

第 8 讲

视觉里程计 2

本节目标

1. 理解光流法跟踪特征点的原理。
2. 理解直接法是如何估计相机位姿的。
3. 使用 g2o 进行直接法的计算。

直接法是视觉里程计另一主要分支，它与特征点法有很大不同。虽然它还没有成为现在 VO 中的主流，但经过近几年的发展，直接法在一定程度上已经能和特征点法平分秋色。本讲，我们将介绍直接法的原理，并利用 g2o 实现直接法中的一些核心算法。

8.1 直接法的引出

上一讲我们介绍了使用特征点估计相机运动的方法。尽管特征点法在视觉里程计中占据主流地位，研究者们认识到它至少有以下几个缺点：

1. 关键点的提取与描述子的计算非常耗时。实践当中，SIFT 目前在 CPU 上是无法实时计算的，而 ORB 也需要近 20 毫秒的计算。如果整个 SLAM 以 30 毫秒/帧的速度运行，那么一大半时间都花在计算特征点上。
2. 使用特征点时，忽略了除特征点以外的所有信息。一张图像有几十万个像素，而特征点只有几百个。只使用特征点丢弃了大部分可能有用的图像信息。
3. 相机有时会运动到特征缺失的地方，往往这些地方没有明显的纹理信息。例如，有时我们会面对一堵白墙，或者一个空荡荡的走廊。这些场景下特征点数量会明显减少，我们可能找不到足够的匹配点来计算相机运动。

我们看到使用特征点确实存在一些问题。有没有什么办法能够克服这些缺点呢？我们有以下几种思路：

- 保留特征点，但只计算关键点，不计算描述子。同时，使用光流法（Optical Flow）来跟踪特征点的运动。这样可以回避计算和匹配描述子带来的时间，但光流本身的计算需要一定时间；
- 只计算关键点，不计算描述子。同时，使用直接法（Direct Method） 算特征点在下一时刻图像的位置。这同样可以跳过描述子的计算过程，而且直接法的计算更加简单。
- 既不计算关键点、也不计算描述子，而是根据像素灰度的差异，直接计算相机运动。

第一种方法仍然使用特征点，只是把匹配描述子替换成了光流跟踪，估计相机运动时仍使用对极几何、PnP 或 ICP 算法。而在后两个方法中，我们会根据图像的像素灰度信息来计算相机运动，它们都称为直接法。

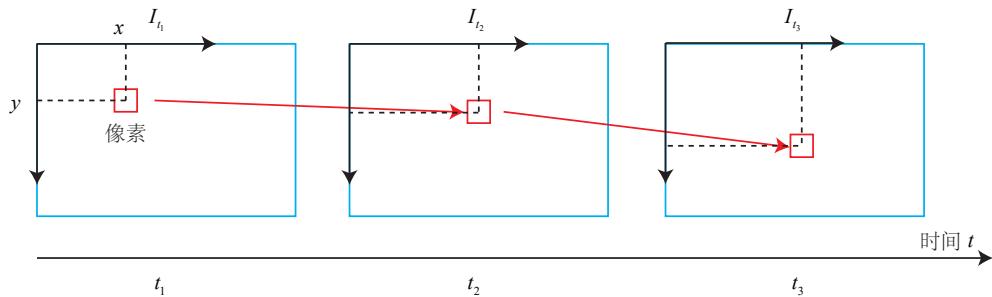
使用特征点法估计相机运动时，我们把特征点看作固定在三维空间的不动点。根据它们在相机中的投影位置，通过最小化重投影误差（Reprojection error）来优化相机运动。在这个过程中，我们需要精确地知道空间点在两个相机中投影后的像素位置——这也就是我们为何要对特征进行匹配或跟踪的理由。同时，我们也知道，计算、匹配特征需要付出大量的计算量。相对的，在直接法中，我们并不需要知道点与点之间之间的对应关系，而是通过最小化光度误差（Photometric error）来求得它们。

直接法是本讲介绍的重点。它是为了克服特征点法的上述缺点而存在的。直接法根据像素的亮度信息，估计相机的运动，可以完全不用计算关键点和描述子，于是，既避免了特征的计算时间，也避免了特征缺失的情况。只要场景中存在明暗变化（可以是渐变，不形成局部的图像梯度），直接法就能工作。根据使用像素的数量，直接法分为稀疏、稠密和半稠密三种。相比于特征点法只能重构稀疏特征点（稀疏地图），直接法还具有恢复稠密或半稠密结构的能力。

历史上，虽然早期也有一些对直接法的使用 [55]，但直到 RGB-D 相机出现后，人们才发现直接法对 RGB-D 相机，进而对于单目相机，都是行之有效的方法。随着一些使用直接法的开源项目的出现（如 SVO[56]、LSD-SLAM[57] 等），它们逐渐地走上主流舞台，成为视觉里程计算法中重要的一部分。

8.2 光流 (Optical Flow)

直接法是从光流演变而来的。它们非常相似，具有相同的假设条件。光流描述了像素在图像中的运动，而直接法则附带着一个相机运动模型。为了说明直接法，我们先来介绍一下光流。



$$\text{灰度不变假设: } I(x_1, y_1, t_1) = I(x_2, y_2, t_2) = I(x_3, y_3, t_3)$$

图 8-1 LK 光流法示意图。

光流是一种描述像素随着时间，在图像之间运动的方法，如图 8-1 所示。随着时间的经过，同一个像素会在图像中运动，而我们希望追踪它的运动过程。计算部分像素运动的称为稀疏光流，计算所有像素的称为稠密光流。稀疏光流以 Lucas-Kanade 光流为代表，并可以在 SLAM 中用于跟踪特征点位置。因此，本节主要介绍 Lucas-Kanade 光流，亦称 LK 光流。

8.2.1 Lucas-Kanade 光流

在 LK 光流中，我们认为来自相机的图像是随时间变化的。图像可以看作时间的函数： $\mathbf{I}(t)$ 。那么，一个在 t 时刻，位于 (x, y) 处的像素，它的灰度可以写成

$$\mathbf{I}(x, y, t).$$

这种方式把图像看成了关于位置与时间的函数，它的值域就是图像中像素的灰度。现在考虑某个固定的空间点，它在 t 时刻的像素坐标为 x, y 。由于相机的运动，它的图像坐标将发生变化。我们希望估计这个空间点在其他时刻里图像的位置。怎么估计呢？这里要引入光流法的基本假设：

灰度不变假设：同一个空间点的像素灰度值，在各个图像中是固定不变的。

对于 t 时刻位于 (x, y) 处的像素，我们设 $t + dt$ 时刻，它运动到 $(x + dx, y + dy)$ 处。由于灰度不变，我们有：

$$\mathbf{I}(x + dx, y + dy, t + dt) = \mathbf{I}(x, y, t). \quad (8.1)$$

灰度不变假设是一个很强的假设，实际当中很可能不成立。事实上，由于物体的材质不同，像素会出现高光和阴影部分；有时，相机会自动调整曝光参数，使得图像整体变亮或变暗。这些时候灰度不变假设都是不成立的，因此光流的结果也不一定可靠。然而，从另一方面来说，所有算法都是在一定假设下工作的。如果我们什么假设都不做，就没法设计实用的算法。所以，暂且让我们认为该假设成立，看看如何计算像素的运动。

对左边进行泰勒展开，保留一阶项，得：

$$\mathbf{I}(x + dx, y + dy, t + dt) \approx \mathbf{I}(x, y, t) + \frac{\partial \mathbf{I}}{\partial x} dx + \frac{\partial \mathbf{I}}{\partial y} dy + \frac{\partial \mathbf{I}}{\partial t} dt. \quad (8.2)$$

因为我们假设了灰度不变，于是下一个时刻的灰度等于之前的灰度，从而

$$\frac{\partial \mathbf{I}}{\partial x} dx + \frac{\partial \mathbf{I}}{\partial y} dy + \frac{\partial \mathbf{I}}{\partial t} dt = 0. \quad (8.3)$$

两边除以 dt ，得：

$$\frac{\partial \mathbf{I}}{\partial x} \frac{dx}{dt} + \frac{\partial \mathbf{I}}{\partial y} \frac{dy}{dt} = -\frac{\partial \mathbf{I}}{\partial t}. \quad (8.4)$$

其中 dx/dt 为像素在 x 轴上运动速度，而 dy/dt 为 y 轴速度，把它们记为 u, v 。同时 $\partial \mathbf{I} / \partial x$ 为图像在该点处 x 方向的梯度，另一项则是在 y 方向的梯度，记为 $\mathbf{I}_x, \mathbf{I}_y$ 。把

图像灰度对时间的变化量记为 \mathbf{I}_t , 写成矩阵形式, 有:

$$\begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = -\mathbf{I}_t. \quad (8.5)$$

我们想计算的是像素的运动 u, v , 但是该式是带有两个变量的一次方程, 仅凭它无法计算出 u, v 。因此, 必须引入额外的约束来计算 u, v 。在 LK 光流中, 我们假设某一个窗口内的像素具有相同的运动。

考虑一个大小为 $w \times w$ 大小的窗口, 它含有 w^2 数量的像素。由于该窗口内像素具有同样的运动, 因此我们共有 w^2 个方程:

$$\begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix}_k \begin{bmatrix} u \\ v \end{bmatrix} = -\mathbf{I}_{tk}, \quad k = 1, \dots, w^2. \quad (8.6)$$

记:

$$\mathbf{A} = \begin{bmatrix} [\mathbf{I}_x, \mathbf{I}_y]_1 \\ \vdots \\ [\mathbf{I}_x, \mathbf{I}_y]_k \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \mathbf{I}_{t1} \\ \vdots \\ \mathbf{I}_{tk} \end{bmatrix}. \quad (8.7)$$

于是整个方程为:

$$\mathbf{A} \begin{bmatrix} u \\ v \end{bmatrix} = -\mathbf{b}. \quad (8.8)$$

这是一个关于 u, v 的超定线性方程, 传统解法是求最小二乘解。最小二乘在很多时候都用到过:

$$\begin{bmatrix} u \\ v \end{bmatrix}^* = -(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}. \quad (8.9)$$

这样就得到了像素在图像间的运动速度 u, v 。当 t 取离散的时刻而不是连续时间时, 我们可以估计某块像素在若干个图像中出现的位置。由于像素梯度仅在局部有效, 所以如果一次迭代不够好的话, 我们会多迭代几次这个方程。在 SLAM 中, LK 光流常被用来跟踪角点的运动, 我们不妨通过程序体会一下。

8.3 实践：LK 光流

8.3.1 使用 TUM 公开数据集

下面，我们来演示如何用 OpenCV 提供的光流法来跟踪特征点。与上一节一样，我们准备了若干张数据集图像，存放在程序目录中的 data/文件夹下。它们来自于慕尼黑工业大学（TUM）提供的公开 RGB-D 数据集^①。以后我们就称之为 TUM 数据集。它含有许多个 RGB-D 视频，可以作为 RGB-D 或单目 SLAM 的实验数据。它还提供了用运动捕捉系统测量的精确轨迹，可以作为标准轨迹以校准 SLAM 系统。由于该数据集比较大，我们没有放到 github 上（否则下载代码的读者要等待很长时间），请读者去数据集主页找到对应的数据。本程序中使用了一部分“freiburg1_desk”数据集中的图像。读者可以在 TUM 数据集主页找到它的下载链接。或者，也可以直接使用本书在 github 上提供的部分。

我们的数据位于本章目录的 data/ 下，以压缩包形式提供 (data.tar.gz)。由于 TUM 数据集是从实际环境中采集的，需要解释一下它的数据格式（数据集一般都有自己定义的格式）。在解压后，你将看到以下这些文件：

1. rgb.txt 和 depth.txt 记录了各文件的采集时间和对应的文件名。
2. rgb/ 和 depth/ 目录存放着采集到的 png 格式图像文件。彩色图像为八位三通道，深度图为 16 位单通道图像。文件名即采集时间。
3. groundtruth.txt 为外部运动捕捉系统采集到的相机位姿，格式为

$$(time, t_x, t_y, t_z, q_x, q_y, q_z, q_w),$$

我们可以把它看成标准轨迹。

请注意彩色图、深度图和标准轨迹的采集都是独立的，轨迹的采集频率比图像高很多。在使用数据之前，需要根据采集时间，对数据进行一次时间上的对齐，以便对彩色图和深度图进行配对。原则上，我们可以把采集时间相近于一个阈值的数据，看成是一对图像。并把相近时间的位姿，看作是该图像的真实采集位置。TUM 提供了一个 python 脚本“associate.py”（或使用 slambook/tools/associate.py）帮我们完成这件事。请把此文件放到数据集目录下，运行：

```
1 python associate.py rgb.txt depth.txt > associate.txt
```

^①<http://vision.in.tum.de/data/datasets/rgbd-dataset/download>

这段脚本会根据输入两个文件中的采集时间进行配对，最后输出到一个文件 associate.txt。输出文件含有被配对的两个图像的时间、文件名信息，可以作为后续处理的来源。此外，TUM 数据集还提供了比较估计轨迹与标准轨迹的工具，我们将在用到的地方再进行介绍。

8.3.2 使用 LK 光流

下面我们来编写程序使用 OpenCV 中的 LK 光流。使用 LK 的目的是跟踪特征点。我们对第一张图像提取 FAST 角点，然后用 LK 光流跟踪它们，并画在图中。

slambook/ch8/useLK/useLK.cpp

```
1 #include <iostream>
2 #include <fstream>
3 #include <list>
4 #include <vector>
5 #include <chrono>
6 using namespace std;
7
8 #include <opencv2/core/core.hpp>
9 #include <opencv2/highgui/highgui.hpp>
10 #include <opencv2/features2d/features2d.hpp>
11 #include <opencv2/video/tracking.hpp>
12
13 int main( int argc, char** argv )
14 {
15     if ( argc != 2 )
16     {
17         cout<<"usage: useLK path_to_dataset"<<endl;
18         return 1;
19     }
20     string path_to_dataset = argv[1];
21     string associate_file = path_to_dataset + "/associate.txt";
22     ifstream fin( associate_file );
23     string rgb_file, depth_file, time_rgb, time_depth;
24     list< cv::Point2f > keypoints; // 因为要删除跟踪失败的点，使用list
25     cv::Mat color, depth, last_color;
26     for ( int index=0; index<100; index++ )
27     {
28         fin>>time_rgb>>rgb_file>>time_depth>>depth_file;
29         color = cv::imread( path_to_dataset+"/"+rgb_file );
30         depth = cv::imread( path_to_dataset+"/"+depth_file, -1 );
31         if (index ==0 )
32         {
33             // 对第一帧提取 FAST 特征点
34             vector<cv::KeyPoint> kps;
```

```
35         cv::Ptr<cv::FastFeatureDetector> detector = cv::FastFeatureDetector::create();
36         detector->detect( color, kps );
37         for ( auto kp:kps )
38             keypoints.push_back( kp.pt );
39         last_color = color;
40         continue;
41     }
42     if ( color.data==nullptr || depth.data==nullptr )
43         continue;
44     // 对其他帧用 LK 跟踪特征点
45     vector<cv::Point2f> next_keypoints;
46     vector<cv::Point2f> prev_keypoints;
47     for ( auto kp:keypoints )
48         prev_keypoints.push_back(kp);
49     vector<unsigned char> status;
50     vector<float> error;
51     chrono::steady_clock::time_point t1 = chrono::steady_clock::now();
52     cv::calcOpticalFlowPyrLK( last_color, color, prev_keypoints, next_keypoints, status, error );
53     chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
54     chrono::duration<double> time_used = chrono::duration_cast<chrono::duration<double>>( t2-t1 );
55     cout<<"LK Flow use time: "<<time_used.count()<<" seconds."<<endl;
56     // 把跟丢的点删掉
57     int i=0;
58     for ( auto iter=keypoints.begin(); iter!=keypoints.end(); i++ )
59     {
60         if ( status[i] == 0 )
61         {
62             iter = keypoints.erase(iter);
63             continue;
64         }
65         *iter = next_keypoints[i];
66         iter++;
67     }
68     cout<<"tracked keypoints: "<<keypoints.size()<<endl;
69     if (keypoints.size() == 0)
70     {
71         cout<<"all keypoints are lost."<<endl;
72         break;
73     }
74     // 画出 keypoints
75     cv::Mat img_show = color.clone();
76     for ( auto kp:keypoints )
77         cv::circle(img_show, kp, 10, cv::Scalar(0, 240, 0), 1);
78     cv::imshow("corners", img_show);
79     cv::waitKey(0);
80     last_color = color;
81 }
82 return 0;
83 }
```

读者应当已经熟悉了 OpenCV 的使用方式，我们就不贴上 CMakeLists.txt 的写法了。该程序的运行参数里需要指定数据集所在的目录，例如：

```
1 ./build/useLK ../data
```

我们会在每次循环后暂停程序，按任意键可以继续运行。你会看到图像中大部分特征点能够顺利跟踪到，但也有特征点会丢失。丢失的特征点或是被移出了视野外，或是被其他物体挡住了。如果我们不提取新的特征点，那么光流的跟踪会越来越少：

```
1 % build/useLK ../data
2 LK Flow use time: 0.0329535 seconds.
3 tracked keypoints: 1749
4 LK Flow use time: 0.0247758 seconds.
5 tracked keypoints: 1742
6 LK Flow use time: 0.0226143 seconds.
7 tracked keypoints: 1703
8 LK Flow use time: 0.0238692 seconds.
9 tracked keypoints: 1676
10 LK Flow use time: 0.0210466 seconds.
11 tracked keypoints: 1664
12 LK Flow use time: 0.0226533 seconds.
13 tracked keypoints: 1656
14 LK Flow use time: 0.0266527 seconds.
15 tracked keypoints: 1641
16 LK Flow use time: 0.0214207 seconds.
17 tracked keypoints: 1634
```

图 8-2 显示了程序运行过程中若干帧的情况（这里使用了完整的数据集，但本书的 git 上只给出了十张图）。最初我们大约有 1700 个特征点。跟踪过程中一部分特征点会丢失，直到 100 帧时我们还有约 178 个特征点，相机视角相对于最初的图像也发生了较大改变。仔细观察特征点的跟踪过程，我们会发现位于物体角点处的特征更加稳定。边缘处的特征会沿着边缘“滑动”，这主要是因为沿着边缘移动时特征块的内容基本不变，因此程序容易认为是同一个地方。而既不在角点，也不在边缘的特征点则会频繁跳动，位置非常不稳定。这个现象很像围棋中的“金角银边草肚皮”：角点具有更好的辨识度，边缘次之，区块最少。

另一方面，读者可以看到光流法的运行时间。在跟踪 1500 个特征点时，LK 光流法大约需要 20 毫秒左右。如果减小特征点的数量，则会明显减少计算时间。我们看到，LK 光流跟踪法避免了描述子的计算与匹配，但本身也需要一定的计算量。在我们的计算平台上，使用 LK 光流能够节省一定的计算量，但在具体 SLAM 系统中使用光流还是匹配描述子，最好是亲自做实验测试一下。

另外，LK 光流跟踪能够直接得到特征点的对应关系。这个对应关系就像是描述子的匹配，但实际上我们大多数时候只会碰到特征点跟丢的情况，而不太会遇到误匹配，这应

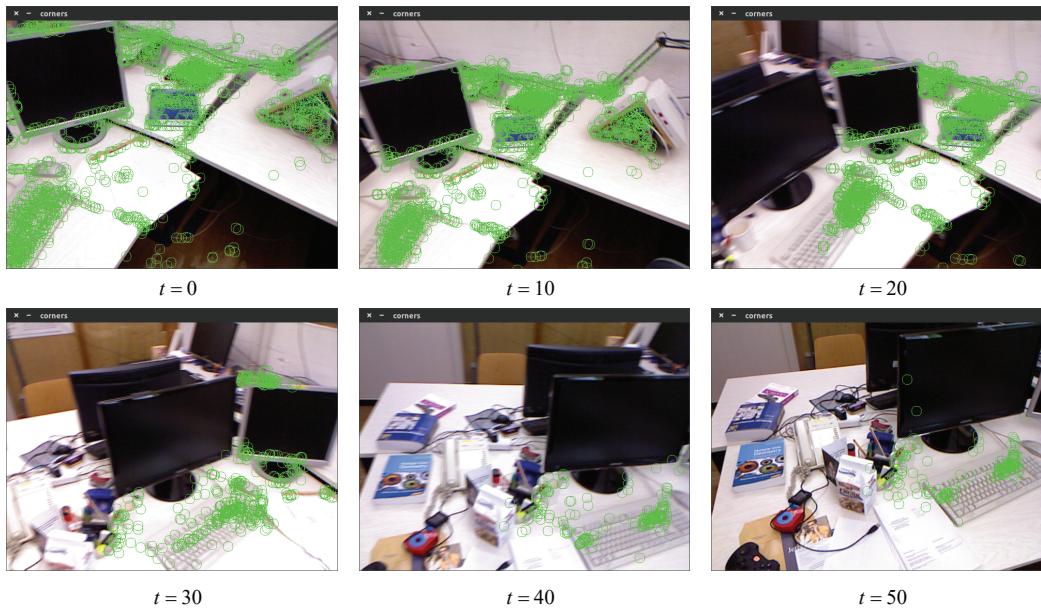


图 8-2 LK 光流法实验。

该是光流相对于描述子的一点优势。但是，匹配描述子的方法在相机运动较大时仍能成功，而光流必须要求相机运动是微小的。从这方面来说，光流的鲁棒性比描述子差一些。

最后，我们可以通过光流跟踪的特征点，用 PnP、ICP 或对极几何来估计相机运动，这些方法在上一章中都讲过，我们不再讨论。总而言之，光流法可以加速基于特征点的视觉里程计算法，避免计算和匹配描述子的过程，但要求相机运动较慢（或采集频率较高）。

8.4 直接法 (Direct Methods)

接下来，我们来讨论与光流有一定相似性的直接法。与前面章节相似，我们先介绍直接法的原理，然后使用 g2o 实现直接法。

8.4.1 直接法的推导

如图8-3所示，考虑某个空间点 P 和两个时刻的相机。 P 的世界坐标为 $[X, Y, Z]$ ，它在两个相机上成像，记非齐次像素坐标为 $\mathbf{p}_1, \mathbf{p}_2$ 。我们的目标是求第一个相机到第二个相机的相对位姿变换。我们以第一个相机为参照系，设第二个相机旋转和平移为 \mathbf{R}, \mathbf{t} （对应

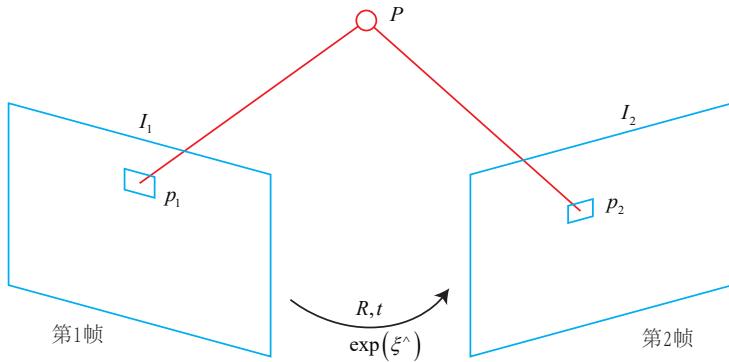


图 8-3 直接法示意图。

李代数为 ξ)。同时, 两相机的内参相同, 记为 \mathbf{K} 。为清楚起见, 我们列写完整的投影方程:

$$\begin{aligned}\mathbf{p}_1 &= \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_1 = \frac{1}{Z_1} \mathbf{K} \mathbf{P}, \\ \mathbf{p}_2 &= \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_2 = \frac{1}{Z_2} \mathbf{K} (\mathbf{R} \mathbf{P} + \mathbf{t}) = \frac{1}{Z_2} \mathbf{K} (\exp(\xi^{\wedge}) \mathbf{P})_{1:3}.\end{aligned}$$

其中 Z_1 是 P 的深度, Z_2 是 P 在第二个相机坐标系下的深度, 也就是 $\mathbf{R} \mathbf{P} + \mathbf{t}$ 的第三个坐标值。由于 $\exp(\xi^{\wedge})$ 只能和齐次坐标相乘, 所以我们乘完之后要取出前三个元素。这和上一讲以及相机模型部分的内容是一致的。

回忆特征点法中, 由于我们通过匹配描述子, 知道了 $\mathbf{p}_1, \mathbf{p}_2$ 的像素位置, 所以可以计算重投影的位置。但在直接法中, 由于没有特征匹配, 我们无从知道哪一个 \mathbf{p}_2 与 \mathbf{p}_1 对应着同一个点。直接法的思路是根据当前相机的位姿估计值, 来寻找 \mathbf{p}_2 的位置。但若相机位姿不够好, \mathbf{p}_2 的外观和 \mathbf{p}_1 会有明显差别。于是, 为了减小这个差别, 我们优化相机的位姿, 来寻找与 \mathbf{p}_1 更相似的 \mathbf{p}_2 。这同样可以通过解一个优化问题, 但此时最小化的不是重投影误差, 而是光度误差 (Photometric Error), 也就是 P 的两个像的亮度误差:

$$e = \mathbf{I}_1(\mathbf{p}_1) - \mathbf{I}_2(\mathbf{p}_2). \quad (8.10)$$

注意这里 e 是一个标量，所以没有加粗。同样的，优化目标为该误差的二范数，暂时取不加权的形式，为：

$$\min_{\xi} J(\xi) = \|e\|^2. \quad (8.11)$$

能够做这种优化的理由，仍是基于灰度不变假设。在直接法中，我们假设一个空间点在各个视角下，成像的灰度是不变的。我们有许多个（比如 N 个）空间点 P_i ，那么，整个相机位姿估计问题变为：

$$\min_{\xi} J(\xi) = \sum_{i=1}^N e_i^T e_i, \quad e_i = \mathbf{I}_1(\mathbf{p}_{1,i}) - \mathbf{I}_2(\mathbf{p}_{2,i}). \quad (8.12)$$

注意这里的优化变量是相机位姿 ξ 。为了求解这个优化问题，我们关心误差 e 是如何随着相机位姿 ξ 变化的，需要分析它们的导数关系。因此，使用李代数上的扰动模型。我们给 $\exp(\xi)$ 左乘一个小扰动 $\exp(\delta\xi)$ ，得：^①

$$\begin{aligned} e(\xi \oplus \delta\xi) &= \mathbf{I}_1\left(\frac{1}{Z_1}\mathbf{K}\mathbf{P}\right) - \mathbf{I}_2\left(\frac{1}{Z_2}\mathbf{K}\exp(\delta\xi^\wedge)\exp(\xi^\wedge)\mathbf{P}\right) \\ &\approx \mathbf{I}_1\left(\frac{1}{Z_1}\mathbf{K}\mathbf{P}\right) - \mathbf{I}_2\left(\frac{1}{Z_2}\mathbf{K}(1 + \delta\xi^\wedge)\exp(\xi^\wedge)\mathbf{P}\right) \\ &= \mathbf{I}_1\left(\frac{1}{Z_1}\mathbf{K}\mathbf{P}\right) - \mathbf{I}_2\left(\frac{1}{Z_2}\mathbf{K}\exp(\xi^\wedge)\mathbf{P} + \frac{1}{Z_2}\mathbf{K}\delta\xi^\wedge\exp(\xi^\wedge)\mathbf{P}\right). \end{aligned}$$

类似于上一章，记

$$\begin{aligned} \mathbf{q} &= \delta\xi^\wedge\exp(\xi^\wedge)\mathbf{P}, \\ \mathbf{u} &= \frac{1}{Z_2}\mathbf{K}\mathbf{q}. \end{aligned}$$

这里的 \mathbf{q} 为 \mathbf{P} 在扰动之后，位于第二个相机坐标系下的坐标，而 \mathbf{u} 为它的像素坐标。

^①为了避免齐次/非齐次坐标转换而导致的公式形式复杂化，我们假设中间隐式地做了所需的变化。它不会影响公式的推导。

利用一阶泰勒展开，有：

$$\begin{aligned} e(\boldsymbol{\xi} \oplus \delta\boldsymbol{\xi}) &= \mathbf{I}_1 \left(\frac{1}{Z_1} \mathbf{K} \mathbf{P} \right) - \mathbf{I}_2 \left(\frac{1}{Z_2} \mathbf{K} \exp(\boldsymbol{\xi}^\wedge) \mathbf{P} + \mathbf{u} \right) \\ &\approx \mathbf{I}_1 \left(\frac{1}{Z_1} \mathbf{K} \mathbf{P} \right) - \mathbf{I}_2 \left(\frac{1}{Z_2} \mathbf{K} \exp(\boldsymbol{\xi}^\wedge) \mathbf{P} \right) - \frac{\partial \mathbf{I}_2}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \delta\boldsymbol{\xi}} \delta\boldsymbol{\xi} \\ &= e(\boldsymbol{\xi}) - \frac{\partial \mathbf{I}_2}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \delta\boldsymbol{\xi}} \delta\boldsymbol{\xi}. \end{aligned}$$

我们看到，一阶导数由于链式法则分成了三项，而这三项都是容易计算的：

1. $\partial \mathbf{I}_2 / \partial \mathbf{u}$ 为 \mathbf{u} 处的像素梯度；
2. $\partial \mathbf{u} / \partial \mathbf{q}$ 为投影方程关于相机坐标系下的三维点的导数。记 $\mathbf{q} = [X, Y, Z]^T$ ，根据上一节的推导，导数为：

$$\frac{\partial \mathbf{u}}{\partial \mathbf{q}} = \begin{bmatrix} \frac{\partial u}{\partial X} & \frac{\partial u}{\partial Y} & \frac{\partial u}{\partial Z} \\ \frac{\partial v}{\partial X} & \frac{\partial v}{\partial Y} & \frac{\partial v}{\partial Z} \end{bmatrix} = \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} \end{bmatrix}. \quad (8.13)$$

3. $\partial \mathbf{q} / \partial \delta\boldsymbol{\xi}$ 为变换后的三维点对变换的导数，这在李代数章节已经介绍过了：

$$\frac{\partial \mathbf{q}}{\partial \delta\boldsymbol{\xi}} = [\mathbf{I}, -\mathbf{q}^\wedge]. \quad (8.14)$$

在实践中，由于后两项只与三维点 \mathbf{q} 有关，而与图像无关，我们经常把它合并在一起：

$$\frac{\partial \mathbf{u}}{\partial \delta\boldsymbol{\xi}} = \begin{bmatrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} & -\frac{f_x X Y}{Z^2} & f_x + \frac{f_x X^2}{Z^2} & -\frac{f_x Y}{Z} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} & -f_y - \frac{f_y Y^2}{Z^2} & \frac{f_y X Y}{Z^2} & \frac{f_y X}{Z} \end{bmatrix}. \quad (8.15)$$

这个 2×6 的矩阵在上一讲中也出现过。于是，我们推导了误差相对于李代数的雅可比矩阵：

$$\mathbf{J} = -\frac{\partial \mathbf{I}_2}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \delta\boldsymbol{\xi}}. \quad (8.16)$$

对于 N 个点的问题，我们可以用这种方法计算优化问题的雅可比，然后使用 G-N 或 L-M 计算增量，迭代求解。至此，我们推导了直接法估计相机位姿的整个流程，下面我们通过程序来演示一下直接法是如何使用的。

8.4.2 直接法的讨论

在我们上面的推导中， P 是一个已知位置的空间点，它是怎么来的呢？在 RGB-D 相机下，我们可以把任意像素反投影到三维空间，然后投影到下一个图像中。如果在单目相机中，这件事情要更为困难，因为我们还需考虑由 P 的深度带来的不确定性。详细的深度估计放到 13 讲中讨论。现在我们先来考虑简单的情况，即 P 深度已知的情况。

根据 P 的来源，我们可以把直接法进行分类：

1. P 来自于稀疏关键点，我们称之为稀疏直接法。通常我们使用数百个至上千个关键点，并且像 L-K 光流那样，假设它周围像素也是不变的。这种稀疏直接法不必计算描述子，并且只使用数百个像素，因此速度最快，但只能计算稀疏的重构。
2. P 来自部分像素。我们看到式 (8.16) 中，如果像素梯度为零，整一项雅可比就为零，不会对计算运动增量有任何贡献。因此，可以考虑只使用带有梯度的像素点，舍弃像素梯度不明显的地方。这称之为半稠密 (Semi-Dense) 的直接法，可以重构一个半稠密结构。
3. P 为所有像素，称为稠密直接法。稠密重构需要计算所有像素（一般几十万至几百万个），因此多数不能在现有的 CPU 上实时计算，需要 GPU 的加速。但是，如前面所讨论的，梯度不明显的点，在运动估计中不会有太大贡献，在重构时也会难以估计位置。

可以看到，从稀疏到稠密重构，都可以用直接法来计算。它们的计算量是逐渐增长的。稀疏方法可以快速地求解相机位姿，而稠密方法可以建立完整地图。具体使用哪种方法，需要视机器人的应用环境而定。特别地，在低端的计算平台上，稀疏直接法可以做到非常快速的效果，适用于实时性较高且计算资源有限的场合 [58]。

8.5 實踐：RGB-D 的直接法

现在，我们来演示如何使用稀疏的直接法。由于本书不涉及 GPU 编程，稠密的直接法就省略掉了。同时，为了保持程序简单，我们使用 RGB-D 数据而非单目数据，这样可以省略掉单目的深度恢复部分。基于特征点的深度恢复已经在上一讲介绍过了，而基于块匹配的深度恢复将在后面章节中介绍。所以本节我们来考虑 RGB-D 上的稀疏直接法 VO。

由于求解直接法最后等价于求解一个优化问题，因此我们可以使用 g2o 或 Ceres 这些优化库帮助我们求解。本节以 g2o 为例设计实验，而 Ceres 部分则留作习题。在使用 g2o 之前，需要把直接法抽象成一个图优化问题。显然，直接法是由以下顶点和边组成的：

1. 优化变量为一个相机位姿，因此需要一个位姿顶点。由于我们在推导中使用了李代数，故程序中使用李代数表达的 $SE(3)$ 位姿顶点。与上一章一样，我们将使用“VertexSE3Expmap”作为相机位姿。
2. 误差项为单个像素的光度误差。由于整个优化过程中 $I_1(\mathbf{p}_1)$ 保持不变，我们可以把它当成一个固定的预设值，然后调整相机位姿，使 $I_2(\mathbf{p}_2)$ 接近这个值。于是，这种边只连接一个顶点，为一元边。由于 g2o 中本身没有计算光度误差的边，我们需要自己定义一种新的边。

在上述的建模中，直接法图优化问题是由一个相机位姿顶点与许多条一元边组成的。如果使用稀疏的直接法，那我们大约会有几百至几千条这样的边；稠密直接法则会有几十万条边。优化问题对应的线性方程是计算李代数增量，本身规模不大 (6×6)，所以主要的计算时间会花费在每条边的误差与雅可比的计算上。下面的实验中，我们先来定义一种用于直接法位姿估计的边，然后，使用该边构建图优化问题并求解之。实验工程位于“slambook/ch8/directMethod”中。

8.5.2 定义直接法的边

首先我们来定义计算光度误差的边。按照前面的推导，还需要给出它的雅可比矩阵：

slambook/ch8/directMethod/direct_sparse.cpp (片段)

```

1 // project a 3d point into an image plane, the error is photometric error
2 // an unary edge with one vertex SE3Expmap (the pose of camera)
3 class EdgeSE3ProjectDirect: public BaseUnaryEdge< 1, double, VertexSE3Expmap>
4 {
5 public:
6     EIGEN_MAKE_ALIGNED_OPERATOR_NEW
7
8     EdgeSE3ProjectDirect() {}
9
10    EdgeSE3ProjectDirect ( Eigen::Vector3d point, float fx, float fy, float cx, float cy, cv::Mat* image
11        ) : x_world_ ( point ), fx_ ( fx ), fy_ ( fy ), cx_ ( cx ), cy_ ( cy ), image_ ( image )
12    {}
13
14    virtual void computeError()
15    {
16        const VertexSE3Expmap* v = static_cast<const VertexSE3Expmap*> ( _vertices[0] );
17        Eigen::Vector3d x_local = v->estimate().map ( x_world_ );
18        float x = x_local[0]*fx_/x_local[2] + cx_;
19        float y = x_local[1]*fy_/x_local[2] + cy_;
20        // check x,y is in the image
21        if ( x<0 || ( x+4 ) >image_->cols || ( y-4 ) <0 || ( y+4 ) >image_->rows )

```

```
21  {
22      _error ( 0,0 ) = 0.0;
23      this->setLevel ( 1 );
24  }
25  else
26  {
27      _error ( 0,0 ) = getPixelValue ( x,y ) - _measurement;
28  }
29 }
30
31 // plus in manifold
32 virtual void linearizeOplus( )
33 {
34     if ( level() == 1 )
35     {
36         _jacobianOplusXi = Eigen::Matrix<double, 1, 6>::Zero();
37         return;
38     }
39     VertexSE3Expmap* vtx = static_cast<VertexSE3Expmap*> ( _vertices[0] );
40     Eigen::Vector3d xyz_trans = vtx->estimate().map ( x_world_ ); // q in book
41
42     double x = xyz_trans[0];
43     double y = xyz_trans[1];
44     double invz = 1.0/xyz_trans[2];
45     double invz_2 = invz*invz;
46
47     float u = x*fx_*invz + cx_;
48     float v = y*fy_*invz + cy_;
49
50     // jacobian from se3 to u,v
51     // NOTE that in g2o the Lie algebra is (\omega, \epsilon), where \omega is so(3) and \epsilon is the
52     // translation
53     Eigen::Matrix<double, 2, 6> jacobian_uv_ksai;
54
55     jacobian_uv_ksai ( 0,0 ) = - x*y*invz_2 *fx_;
56     jacobian_uv_ksai ( 0,1 ) = ( 1+ ( x*x*invz_2 ) ) *fx_;
57     jacobian_uv_ksai ( 0,2 ) = - y*invz *fx_;
58     jacobian_uv_ksai ( 0,3 ) = invz *fx_;
59     jacobian_uv_ksai ( 0,4 ) = 0;
60     jacobian_uv_ksai ( 0,5 ) = -x*invz_2 *fx_;
61
62     jacobian_uv_ksai ( 1,0 ) = - ( 1+y*y*invz_2 ) *fy_;
63     jacobian_uv_ksai ( 1,1 ) = x*y*invz_2 *fy_;
64     jacobian_uv_ksai ( 1,2 ) = x*invz *fy_;
65     jacobian_uv_ksai ( 1,3 ) = 0;
66     jacobian_uv_ksai ( 1,4 ) = invz *fy_;
67     jacobian_uv_ksai ( 1,5 ) = -y*invz_2 *fy_;
68
69     Eigen::Matrix<double, 1, 2> jacobian_pixel_uv;
```

```

70     jacobian_pixel_uv ( 0,0 ) = ( getPixelValue ( u+1,v )-getPixelValue ( u-1,v ) ) /2;
71     jacobian_pixel_uv ( 0,1 ) = ( getPixelValue ( u,v+1 )-getPixelValue ( u,v-1 ) ) /2;
72
73     _jacobianOplusXi = jacobian_pixel_uv*jacobian_uv_ksai;
74 }
75
76 // dummy read and write functions because we don't care...
77 virtual bool read ( std::istream& in ) {}
78 virtual bool write ( std::ostream& out ) const {}
79
80 protected:
81     // get a gray scale value from reference image (bilinear interpolated)
82     inline float getPixelValue ( float x, float y )
83     {
84         uchar* data = & image_->data[ int ( y ) * image_->step + int ( x ) ];
85         float xx = x - floor ( x );
86         float yy = y - floor ( y );
87         return float (
88             ( 1-xx ) * ( 1-yy ) * data[0] +
89             xx* ( 1-yy ) * data[1] +
90             ( 1-xx ) *yy*data[ image_->step ] +
91             xx*yy*data[ image_->step+1 ]
92         );
93     }
94 public:
95     Eigen::Vector3d x_world_; // 3D point in world frame
96     float cx_=0, cy_=0, fx_=0, fy_=0; // Camera intrinsics
97     cv::Mat* image_=nullptr; // reference image
98 };

```

我们的边继承自 g2o::BaseUnaryEdge。在继承时，需要在模板参数里填入测量值的维度、类型，以及连接此边的顶点，同时，我们把空间点 P 、相机内参和图像存储在该边的成员变量中。为了让 g2o 优化该边对应的误差，我们需要覆写两个虚函数：用 computeError() 计算误差值，用 linearizeOplus() 计算雅可比。可以看到，这里的雅可比计算与式 (8.16) 是一致的。注意我们在程序中的误差计算里，使用了 $I_2(p_2) - I_1(p_1)$ 的形式，因此前面的负号可以省去，只需把像素梯度乘以像素到李代数的梯度即可。

在程序中，相机位姿是用浮点数表示的，投影到像素坐标也是浮点形式。为了更精细地计算像素亮度，我们要对图像进行插值。我们这里采用了简单的双线性插值，也可以使用更复杂的插值方式，但计算代价可能会变高一些。

8.5.3 使用直接法估计相机运动

定义了 g2o 边后，我们将节点和边组合成图，就可以调用 g2o 进行优化了。实现代码位于 slambook/ch8/directMethod/direct_sparse.cpp 中，请读者阅读该部分代码并编译

它。

在这个实验中，我们读取数据集的 RGB-D 图像序列。以第一个图像为参考帧，然后用直接法求解后续图像的位姿。在参考帧中，对第一张图像提取 FAST 关键点（不需要描述子），并使用直接法估计这些关键点在第二个图像中的位置，以及第二个图像的相机位姿。这就构成了一种简单的稀疏直接法。最后，我们画出这些关键点在第二个图像中的投影。

运行：

```
1 build/direct_sparse ~/dataset/rgbd_dataset_freiburg1_desk
```

程序会在作图之后暂停，你可以看到特征点的位置关系，终端也会输出迭代误差的下降过程。



图 8-4 稀疏直接法的实验。左：误差随着迭代下降。右：参考帧与后 1 至 9 帧对比（选取部分关键点）。

如图 8-4 所示，我们看到在两个图像相差不多的时候，直接法会调整相机的位姿，使得大部分像素都能够正确跟踪。但是，在稍长一点的时间内，比如说 0-9 帧之间的对比，我们发现由于相机位姿估计不准确，特征点出现了明显的偏移现象。我们会在本讲末尾对它进行分析。

8.5.4 半稠密直接法

我们很容易就能把程序拓展成半稠密的直接法形式。对参考帧中，先提取梯度较明显的像素，然后用直接法，以这些像素为图优化边，来估计相机运动。对先前的程序做如下

的修改：

slambook/ch8/direct_semidense.cpp

```

1 // select the pixels with high gradients
2 for ( int x=10; x<gray.cols-10; x++ )
3     for ( int y=10; y<gray.rows-10; y++ )
4     {
5         Eigen::Vector2d delta (
6             gray.ptr<uchar>(y)[x+1] - gray.ptr<uchar>(y)[x-1],
7             gray.ptr<uchar>(y+1)[x] - gray.ptr<uchar>(y-1)[x]
8         );
9         if ( delta.norm() < 50 )
10            continue;
11         ushort d = depth.ptr<ushort> (y)[x];
12         if ( d==0 )
13            continue;
14         Eigen::Vector3d p3d = project2Dto3D ( x, y, d, fx, fy, cx, cy, depth_scale );
15         float grayscale = float ( gray.ptr<uchar> (y) [x] );
16         measurements.push_back ( Measurement ( p3d, grayscale ) );
17     }

```

这只是一个很简单的改动。我们把先前的稀疏特征点改成了带有明显梯度的像素。于是在图优化中会增加许多的边。这些边都会参与估计相机位姿的优化问题，利用大量的像素而不单单是稀疏的特征点。由于我们并没有使用所有的像素，所以这种方式又称为**半稠密方法 (Semi-dense)**。我们把参与估计的像素取出来并把它们在图像中显示出来，如图 8-5 所示。

如果读者亲自做了实验，就可以看到参与估计的像素，像是固定在空间中一样。当相机旋转时，它们的位置似乎没有发生变化。这代表了我们估计的相机运动是正确的。同时，你可以检查我们使用的像素数量与优化时间的关系。显然，当像素增多时，优化会更加费时，所以为了实时性，需要考虑使用较好的像素点，或者降低图像的分辨率。不过对于演示实验来说，我认为这样已经能够让读者理解直接法的意义了。

8.5.5 直接法的讨论

相比于特征点法，直接法完全依靠优化来求解相机位姿。从式 (8.16) 中可以看到，像素梯度引导着优化的方向。如果我们想要得到正确的优化结果，就必须保证大部分像素梯度能够把优化引导到正确的方向。

这是什么意思呢？我们不妨设身处地地扮演一下优化算法。假设对于参考图像，我们测量到一个灰度值为 229 的像素。并且，由于我们知道它的深度，可以推断出空间点 P 的位置（图 8-6 中在 I_1 中测量到的灰度）。

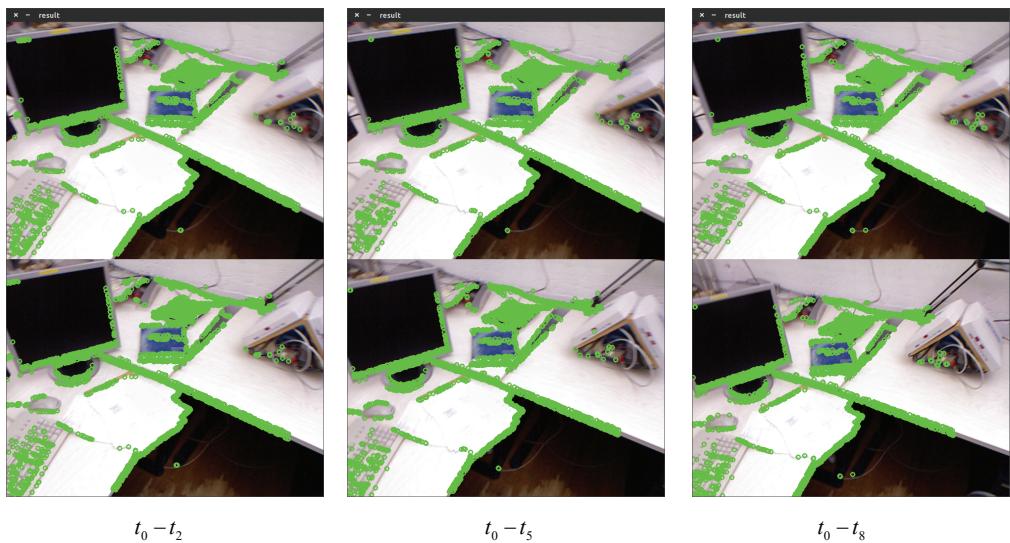


图 8-5 半稠密直接法的实验。参考帧与 2,5,8 帧的对比，绿色为参与优化的像素。

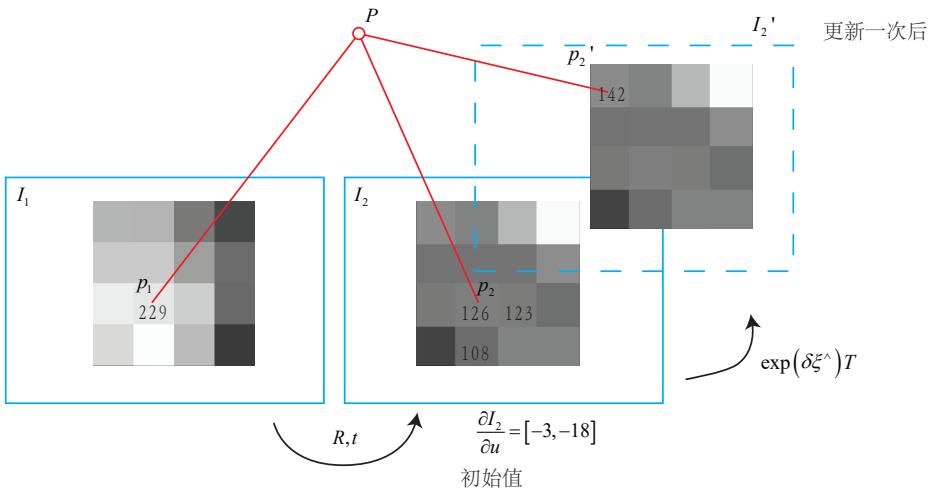


图 8-6 一次迭代的图形化显示。

此时我们又得到了一张新的图像，需要估计它的相机位姿。这个位姿是由一个初值不断地优化迭代得到的。假设我们的初值比较差，在这个初值下，空间点 P 投影后的像素灰度值是 126。于是，这个像素的误差为 $229 - 126 = 103$ 。为了减小这个误差，我们希望微调相机的位姿，使像素更亮一些。

怎么知道往哪里微调，像素会更亮呢？这就需要用到局部的像素梯度。我们在图像中发现，沿 u 轴往前走一步，该处的灰度值变成了 123，即减去了 3。同样地，沿 v 轴往前走一步，灰度值减 18，变成 108。在这个像素周围，我们看到梯度是 $[-3, -18]$ ，为了提高亮度，我们会建议优化算法微调相机，使 P 的像往左上方移动。在这个过程中，我们用像素的局部梯度近似了它附近的灰度分布，不过请注意真实图像并不是光滑的，所以这个梯度在远处就不成立了。

但是，优化算法不能只听这个像素的一面之词，还需要听取其他像素的建议^①。综合听取了许多像素的意见之后，优化算法选择了一个和我们建议的方向偏离不远的地方，计算出一个更新量 $\exp(\xi^\wedge)$ 。加上更新量后，图像从 I_2 移动到了 I'_2 ，像素的投影位置也变到了一个更亮的地方。我们看到，通过这次更新，误差变小了。在理想情况下，我们期望误差会不断下降，最后收敛。

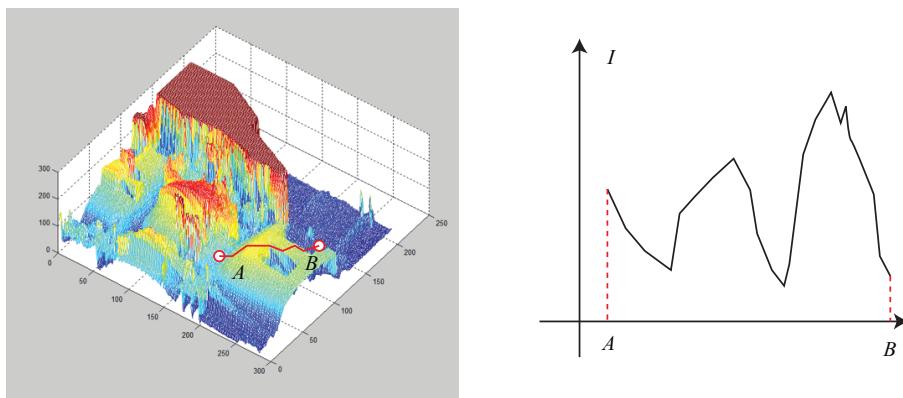


图 8-7 一张图像的三维化显示。从图像中的一个点运动到另一个点的路径不见得是“笔直的下坡路”，而需要经常的“翻山越岭”。这体现了图像本身的非凸性。

但是实际是不是这样呢？我们是否真的只要沿着梯度方向走，就能走到一个最优值？注意到，直接法的梯度是直接由图像梯度确定的，因此我们必须保证沿着图像梯度走时，灰度误差会不断下降。然而，图像通常是一个很强烈的非凸函数，如图 8-7 所示。实际当中，如果我们沿着图像梯度前进，很容易由于图像本身的非凸性（或噪声）落进一个局部极小

^① 这可能是一种不严谨的拟人化说法，不过有助于理解。

值中，无法继续优化。只有当相机运动很小，图像中的梯度不会有很强的非凸性时，直接法才能成立。

在例程中，我们只计算了单个像素的差异，并且这个差异是由灰度直接相减得到的。然而，单个像素没有什么区分性，周围很可能有好多像素和它的亮度差不多。所以，我们有时会使用小的图像块（patch），并且使用更复杂的差异度量方式，例如归一化相关性（Normalized Cross Correlation, NCC）等（见 13 讲）。而例程为了简单起见，使用了误差的平方和，以保持和推导的一致性。

8.5.6 直接法优缺点总结

最后，我们总结一下直接法的优缺点。大体来说，它的优点如下：

- 可以省去计算特征点、描述子的时间。
- 只要求有像素梯度即可，无须特征点。因此，直接法可以在特征缺失的场合下使用。比较极端的例子是只有渐变的一张图像。它可能无法提取角点类特征，但可以用直接法估计它的运动。
- 可以构建半稠密乃至稠密的地图，这是特征点法无法做到的。

另一方面，它的缺点也很明显：

- **非凸性**——直接法完全依靠梯度搜索，降低目标函数来计算相机位姿。其目标函数中需要取像素点的灰度值，而图像是强烈非凸的函数。这使得优化算法容易进入极小，只在运动很小时直接法才能成功。
- **单个像素没有区分度**。找一个和他像的实在太多了！——于是我们要么计算图像块，要么计算复杂的相关性。由于每个像素对改变相机运动的“意见”不一致。只能少数服从多数，以数量代替质量。
- **灰度值不变是很强的假设**。如果相机是自动曝光的，当它调整曝光参数时，会使得图像整体变亮或变暗。光照变化时亦会出现这种情况。特征点法对光照具有一定的容忍性，而直接法由于计算灰度间的差异，整体灰度变化会破坏灰度不变假设，使算法失败。针对这一点，目前的直接法开始使用更细致的光度模型标定相机，以便在曝光时间变化时也能让直接法工作。

习题

1. 除了 LK 光流之外，还有哪些光流方法？它们各有什么特点？
2. 在本节的程序的求图像梯度过程中，我们简单地求了 $u + 1$ 和 $u - 1$ 的灰度之差除 2，作为 u 方向上的梯度值。这种做法有什么缺点？提示：对于距离较近的特征，变化应该较快；而距离较远的特征在图像中变化较慢，求梯度时能否利用此信息？
3. 在稀疏直接法中，假设单个像素周围小块的光度也不变，是否可以提高算法鲁棒性？请编程实现此事。
4. * 使用 Ceres 实现 RGB-D 上的稀疏直接法和半稠密直接法。
5. 相比于 RGB-D 的直接法，单目直接法往往更加复杂。除了匹配未知之外，像素的距离也是待估计的。我们需要在优化时把像素深度也作为优化变量。阅读 [59, 57]，你能理解它的原理吗？如果不能，请在 13 讲之后再回来阅读。
6. 由于图像的非凸性，直接法目前还只能用于短距离，非自动曝光的相机。你能否提出增强直接法鲁棒性的方案？阅读 [58, 60] 可能会给你一些灵感。