmatrix} = 0. \quad (7.27)$$

由于 \mathbf{t} 一共有 12 维, 因此最少通过六对匹配点, 即可实现矩阵 \mathbf{T} 的线性求解, 这种方法(也)称为直接线性变换(Direct Linear Transform, DLT)。当匹配点大于六对时, (又)可以使用 SVD 等方法对超定方程求最小二乘解。

在 DLT 求解中, 我们直接将 \mathbf{T} 矩阵看成了 12 个未知数, 忽略了它们之间的联系。因为旋转矩阵 $\mathbf{R} \in SO(3)$, 用 DLT 求出的解不一定满足该约束, 它是一个一般矩阵。平移向量比较好办, 它属于向量空间。对于旋转矩阵 \mathbf{R} , 我们必须针对 DLT 估计的 \mathbf{T} 的左边

3×3 的矩阵块，寻找一个最好的旋转矩阵对它进行近似。这可以由 QR 分解完成 [3, 48]，相当于把结果从矩阵空间重新投影到 $SE(3)$ 流形上，转换成旋转和平移两部分。

需要解释的是，我们这里的 x_1 使用了归一化平面坐标，去掉了内参矩阵 \mathbf{K} 的影响——这是因为内参 \mathbf{K} 在 SLAM 中通常假设为已知。如果内参未知，那么我们也能用 PnP 去估计 $\mathbf{K}, \mathbf{R}, \mathbf{t}$ 三个量。然而由于未知量的增多，效果会差一些。

7.7.2 P3P

下面讲的 P3P 是另一种解 PnP 的方法。它仅使用三对匹配点，对数据要求较少，因此这里也简单介绍一下（这部分推导借鉴了 [49]）。

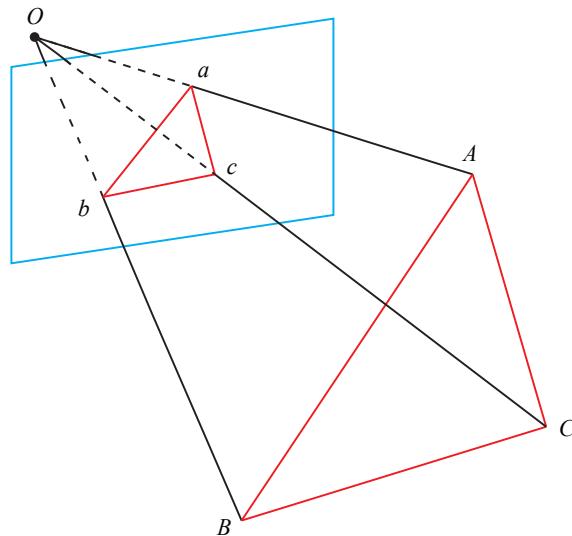


图 7-11 P3P 问题示意图。

P3P 需要利用给定的三个点的几何关系。它的输入数据为三对 3D-2D 匹配点。记 3D 点为 A, B, C , 2D 点为 a, b, c , 其中小写字母代表的点为大写字母在相机成像平面上的投影，如图 7-11 所示。此外，P3P 还需要使用一对验证点，以从可能的解出选出正确的那个（类似于对极几何情形）。记验证点对为 $D - d$ ，相机光心为 O 。请注意，我们知道的是 A, B, C 在世界坐标系中的坐标，而不是在相机坐标系中的坐标。一旦 3D 点在相机坐标系下的坐标能够算出，我们就得到了 3D-3D 的对应点，把 PnP 问题转换为了 ICP 问题。

首先，显然，三角形之间存在对应关系：

$$\Delta Oab = \Delta OAB, \quad \Delta Obc = \Delta OBC, \quad \Delta Oac = \Delta OAC. \quad (7.28)$$

来考虑 Oab 和 OAB 的关系。利用余弦定理，有：

$$OA^2 + OB^2 - 2OA \cdot OB \cdot \cos \langle a, b \rangle = AB^2. \quad (7.29)$$

对于其他两个三角形亦有类似性质，于是有：

$$\begin{aligned} OA^2 + OB^2 - 2OA \cdot OB \cdot \cos \langle a, b \rangle &= AB^2 \\ OB^2 + OC^2 - 2OB \cdot OC \cdot \cos \langle b, c \rangle &= BC^2 \\ OA^2 + OC^2 - 2OA \cdot OC \cdot \cos \langle a, c \rangle &= AC^2. \end{aligned} \quad (7.30)$$

对上面三式全体除以 OC^2 ，并且记 $x = OA/OC, y = OB/OC$ ，得：

$$\begin{aligned} x^2 + y^2 - 2xy \cos \langle a, b \rangle &= AB^2/OC^2 \\ y^2 + 1^2 - 2y \cos \langle b, c \rangle &= BC^2/OC^2 \\ x^2 + 1^2 - 2x \cos \langle a, c \rangle &= AC^2/OC^2. \end{aligned} \quad (7.31)$$

记 $v = AB^2/OC^2, uv = BC^2/OC^2, wv = AC^2/OC^2$ ，有：

$$\begin{aligned} x^2 + y^2 - 2xy \cos \langle a, b \rangle - v &= 0 \\ y^2 + 1^2 - 2y \cos \langle b, c \rangle - uv &= 0 \\ x^2 + 1^2 - 2x \cos \langle a, c \rangle - wv &= 0. \end{aligned} \quad (7.32)$$

我们可以把第一个式子中的 v 放到等式一边，并代入第 2, 3 两式，得：

$$\begin{aligned} (1-u)y^2 - ux^2 - \cos \langle b, c \rangle y + 2uxy \cos \langle a, b \rangle + 1 &= 0 \\ (1-w)x^2 - wy^2 - \cos \langle a, c \rangle x + 2wxy \cos \langle a, b \rangle + 1 &= 0. \end{aligned} \quad (7.33)$$

注意这些方程中的已知量和未知量。由于我们知道 2D 点的图像位置，三个余弦角 $\cos \langle a, b \rangle, \cos \langle b, c \rangle, \cos \langle a, c \rangle$ 是已知的。同时， $u = BC^2/AB^2, w = AC^2/AB^2$ 可以通过 A, B, C 在世界坐标系下的坐标算出，变换到相机坐标系下之后，并不改变这个比值。该式中的 x, y 是未知的，随着相机移动会发生变化。因此，该方程组是关于 x, y 的一个二元二次方程（多项式方程）。解析地求解该方程组是一个复杂的过程，需要用吴消元法。这里不展开对该方程解法的介绍，感兴趣的读者请参照 [45]。类似于分解 E 的情况，该方程最多可能得到四个解，但我们可以用验证点来计算最可能的解，得到 A, B, C 在相机坐标系下

的 3D 坐标。然后，根据 3D-3D 的点对，计算相机的运动 \mathbf{R}, \mathbf{t} 。这部分将在 7.9 小结介绍。

从 P3P 的原理上可以看出，为了求解 PnP，我们利用了三角形相似性质，求解投影点 a, b, c 在相机坐标系下的 3D 坐标，最后把问题转换成一个 3D 到 3D 的位姿估计问题。后文将看到，带有匹配信息的 3D-3D 位姿求解非常容易，所以这种思路是非常有效的。其他的一些方法，例如 EPnP，亦采用了这种思路。然而，P3P 也存在着一些问题：

1. P3P 只利用三个点的信息。当给定的配对点多于 3 组时，难以利用更多的信息。
2. 如果 3D 点或 2D 点受噪声影响，或者存在误匹配，则算法失效。

所以后续人们还提出了许多别的方法，如 EPnP、UPnP 等。它们利用更多的信息，而且用迭代的方式对相机位姿进行优化，以尽可能地消除噪声的影响。不过，相对于 P3P 来说，原理会更加复杂一些，所以我们建议读者阅读原始的论文，或通过实践来理解 PnP 过程。在 SLAM 当中，通常的做法是先使用 P3P/EPnP 等方法估计相机位姿，然后构建最小二乘优化问题对估计值进行调整（Bundle Adjustment）。接下来我们从非线性优化角度来看一下 PnP 问题。

7.7.3 Bundle Adjustment

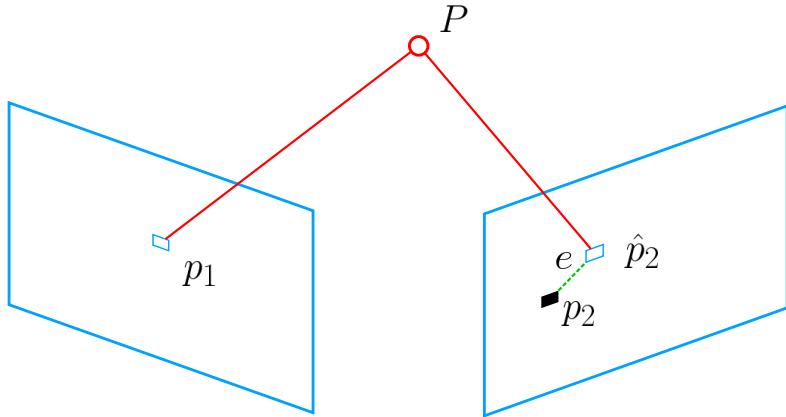


图 7-12 重投影误差示意图。

除了使用线性方法之外，我们可以把 PnP 问题构建成一个定义于李代数上的非线性最小二乘问题。这将用到本书第四章和第五章的知识。前面说的线性方法，往往是先求相

机位姿, 再求空间点位置, 而非线性优化则是把它们都看成优化变量, 放在一起优化。这是一种非常通用的求解方式, 我们可以用它对 PnP 或 ICP 给出的结果进行优化。在 PnP 中, 这个 Bundle Adjustment 问题, 是一个最小化重投影误差 (Reprojection error) 的问题。我们在本节给出此问题在两个视图下的基本形式, 然后在第十讲讨论较大规模的 BA 问题。

考虑 n 个三维空间点 P 和它们的投影 p , 我们希望计算相机的位姿 \mathbf{R}, \mathbf{t} , 它的李代数表示为 $\boldsymbol{\xi}$ 。假设某空间点坐标为 $\mathbf{P}_i = [X_i, Y_i, Z_i]^T$, 其投影的像素坐标为 $\mathbf{u}_i = [u_i, v_i]^T$ 。根据第五章的内容, 像素位置与空间点位置的关系如下:

$$s_i \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \mathbf{K} \exp(\boldsymbol{\xi}^\wedge) \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}. \quad (7.34)$$

除了用 $\boldsymbol{\xi}$ 为李代数表示的相机姿态之外, 别的都和前面的定义保持一致。写成矩阵形式就是:

$$s_i \mathbf{u}_i = \mathbf{K} \exp(\boldsymbol{\xi}^\wedge) \mathbf{P}_i.$$

请读者脑补中间隐含着的齐次坐标到非齐次的转换, 否则按矩阵的乘法来说, 维度是不对的^①。现在, 由于相机位姿未知以及观测点的噪声, 该等式存在一个误差。因此, 我们把误差求和, 构建最小二乘问题, 然后寻找最好的相机位姿, 使它最小化:

$$\boldsymbol{\xi}^* = \arg \min_{\boldsymbol{\xi}} \frac{1}{2} \sum_{i=1}^n \left\| \mathbf{u}_i - \frac{1}{s_i} \mathbf{K} \exp(\boldsymbol{\xi}^\wedge) \mathbf{P}_i \right\|_2^2. \quad (7.35)$$

该问题的误差项, 是将像素坐标 (观测到的投影位置) 与 3D 点按照当前估计的位姿进行投影得到的位置相比较得到的误差, 所以称之为重投影误差。使用齐次坐标时, 这个误差有 3 维。不过, 由于 \mathbf{u} 最后一维为 1, 该维度的误差一直为零, 因而我们更多时候使用非齐次坐标, 于是误差就只有 2 维了。如图 7-12 所示, 我们通过特征匹配, 知道了 p_1 和 p_2 是同一个空间点 P 的投影, 但是我们不知道相机的位姿。在初始值中, P 的投影 \hat{p}_2 与实际的 p_2 之间有一定的距离。于是我们调整相机的位姿, 使得这个距离变小。不过, 由于这个调整需要考虑很多个点, 所以最后每个点的误差通常都不会精确为零。

最小二乘优化问题已经在第六讲介绍过了。使用李代数, 可以构建无约束的优化问题,

^① $\exp(\boldsymbol{\xi}^\wedge) \mathbf{P}_i$ 结果是 4×1 的, 而它左侧的 \mathbf{K} 是 3×3 的, 所以必须把 $\exp(\boldsymbol{\xi}^\wedge) \mathbf{P}_i$ 的前三维取出来, 变成三维的非齐次坐标。这在前边章节说过

很方便地通过 G-N, L-M 等优化算法进行求解。不过，在使用 G-N 和 L-M 之前，我们需要知道每个误差项关于优化变量的导数，也就是线性化：

$$\mathbf{e}(\mathbf{x} + \Delta\mathbf{x}) \approx \mathbf{e}(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}. \quad (7.36)$$

这里的 \mathbf{J} 的形式是值得讨论的，甚至可以说是关键所在。我们固然可以使用数值导数，但如果能够推导解析形式时，我们会优先考虑解析导数。现在，当 \mathbf{e} 为像素坐标误差（2 维）， \mathbf{x} 为相机位姿（6 维）时， \mathbf{J} 将是一个 2×6 的矩阵。我们来推导 \mathbf{J} 的形式。

回忆李代数的内容，我们介绍了如何使用扰动模型来求李代数的导数。首先，记变换到相机坐标系下的空间点坐标为 \mathbf{P}' ，并且把它前三维取出来：

$$\mathbf{P}' = (\exp(\hat{\boldsymbol{\xi}}) \mathbf{P})_{1:3} = [X', Y', Z']^T. \quad (7.37)$$

那么，相机投影模型相对于 \mathbf{P}' 则为：

$$s\mathbf{u} = \mathbf{K}\mathbf{P}'. \quad (7.38)$$

展开之：

$$\begin{bmatrix} su \\ sv \\ s \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix}. \quad (7.39)$$

利用第 3 行消去 s （实际上就是 \mathbf{P}' 的距离），得：

$$u = f_x \frac{X'}{Z'} + c_x, \quad v = f_y \frac{Y'}{Z'} + c_y. \quad (7.40)$$

这与之前讲的相机模型是一致的。当我们求误差时，可以把这里的 u, v 与实际的测量值比较，求差。在定义了中间变量后，我们对 $\hat{\boldsymbol{\xi}}$ 左乘扰动量 $\delta\boldsymbol{\xi}$ ，然后考虑 \mathbf{e} 的变化关于扰动量的导数。利用链式法则，可以列写如下：

$$\frac{\partial \mathbf{e}}{\partial \delta\boldsymbol{\xi}} = \lim_{\delta\boldsymbol{\xi} \rightarrow 0} \frac{\mathbf{e}(\delta\boldsymbol{\xi} \oplus \hat{\boldsymbol{\xi}})}{\delta\boldsymbol{\xi}} = \frac{\partial \mathbf{e}}{\partial \mathbf{P}'} \frac{\partial \mathbf{P}'}{\partial \delta\boldsymbol{\xi}}. \quad (7.41)$$

这里的 \oplus 指李代数上的左乘扰动。第一项是误差关于投影点的导数，在式 (7.40) 已经列出了变量之间的关系，易得：

$$\frac{\partial e}{\partial \mathbf{P}'} = - \begin{bmatrix} \frac{\partial u}{\partial X'} & \frac{\partial u}{\partial Y'} & \frac{\partial u}{\partial Z'} \\ \frac{\partial v}{\partial X'} & \frac{\partial v}{\partial Y'} & \frac{\partial v}{\partial Z'} \end{bmatrix} = - \begin{bmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} \end{bmatrix}. \quad (7.42)$$

而第二项为变换后的点关于李代数的导数，根据在4.3.5中的推导，得：

$$\frac{\partial (\mathbf{T}\mathbf{P})}{\partial \delta \xi} = (\mathbf{T}\mathbf{P})^\odot = \begin{bmatrix} \mathbf{I} & -\mathbf{P}'^\wedge \\ \mathbf{0}^T & \mathbf{0}^T \end{bmatrix}. \quad (7.43)$$

而在 \mathbf{P}' 的定义中，我们取出了前三维，于是得：

$$\frac{\partial \mathbf{P}'}{\partial \delta \xi} = [\mathbf{I}, -\mathbf{P}'^\wedge]. \quad (7.44)$$

将这两项相乘，就得到了 2×6 的雅可比矩阵：

$$\frac{\partial e}{\partial \delta \xi} = - \begin{bmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} & -\frac{f_x X' Y'}{Z'^2} & f_x + \frac{f_x X^2}{Z'^2} & -\frac{f_x Y'}{Z'} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} & -f_y - \frac{f_y Y'^2}{Z'^2} & \frac{f_y X' Y'}{Z'^2} & \frac{f_y X'}{Z'} \end{bmatrix}. \quad (7.45)$$

这个雅可比矩阵描述了重投影误差关于相机位姿李代数的一阶变化关系。我们保留了前面的负号，因为这是由于误差是由观测值减预测值定义的。它当然也可反过来，定义成“预测减观测”的形式。在那种情况下，只要去掉前面的负号即可。此外，如果 $\mathfrak{se}(3)$ 的定义方式是旋转在前，平移在后时，只要把这个矩阵的前三列与后三列对调即可。

另一方面，除了优化位姿，我们还希望优化特征点的空间位置。因此，需要讨论 e 关于空间点 \mathbf{P} 的导数。所幸这个导数矩阵相对来说容易一些。仍利用链式法则，有：

$$\frac{\partial e}{\partial \mathbf{P}} = \frac{\partial e}{\partial \mathbf{P}'} \frac{\partial \mathbf{P}'}{\partial \mathbf{P}}. \quad (7.46)$$

第一项已在前面推导了，第二项，按照定义

$$\mathbf{P}' = \exp(\xi^\wedge) \mathbf{P} = \mathbf{R}\mathbf{P} + \mathbf{t}.$$

我们发现 \mathbf{P}' 对 \mathbf{P} 求导后只剩下 \mathbf{R} 。于是

$$\frac{\partial e}{\partial \mathbf{P}} = - \begin{bmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} \end{bmatrix} \mathbf{R}. \quad (7.47)$$

于是，我们推导了观测相机方程关于相机位姿与特征点的两个导数矩阵。它们十分重要，能够在优化过程中提供重要的梯度方向，指导优化的迭代。

7.8 实践：求解 PnP

7.8.1 使用 EPnP 求解位姿

下面，我们通过实验理解一下 PnP 的过程。首先，我们用 OpenCV 提供的 EPnP 求解 PnP 问题，然后通过 g2o 对结果进行优化。由于 PnP 需要使用 3D 点，为了避免初始化带来的麻烦，我们使用了 RGB-D 相机中的深度图（1_depth.png），作为特征点的 3D 位置。首先来看 OpenCV 提供的 PnP 函数：

`slambook/ch7/pose_estimation_3d2d.cpp` (片段)

```

1 int main( int argc, char** argv )
2 {
3     .....
4     // 建立 3D 点
5     Mat d1 = imread( argv[3], CV_LOAD_IMAGE_UNCHANGED ); // 深度图为 16 位无符号数，单通道图像
6     Mat K = ( Mat<double> ( 3, 3 ) << 520.9, 0, 325.1, 0, 521.0, 249.7, 0, 0, 1 );
7     vector<Point3f> pts_3d;
8     vector<Point2f> pts_2d;
9     for ( DMatch m:matches )
10    {
11        ushort d = d1.ptr<unsigned short> ( int(keypoints_1[m.queryIdx].pt.y) )[ int(keypoints_1[m.
12           queryIdx].pt.x) ];
13        if ( d == 0 ) // bad depth
14            continue;
15        float dd = d/1000.0;
16        Point2d p1 = pixel2cam( keypoints_1[m.queryIdx].pt, K );
17        pts_3d.push_back( Point3f(p1.x*dd, p1.y*dd, dd) );
18        pts_2d.push_back ( keypoints_2[m.trainIdx].pt );
19    }
20
21    cout<<"3d-2d pairs: "<<pts_3d.size()<<endl;
22
23    Mat r, t;
24    // 调用 OpenCV 的 PnP 求解，可选择 EPnP , DLS 等方法
25    solvePnP( pts_3d, pts_2d, K, Mat(), r, t, false, cv::SOLVEPNP_EPNP );
26    Mat R;
27    cv::Rodrigues(r, R); // r 为旋转向量形式，用 Rodrigues 公式转换为矩阵
28
29    cout<<"R="<<endl<<R<<endl;
30    cout<<"t="<<endl<<t<<endl;
}

```

在例程中，我们得到配对特征点后，在第一个图的深度图中寻找它们的深度，并求出

空间位置。以此空间位置为 3D 点，再以第二个图像的像素位置为 2D 点，调用 EPnP 求解 PnP 问题。程序输出如下：

```

1 % build/pose_estimation_3d2d 1.png 2.png d1.png d2.png
2 -- Max dist : 95.000000
3 -- Min dist : 4.000000
4 一共找到了79组匹配点
5 3d-2d pairs: 78
6 R=
7 [0.9977970937403702, -0.05195299069131867, 0.04125344205637558;
8 0.05073872610592159, 0.9982626103770279, 0.02995567385972873;
9 -0.04273805559942161, -0.02779653722084675, 0.9986995599889442]
10 t=
11 [-0.6455324432075111;
12 -0.05776758294184359;
13 0.2844565219506077]
```

读者可以对比先前 2D-2D 情况下求解的 R, t 有什么不同。可以看到，在有 3D 信息时，估计的 R 几乎是相同的，而 t 相差的较多。这是由于我们引入了新的深度信息所致。不过，由于 Kinect 采集的深度图本身会有一些误差，所以这里的 3D 点也不是准确的。我们会希望把位姿 ξ 和所有三维特征点 P 同时优化。

7.8.2 使用 BA 优化

下面，我们来演示如何进行 Bundle Adjustment。我们将使用前一步的估计值作为初始值。优化可以使用前面讲的 Ceres 或 g2o 库实现，这里采用 g2o 作为例子。

g2o 的基本知识在第六讲中已经介绍过了。在使用 g2o 之前，我们要把问题建模成一个最小二乘的图优化问题，如图 7-13 所示。在这个图优化中，节点和边的选择为：

1. 节点：第二个相机的位姿节点 $\xi \in \mathfrak{se}(3)$ ，以及所有特征点的空间位置 $P \in \mathbb{R}^3$ 。
2. 边：每个 3D 点在第二个相机中的投影，以观测方程来描述：

$$z_j = h(\xi, P_j).$$

由于第一个相机位姿固定为零，我们没有把它写到优化变量里，但在习题中，我希望你能够把第一个相机的位姿与观测也考虑进来。现在我们根据一组 3D 点和第二个图像中的 2D 投影，估计第二个相机的位姿。所以我们把第一个相机画成虚线，表明我们不希望考虑它。

g2o 提供了许多关于 BA 的节点和边，我们不必自己从头实现所有的计算。在 g2o/types/sba/types_six_dof_expmap.h 中则提供了李代数表达的节点和边。请读者打

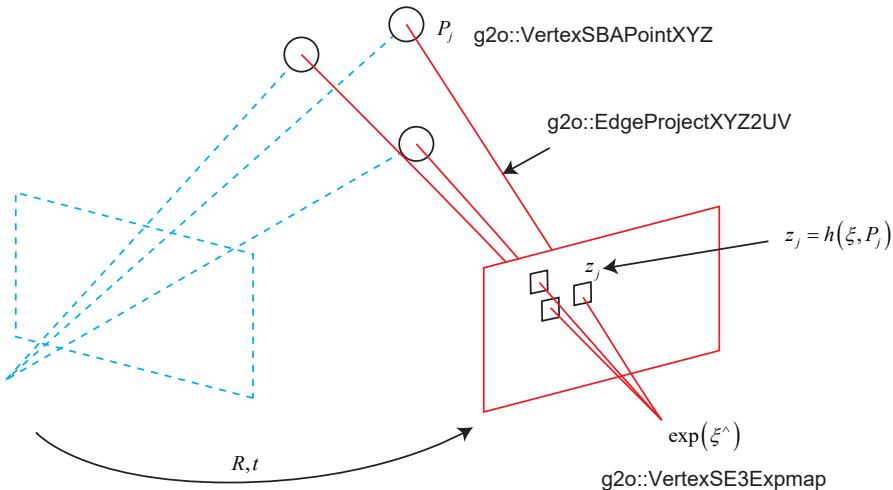


图 7-13 PnP 的 Bundle Adjustment 的图优化表示。

开这个文件，找到 `VertexSE3Expmap`（李代数位姿）、`VertexSBAPointXYZ`（空间点位置）和 `EdgeProjectXYZ2UV`（投影方程边）这三个类。我们来简单看一下它们的类定义，例如 `VertexSE3Expmap`：

```

1 class G2O_TYPES_SBA_API VertexSE3Expmap : public BaseVertex<6, SE3Quat>{
2 public:
3     EIGEN_MAKE_ALIGNED_OPERATOR_NEW
4
5     VertexSE3Expmap();
6
7     bool read(std::istream& is);
8
9     bool write(std::ostream& os) const;
10
11    virtual void setToOriginImpl() {
12        _estimate = SE3Quat();
13    }
14
15    virtual void oplusImpl(const double* update_) {
16        Eigen::Map<const Vector6d> update(update_);
17        setEstimate( SE3Quat::exp(update)*estimate());
18    }
19};

```

请注意它的模板参数。第一个参数 6 表示它内部存储的优化变量维度，可以看到这是一个 6 维的李代数。第二参数是优化变量的类型，这里使用了 g2o 定义的相机位姿：`SE3Quat`。这个类内部使用了四元数加位移向量来存储位姿，但同时也支持李代数上的运算，例如对

数映射 (log 函数) 和李代数上增量 (update 函数) 等操作。我们可以对照它的实现代码，看看 g2o 对李代数是如何操作的：

```

1 class G2O_TYPES_SBA_API VertexSBAPointXYZ : public BaseVertex<3, Vector3D>
2 {
3     .....
4 };
5
6 class G2O_TYPES_SBA_API EdgeProjectXYZ2UV : public BaseBinaryEdge<2, Vector2D, VertexSBAPointXYZ,
7 VertexSE3Expmap>
8 {
9     .....
10    void computeError() {
11        const VertexSE3Expmap* v1 = static_cast<const VertexSE3Expmap*>(_vertices[1]);
12        const VertexSBAPointXYZ* v2 = static_cast<const VertexSBAPointXYZ*>(_vertices[0]);
13        const CameraParameters * cam
14            = static_cast<const CameraParameters *>(parameter(0));
15        Vector2D obs(_measurement);
16        _error = obs.cam->cam_map(v1->estimate().map(v2->estimate()));
17    }
18};

```

我就不把整个类定义都搬过来了。从模板参数可以看到，空间点位置类的维度为 3，类型是 Eigen 的 Vector3D。另一方面，边 EdgeProjectXYZ2UV 连接了两个前面说的两个顶点，它的观测值为 2 维，由 Vector2D 表示，实际上就是空间点的像素坐标。它的误差计算函数表达了投影方程的误差计算方法，也就是我们前面提到的 $z - h(\xi, \mathbf{P})$ 的方式。

现在，进一步观察 EdgeProjectXYZ2UV 的 linearizeOplus 函数的实现。这里用到了我们前面推导的雅可比矩阵：

```

1 void EdgeProjectXYZ2UV::linearizeOplus() {
2     VertexSE3Expmap * vj = static_cast<VertexSE3Expmap *>(_vertices[1]);
3     SE3Quat T(vj->estimate());
4     VertexSBAPointXYZ* vi = static_cast<VertexSBAPointXYZ*>(_vertices[0]);
5     Vector3D xyz = vi->estimate();
6     Vector3D xyz_trans = T.map(xyz);
7
8     double x = xyz_trans[0];
9     double y = xyz_trans[1];
10    double z = xyz_trans[2];
11    double z_2 = z*z;
12
13    const CameraParameters * cam = static_cast<const CameraParameters *>(parameter(0));
14
15    Matrix<double,2,3,Eigen::ColMajor> tmp;
16    tmp(0,0) = cam->focal_length;
17    tmp(0,1) = 0;
18    tmp(0,2) = -x/z*cam->focal_length;
19

```

```

20     tmp(1,0) = 0;
21     tmp(1,1) = cam->focal_length;
22     tmp(1,2) = -y/z*cam->focal_length;
23
24     _jacobianOplusXi = -1./z * tmp * T.rotation().toRotationMatrix();
25
26     _jacobianOplusXj(0,0) = x*y/z_2 *cam->focal_length;
27     _jacobianOplusXj(0,1) = -(1+(x*x/z_2)) *cam->focal_length;
28     _jacobianOplusXj(0,2) = y/z *cam->focal_length;
29     _jacobianOplusXj(0,3) = -1./z *cam->focal_length;
30     _jacobianOplusXj(0,4) = 0;
31     _jacobianOplusXj(0,5) = x/z_2 *cam->focal_length;
32
33     _jacobianOplusXj(1,0) = (1+y*y/z_2) *cam->focal_length;
34     _jacobianOplusXj(1,1) = -x*y/z_2 *cam->focal_length;
35     _jacobianOplusXj(1,2) = -x/z *cam->focal_length;
36     _jacobianOplusXj(1,3) = 0;
37     _jacobianOplusXj(1,4) = -1./z *cam->focal_length;
38     _jacobianOplusXj(1,5) = y/z_2 *cam->focal_length;
39 }

```

仔细研究此段代码，我们会发现它与式 (7.45) 和 (7.47) 是一致的。成员变量 “`_jacobianOplusXi`” 是误差到空间点的导数，“`_jacobianOplusXj`” 是误差到相机位姿的导数，以李代数的左乘扰动表达。稍有差别的是，g2o 的相机里用 f 统一描述 f_x, f_y ，并且李代数定义顺序不同（g2o 是旋转在前，平移在后；我们是平移在前，旋转在后），所以矩阵前三列和后三列与我们的定义是颠倒的。此外都是一致的。

值得一提的是，我们亦可自己实现相机位姿节点，并使用 Sophus::SE3 来表达位姿，提供类似的求导过程。然而，既然 g2o 已经提供了这样的类，在没有额外要求的情况下，自己重新实现就没有必要了。现在，我们在上一个 PnP 例程的基础上，加上 g2o 提供的 Bundle Adjustment。

slambook/ch7/pose_estimation_3d2d.cpp (片段)

```

1 void bundleAdjustment (
2     const vector< Point3f > points_3d,
3     const vector< Point2f > points_2d,
4     const Mat& K,
5     Mat& R, Mat& t )
6 {
7     // 初始化g2o
8     typedef g2o::BlockSolver< g2o::BlockSolverTraits<6,3> > Block; // pose 维度为 6, landmark 维度为 3
9     Block::LinearSolverType* linearSolver = new g2o::LinearSolverCSpars<Block::PoseMatrixType>();
10    Block* solver_ptr = new Block( linearSolver );
11    g2o::OptimizationAlgorithmLevenberg* solver = new g2o::OptimizationAlgorithmLevenberg( solver_ptr )
12 ;

```

```
12 g2o::SparseOptimizer optimizer;
13 optimizer.setAlgorithm( solver );
14
15 // vertex
16 g2o::VertexSE3Expmap* pose = new g2o::VertexSE3Expmap(); // camera pose
17 Eigen::Matrix3d R_mat;
18 R_mat <<
19     R.at<double>(0,0), R.at<double>(0,1), R.at<double>(0,2),
20     R.at<double>(1,0), R.at<double>(1,1), R.at<double>(1,2),
21     R.at<double>(2,0), R.at<double>(2,1), R.at<double>(2,2);
22 pose->setId(0);
23 pose->setEstimate( g2o::SE3Quat(
24     R_mat,
25     Eigen::Vector3d( t.at<double>(0,0), t.at<double>(1,0), t.at<double>(2,0) )
26 ) );
27 optimizer.addVertex( pose );
28
29 int index = 1;
30 for ( const Point3f p:points_3d ) // landmarks
31 {
32     g2o::VertexSBAPointXYZ* point = new g2o::VertexSBAPointXYZ();
33     point->setId( index++ );
34     point->setEstimate( Eigen::Vector3d(p.x, p.y, p.z) );
35     point->setMarginalized( true );
36     optimizer.addVertex( point );
37 }
38
39 // parameter: camera intrinsics
40 g2o::CameraParameters* camera = new g2o::CameraParameters(
41 K.at<double>(0,0), Eigen::Vector2d(K.at<double>(0,2), K.at<double>(1,2)), 0
42 );
43 camera->setId(0);
44 optimizer.addParameter( camera );
45
46 // edges
47 index = 1;
48 for ( const Point2f p:points_2d )
49 {
50     g2o::EdgeProjectXYZ2UV* edge = new g2o::EdgeProjectXYZ2UV();
51     edge->setId( index );
52     edge->setVertex( 0, dynamic_cast<g2o::VertexSBAPointXYZ*>(optimizer.vertex(index)) );
53     edge->setVertex( 1, pose );
54     edge->setMeasurement( Eigen::Vector2d( p.x, p.y ) );
55     edge->setParameterId(0,0);
56     edge->setInformation( Eigen::Matrix2d::Identity() );
57     optimizer.addEdge(edge);
58     index++;
59 }
60
61 chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
```

```

62     optimizer.setVerbose( true );
63     optimizer.initializeOptimization();
64     optimizer.optimize(100);
65     chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
66     chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double>>(t2-t1);
67     cout<<"optimization costs time: "<<time_used.count()<<" seconds."<<endl;
68
69     cout<<endl<<"after optimization:"<<endl;
70     cout<<"T="<<endl<<Eigen::Isometry3d( pose->estimate() ).matrix()<<endl;
71 }
```

程序大体上和第六章的 g2o 类似。我们首先声明了 g2o 图优化，配置优化求解器和梯度下降方法，然后根据估计到的特征点，将位姿和空间点放到图中。最后调用优化函数进行求解。读者可以看到优化的结果：

```

1 calling bundle adjustment
2 iteration= 0 chi2=  1.083180 time= 0.000107183 cumTime= 0.000107183 edges= 76 schur= 1 lambda=
78.907222 levenbergIter= 1
3 iteration= 1 chi2=  0.000798 time= 5.8615e-05 cumTime= 0.000165798 edges= 76 schur= 1 lambda= 26.302407
levenbergIter= 1
4 iteration= 2 chi2=  0.000000 time= 3.0203e-05 cumTime= 0.000196001 edges= 76 schur= 1 lambda= 17.534938
levenbergIter= 1
5 ..... 中间过程略
6 iteration= 11 chi2=  0.000000 time= 2.8394e-05 cumTime= 0.000525203 edges= 76 schur= 1 lambda=
11209.703029 levenbergIter= 1
7 optimization costs time: 0.00132938 seconds.

8
9 after optimization:
10 T=
11 0.997776 -0.0519476 0.0417755 -0.649778
12 0.050735 0.998274 0.0295806 -0.0545231
13 -0.0432401 -0.0273953 0.998689 0.295564
14 0 0 1
```

迭代 11 轮后，LM 发现优化目标函数接近不变，于是停止了优化。我们输出了最后得到位姿变换矩阵 T ，对比之前直接做 PnP 的结果，大约在小数点后第三位发生了一些变化。这主要是由于我们同时优化了特征点和相机位姿导致的。

Bundle Adjustment 是一种通用的做法。它可以不限于两个图像。我们完全可以放入多个图像匹配到的位姿和空间点进行迭代优化，甚至可以把整个 SLAM 过程放进来。那种做法规模较大，主要在后端使用，我们会在第十章重新遇到这个问题。在前端，我们通常考虑局部相机位姿和特征点的小型 Bundle Adjustment 问题，希望实时对它进行求解和优化。

7.9 3D-3D: ICP

最后，我们来介绍 3D-3D 的位姿估计问题。假设我们有一组配对好的 3D 点（比如我们对两个 RGB-D 图像进行了匹配）：

$$\mathbf{P} = \{\mathbf{p}_1, \dots, \mathbf{p}_n\}, \quad \mathbf{P}' = \{\mathbf{p}'_1, \dots, \mathbf{p}'_n\},$$

现在，想要找一个欧氏变换 \mathbf{R}, \mathbf{t} ，使得：

$$\forall i, \mathbf{p}_i = \mathbf{R}\mathbf{p}'_i + \mathbf{t}.$$

这个问题可以用迭代最近点（Iterative Closest Point, ICP）求解。读者应该注意到，3D-3D 位姿估计问题中，并没有出现相机模型，也就是说，仅考虑两组 3D 点之间的变换时，和相机并没有关系。因此，在激光 SLAM 中也会碰到 ICP，不过由于激光数据特征不够丰富，我们无从知道两个点集之间的匹配关系，只能认为距离最近的两个点为同一个，所以这个方法称为迭代最近点。而在视觉中，特征点为我们提供了较好的匹配关系，所以整个问题就变得更简单了。在 RGB-D SLAM 中，可以用这种方式估计相机位姿。下文我们用 ICP 指代匹配好的两组点间运动估计问题。

和 PnP 类似，ICP 的求解也分为两种方式：利用线性代数的求解（主要是 SVD），以及利用非线性优化方式的求解（类似于 Bundle Adjustment）。下面分别来介绍它们。

7.9.1 SVD 方法

首先我们看以 SVD 为代表的代数方法。根据前面描述的 ICP 问题，我们先定义第 i 对点的误差项：

$$\mathbf{e}_i = \mathbf{p}_i - (\mathbf{R}\mathbf{p}'_i + \mathbf{t}). \quad (7.48)$$

然后，构建最小二乘问题，求使误差平方和达到极小的 \mathbf{R}, \mathbf{t} ：

$$\min_{\mathbf{R}, \mathbf{t}} J = \frac{1}{2} \sum_{i=1}^n \|(\mathbf{p}_i - (\mathbf{R}\mathbf{p}'_i + \mathbf{t}))\|_2^2. \quad (7.49)$$

下面我们来推导它的求解方法。首先，定义两组点的质心：

$$\mathbf{p} = \frac{1}{n} \sum_{i=1}^n (\mathbf{p}_i), \quad \mathbf{p}' = \frac{1}{n} \sum_{i=1}^n (\mathbf{p}'_i). \quad (7.50)$$

请注意质心是没有下标的。随后，在误差函数中，我们作如下的处理：

$$\begin{aligned}\frac{1}{2} \sum_{i=1}^n \|\mathbf{p}_i - (\mathbf{R}\mathbf{p}'_i + \mathbf{t})\|^2 &= \frac{1}{2} \sum_{i=1}^n \|\mathbf{p}_i - \mathbf{R}\mathbf{p}'_i - \mathbf{t} - \mathbf{p} + \mathbf{R}\mathbf{p}' + \mathbf{p} - \mathbf{R}\mathbf{p}'\|^2 \\&= \frac{1}{2} \sum_{i=1}^n \|(\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}'_i - \mathbf{p}')) + (\mathbf{p} - \mathbf{R}\mathbf{p}' - \mathbf{t})\|^2 \\&= \frac{1}{2} \sum_{i=1}^n (\|\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}'_i - \mathbf{p}')\|^2 + \|\mathbf{p} - \mathbf{R}\mathbf{p}' - \mathbf{t}\|^2 + \\&\quad 2(\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}'_i - \mathbf{p}'))^T (\mathbf{p} - \mathbf{R}\mathbf{p}' - \mathbf{t})).\end{aligned}$$

注意到交叉项部分中， $(\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}'_i - \mathbf{p}'))$ 在求和之后是为零的，因此优化目标函数可以简化为：

$$\min_{\mathbf{R}, \mathbf{t}} J = \frac{1}{2} \sum_{i=1}^n \|\mathbf{p}_i - \mathbf{p} - \mathbf{R}(\mathbf{p}'_i - \mathbf{p}')\|^2 + \|\mathbf{p} - \mathbf{R}\mathbf{p}' - \mathbf{t}\|^2. \quad (7.51)$$

仔细观察左右两项，我们发现左边只和旋转矩阵 \mathbf{R} 相关，而右边既有 \mathbf{R} 也有 \mathbf{t} ，但只和质心相关。只要我们获得了 \mathbf{R} ，令第二项为零就能得到 \mathbf{t} 。于是，ICP 可以分为以下三个步骤求解：

1. 计算两组点的质心位置 \mathbf{p}, \mathbf{p}' ，然后计算每个点的去质心坐标：

$$\mathbf{q}_i = \mathbf{p}_i - \mathbf{p}, \quad \mathbf{q}'_i = \mathbf{p}'_i - \mathbf{p}'.$$

2. 根据以下优化问题计算旋转矩阵：

$$\mathbf{R}^* = \arg \min_{\mathbf{R}} \frac{1}{2} \sum_{i=1}^n \|\mathbf{q}_i - \mathbf{R}\mathbf{q}'_i\|^2. \quad (7.52)$$

3. 根据第二步的 \mathbf{R} ，计算 \mathbf{t} ：

$$\mathbf{t}^* = \mathbf{p} - \mathbf{R}\mathbf{p}'. \quad (7.53)$$

我们看到，只要求出了两组点之间的旋转，平移量是非常容易得到的。所以我们重点

关注 \mathbf{R} 的计算。展开关于 \mathbf{R} 的误差项，得：

$$\frac{1}{2} \sum_{i=1}^n \| \mathbf{q}_i - \mathbf{R} \mathbf{q}'_i \|^2 = \frac{1}{2} \sum_{i=1}^n \mathbf{q}_i^T \mathbf{q}_i + \mathbf{q}'_i^T \mathbf{R}^T \mathbf{R} \mathbf{q}'_i - 2 \mathbf{q}_i^T \mathbf{R} \mathbf{q}'_i. \quad (7.54)$$

注意到第一项和 \mathbf{R} 无关，第二项由于 $\mathbf{R}^T \mathbf{R} = \mathbf{I}$ ，亦与 \mathbf{R} 无关。因此，实际上优化目标函数变为：

$$\sum_{i=1}^n -\mathbf{q}_i^T \mathbf{R} \mathbf{q}'_i = \sum_{i=1}^n -\text{tr}(\mathbf{R} \mathbf{q}'_i \mathbf{q}_i^T) = -\text{tr}\left(\mathbf{R} \sum_{i=1}^n \mathbf{q}'_i \mathbf{q}_i^T\right). \quad (7.55)$$

接下来，我们介绍怎样通过 SVD 解出上述问题中最优的 \mathbf{R} ，但是关于最优性的证明较为复杂，感兴趣的读者请参考 [50, 51]。为了解 \mathbf{R} ，先定义矩阵：

$$\mathbf{W} = \sum_{i=1}^n \mathbf{q}_i \mathbf{q}'_i^T. \quad (7.56)$$

\mathbf{W} 是一个 3×3 的矩阵，对 \mathbf{W} 进行 SVD 分解，得：

$$\mathbf{W} = \mathbf{U} \Sigma \mathbf{V}^T. \quad (7.57)$$

其中， Σ 为奇异值组成的对角矩阵，对角线元素从大到小排列，而 \mathbf{U} 和 \mathbf{V} 为正交矩阵。当 \mathbf{W} 满秩时， \mathbf{R} 为：

$$\mathbf{R} = \mathbf{U} \Sigma^T \mathbf{V}. \quad (7.58)$$

解得 \mathbf{R} 后，按式 (7.53) 求解 \mathbf{t} 即可。

7.9.2 非线性优化方法

求解 ICP 的另一种方式是使用非线性优化，以迭代的方式去找最值。该方法和我们前面讲述的 PnP 非常相似。以李代数表达位姿时，目标函数可以写成：

$$\min_{\xi} = \frac{1}{2} \sum_{i=1}^n \| (\mathbf{p}_i - \exp(\xi^\wedge) \mathbf{p}'_i) \|_2^2. \quad (7.59)$$

单个误差项关于位姿导数已经在前面推导过了，使用李代数扰动模型即可：

$$\frac{\partial e}{\partial \delta \xi} = -(\exp(\xi^\wedge) \mathbf{p}'_i)^\odot. \quad (7.60)$$

于是，在非线性优化中只需不断迭代，我们就能找到极小值。而且，可以证明 [6]，ICP 问题存在唯一解或无穷多解的情况。在唯一解的情况下，只要我们能找到极小值解，那么这个极小值就是全局最优值——因此不会遇到局部极小而非全局最小的情况。这也意味着 ICP 求解可以任意选定初始值。这是已经匹配点时求解 ICP 的一大好处。

需要说明的是，我们这里讲的 ICP，是指已经由图像特征给定了匹配的情况下，进行位姿估计的问题。在匹配已知的情况下，这个最小二乘问题实际上具有解析解 [52, 53, 54]，所以并没有必要进行迭代优化。ICP 的研究者们往往更加关心匹配未知的情况。不过，在 RGB-D SLAM 中，由于一个像素的深度数据可能测量不到，所以我们可以混合着使用 PnP 和 ICP 优化：对于深度已知的特征点，用建模它们的 3D-3D 误差；对于深度未知的特征点，则建模 3D-2D 的重投影误差。于是，可以将所有的误差放在同一个问题中考虑，使得求解更加方便。

7.10 实践：求解 ICP

7.10.1 SVD 方法

下面，我们来演示一下如何使用 SVD 以及非线性优化来求解 ICP。本节我们使用两个 RGB-D 图像，通过特征匹配获取两组 3D 点，最后用 ICP 计算它们的位姿变换。由于 OpenCV 目前还没有计算两组带匹配点的 ICP 的方法，而且它的原理也并不复杂，所以我们自己来实现一个 ICP。

slambook/ch7/pose_estimation_3d3d.cpp（片段）

```
1 void pose_estimation_3d3d(
2     const vector<Point3f>& pts1,
3     const vector<Point3f>& pts2,
4     Mat& R, Mat& t
5 )
6 {
7     Point3f p1, p2; // center of mass
8     int N = pts1.size();
9     for (int i=0; i<N; i++)
10    {
11        p1 += pts1[i];
12        p2 += pts2[i];
13    }
14    p1 /= N; p2 /= N;
15    vector<Point3f> q1(N), q2(N); // remove the center
16    for (int i=0; i<N; i++)
17    {
18        q1[i] = pts1[i] - p1;
19        q2[i] = pts2[i] - p2;
20    }
```

```

21 // compute q1*q2^T
22 Eigen::Matrix3d W = Eigen::Matrix3d::Zero();
23 for ( int i=0; i<N; i++ )
24 {
25     W += Eigen::Vector3d( q1[i].x, q1[i].y, q1[i].z ) * Eigen::Vector3d( q2[i].x, q2[i].y, q2[i].z )
26         .transpose();
27 }
28 cout<<"W="<<W<<endl;
29
30 // SVD on W
31 Eigen::JacobiSVD<Eigen::Matrix3d> svd(W, Eigen::ComputeFullU|Eigen::ComputeFullV);
32 Eigen::Matrix3d U = svd.matrixU();
33 Eigen::Matrix3d V = svd.matrixV();
34 cout<<"U="<<U<<endl;
35 cout<<"V="<<V<<endl;
36
37 Eigen::Matrix3d R_ = U*(V.transpose());
38 Eigen::Vector3d t_ = Eigen::Vector3d( p1.x, p1.y, p1.z ) - R_ * Eigen::Vector3d( p2.x, p2.y, p2.z )
39 ;
40
41 // convert to cv::Mat
42 R = ( Mat<double>(3,3) <<
43     R_(0,0), R_(0,1), R_(0,2),
44     R_(1,0), R_(1,1), R_(1,2),
45     R_(2,0), R_(2,1), R_(2,2)
46 );
47 t = ( Mat<double>(3,1) << t_(0,0), t_(1,0), t_(2,0) );
}

```

ICP 的实现方式和前文讲述的是一致的。我们调用 Eigen 进行 SVD，然后计算 \mathbf{R}, \mathbf{t} 矩阵。我们输出了匹配后的结果，不过请注意，由于前面的推导是按照 $\mathbf{p}_i = \mathbf{R}\mathbf{p}'_i + \mathbf{t}$ 进行的，这里的 \mathbf{R}, \mathbf{t} 是第二帧到第一帧的变换，与前面 PnP 部分是相反的。所以在输出结果中，我们同时打印了逆变换：

```

1 % build/pose_estimation_3d3d 1.png 2.png 1_depth.png 2_depth.png
2 -- Max dist : 95.000000
3 -- Min dist : 4.000000
4 一共找到了 79 组匹配点
5 3d-3d pairs: 74
6 W= 298.51 -14.1815 41.0456
7 -44.8208 107.825 -164.404
8 78.1978 -163.954 271.439
9 U= 0.474143 -0.880373 -0.0114952
10 -0.460275 -0.258979 0.849163
11 0.750556 0.397334 0.528006
12 V= 0.535211 -0.844064 -0.0332488
13 -0.434767 -0.309001 0.84587
14 0.724242 0.438263 0.532352

```

```

15 ICP via SVD results:
16 R = [0.9972395976914055, 0.05617039049497474, -0.04855998381307948;
17 -0.05598344580804095, 0.9984181433274515, 0.005202390798842771;
18 0.04877538920134394, -0.002469474885032297, 0.998806719591959]
19 t = [0.7086246277241892;
20 -0.2775515782948791;
21 -0.1559573762377209]
22 R_inv = [0.9972395976914055, -0.05598344580804095, 0.04877538920134394;
23 0.05617039049497474, 0.9984181433274515, -0.002469474885032297;
24 -0.04855998381307948, 0.005202390798842771, 0.998806719591959]
25 t_inv = [-0.7145999506834847;
26 0.2369236766013986;
27 0.1916260075851286]

```

读者可以比较一下 ICP 与 PnP, 对极几何的运动估计结果之间的差异。可以认为, 在这个过程中我们使用了越来越多的信息(没有深度——有一个图的深度——有两个图的深度), 因此, 在深度准确的情况下, 得到的估计也将越来越准确。但是, 由于 Kinect 的深度图存在噪声, 而且有可能存在数据丢失的情况, 使得我们不得不丢弃一些没有深度数据的特征点。这可能导致 ICP 的估计不够准确, 并且, 如果特征点丢弃得太多, 可能引起由于特征点太少, 无法进行运动估计的情况。

7.10.2 非线性优化方法

下面我们考虑用非线性优化来计算 ICP。我们依然使用李代数来表达相机位姿。与 SVD 思路不同的地方在于, 在优化中我们不仅考虑相机的位姿, 同时会优化 3D 点的空间位置。对我们来说, RGB-D 相机每次可以观测到路标点的三维位置, 从而产生一个 3D 观测数据。不过, 由于 g2o/sba 中没有提供 3D 到 3D 的边, 而我们又想使用 g2o/sba 中李代数实现的位姿节点, 所以最好的方式是自定义一种这样的边, 并向 g2o 提供解析求导方式。

slambook/ch7/pose_estimation_3d3d.cpp

```

1 class EdgeProjectXYZRGBDPoseOnly : public g2o::BaseUnaryEdge<3, Eigen::Vector3d, g2o::VertexSE3Expmap>
2 {
3     public:
4         EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
5         EdgeProjectXYZRGBDPoseOnly( const Eigen::Vector3d& point ) :
6             _point(point) {}
7
8         virtual void computeError()
9         {
10             const g2o::VertexSE3Expmap* pose = static_cast<const g2o::VertexSE3Expmap*>( _vertices[0] );
11             // measurement is p, point is p'

```

```
12     _error = _measurement - pose->estimate().map( _point );
13 }
14
15 virtual void linearizeOplus()
16 {
17     g2o::VertexSE3Expmap* pose = static_cast<g2o::VertexSE3Expmap*>(_vertices[0]);
18     g2o::SE3Quat T(pose->estimate());
19     Eigen::Vector3d xyz_trans = T.map(_point);
20     double x = xyz_trans[0];
21     double y = xyz_trans[1];
22     double z = xyz_trans[2];
23
24     _jacobianOplusXi(0,0) = 0;
25     _jacobianOplusXi(0,1) = -z;
26     _jacobianOplusXi(0,2) = y;
27     _jacobianOplusXi(0,3) = -1;
28     _jacobianOplusXi(0,4) = 0;
29     _jacobianOplusXi(0,5) = 0;
30
31     _jacobianOplusXi(1,0) = z;
32     _jacobianOplusXi(1,1) = 0;
33     _jacobianOplusXi(1,2) = -x;
34     _jacobianOplusXi(1,3) = 0;
35     _jacobianOplusXi(1,4) = -1;
36     _jacobianOplusXi(1,5) = 0;
37
38     _jacobianOplusXi(2,0) = -y;
39     _jacobianOplusXi(2,1) = x;
40     _jacobianOplusXi(2,2) = 0;
41     _jacobianOplusXi(2,3) = 0;
42     _jacobianOplusXi(2,4) = 0;
43     _jacobianOplusXi(2,5) = -1;
44 }
45
46 bool read ( istream& in ) {}
47 bool write ( ostream& out ) const {}
48 protected:
49     Eigen::Vector3d _point;
50 };
```

这是一个一元边，写法类似于前面提到的 g2o::EdgeSE3ProjectXYZ，不过观测量从 2 维变成了 3 维，内部没有相机模型，并且只关联到一个节点。请读者注意这里雅可比矩阵的书写，它必须与我们前面的推导一致。雅可比矩阵给出了关于相机位姿的导数，是一个 3×6 的矩阵。

调用 g2o 进行优化的代码是相似的，我们设定好图优化的节点和边即可。这部分代码请读者查看源文件，我们就不在书中列出了。现在，来看看优化的结果：

```
1 calling bundle adjustment
```

```

2 | iteration= 0 chi2= 452884.696837 time= 3.8443e-05 cumTime= 3.8443e-05 edges= 74 schur= 0
3 | iteration= 1 chi2= 452762.638918 time= 1.436e-05 cumTime= 5.2803e-05 edges= 74 schur= 0
4 | iteration= 2 chi2= 452762.618632 time= 1.1943e-05
5 | ..... 中间略
6 | iteration= 9 chi2= 452762.618615 time= 1.0772e-05 cumTime= 0.000140108 edges= 74 schur= 0
7 | optimization costs time: 0.000528066 seconds.
8 |
9 | after optimization:
10 | T=
11 | 0.99724 0.0561704 -0.04856 0.708625
12 | -0.0559834 0.998418 0.00520239 -0.277551
13 | 0.0487754 -0.00246948 0.998807 -0.155957
14 | 0 0 1

```

我们发现只迭代一次后，总体误差就已经稳定不变，说明仅在一次迭代之后算法即已收敛。从位姿求解的结果可以看出，它和前面 SVD 给出的位姿结果几乎一模一样，这说明 SVD 已经给出了优化问题的解析解。所以，本实验中可以认为 SVD 给出的结果是相机位姿的最优值。

需要说明的是，在本例的 ICP 中，我们使用了在两个图都有深度读数的特征点。然而，事实上，只要其中一个图深度确定，我们就能用类似于 PnP 的误差方式，把它们也加到优化中来。同时，除了相机位姿之外，将空间点也作为优化变量考虑，亦是一种解决问题的方式。我们应当清楚，实际的求解是非常灵活的，不必拘泥于某种固定的形式。如果同时考虑点和相机，整个问题就变得更自由了，你可能会得到其他的解。比如，可以让相机少转一些角度，而把点多移动一些。这从另一侧面反映出，在 Bundle Adjustment 里面，我们会希望有尽可能多的约束，因为多次观测会带来更多的信息，使我们能够更准确地估计每个变量。

7.11 小结

本节介绍了基于特征点的视觉里程计中的几个重要的问题。包括：

1. 特征点是如何提取并匹配的；
2. 如何通过 2D-2D 的特征点估计相机运动；
3. 如何从 2D-2D 的匹配估计一个点的空间位置；
4. 3D-2D 的 PnP 问题，它的线性解法和 Bundle Adjustment 解法；
5. 3D-3D 的 ICP 问题，其线性解法和 Bundle Adjustment 解法。

本章内容较为丰富，且结合应用了前几章的基本知识。读者若觉得理解有困难，可以对前面知识稍加回顾。最好亲自做一遍实验，以理解整个运动估计的内容。

需要解释的是，为保证行文流畅，我们省略了大量的，关于某些特殊情况的讨论。例如，如果在对极几何求解过程中，给定的特征点共面，会发生什么情况（这在单应矩阵 H 中提到）？共线又会发生什么情况？在 PnP 和 ICP 中若给定这样的解，又会导致什么情况？求解算法能否识别这些特殊的情况，并报告所得的解可能不可靠？——尽管它们都是值得研究和探索的，然而对它们的讨论势必让本书变得特别繁琐。而且在工程实现中，这些情况甚少出现，所以本书介绍的方法，是指在实际工程中能够有效运行的方法，我们假定了那些少见的情况并不发生。如果你关心这些少见的情况，可以阅读 [3] 等论文，在文献中我们会经常研究一些特殊情况下的解决方案。

习题

1. 除了本书介绍的 ORB 特征点外，你还能找到哪些其他的特征点？请说说 SIFT 或 SURF 的原理，对比它们与 ORB 之间的优劣。
2. 设计程序，调用 OpenCV 中的其他种类特征点。统计在提取 1000 个特征点时，在你的机器上所用的时间。
3. * 我们发现 OpenCV 提供的 ORB 特征点，在图像当中分布不够均匀。你是否能够找到或提出让特征点分布更加均匀的方法？
4. 研究 FLANN 为何能够快速处理匹配问题。除了 FLANN 之外，还能哪些可以加速匹配的手段？
5. 把演示程序使用的 EPnP 改成其他 PnP 方法，并研究它们的工作原理。
6. 在 PnP 优化中，将第一个相机的观测也考虑进来，程序应如何书写？最后结果会有何变化？
7. 在 ICP 程序中，将空间点也作为优化变量考虑进来，程序应如何书写？最后结果会有何变化？
8. * 在特征点匹配过程中，不可避免地会遇到误匹配的情况。如果我们把错误匹配输入到 PnP 或 ICP 中，会发生怎样的情况？你能想到哪些避免误匹配的方法？
9. * 使用 Sophus 的 SE3 类，自己设计 g2o 的节点与边，实现 PnP 和 ICP 的优化。
10. * 在 Ceres 中实现 PnP 和 ICP 的优化。