

第六章作业

Student name: Francisrk

Due date: March 13th, 2022

1 第 1 题

已阅。

2 第 2 题

2.1 光流文献综述

1. 按此文的分类，光流法可分为哪几类？
2. 在 compositional 中，为什么有时候需要做原始图像的 wrap？该 wrap 有何物理意义？
3. forward 和 inverse 有何差别？

2.2 forward-addtive Gauss-Newton 光流的实现

1. 从最小二乘角度来看，每个像素的误差怎么定义？

参考 [2] 定义原始图中每个点的像素为 $I_1(x_i, y_i)$ ，第二张图中的每个点的像素为 $I_2(x_i + \Delta x_i, y_i + \Delta y_i)$ ，则从最小二乘的角度来看，每个点的像素误差可以定义为：

$$e_i = \min_{\Delta x_i, \Delta y_i} \|I_1(x_i, y_i) - I_2(x_i + \Delta x_i, y_i + \Delta y_i)\|_2^2 \quad (2.1)$$

2. 误差相对于自变量的导数如何定义?

补充关于差分, 微分和导数的区别 [1] 这里, 待估计的变量是 Δx_i 和 Δy_i , 定义相应的导数为:

$$\begin{aligned}\frac{\partial e_i}{\partial \Delta x_i} &= -\frac{\partial I_2}{\partial \Delta x_i} \\ \frac{\partial e_i}{\partial \Delta y_i} &= -\frac{\partial I_2}{\partial \Delta y_i}\end{aligned}\quad (2.2)$$

因为图像中每个点之间的像素值是离散的, 不能直接用微分来求导, 这里使用差分来代替微分来求导, 使用中心差方式来进行求导计算。

令

$$\begin{aligned}f(x+1, y) &= I_2(x_i + \Delta x_i + 1, y_i + \Delta y_i) \\ f(x-1, y) &= I_2(x_i + \Delta x_i - 1, y_i + \Delta y_i)\end{aligned}\quad (2.3)$$

对其进行一阶泰勒展开:

$$\begin{aligned}f(x+1, y) &= f(x, y) + f'(x) \\ f(x-1, y) &= f(x, y) - f'(x)\end{aligned}\quad (2.4)$$

故对 x 有

$$f'(x) = \frac{f(x+1, y) - f(x-1, y)}{2}\quad (2.5)$$

对 y 有

$$f'(y) = \frac{f(x, y+1) - f(x, y-1)}{2}\quad (2.6)$$

最终, 相应的导数为:

$$\begin{aligned}\frac{\partial e_i}{\partial \Delta x_i} &= -\frac{\partial I_2}{\partial \Delta x_i} = -\frac{I_2(x_i + \Delta x_i + 1, y_i + \Delta y_i) - I_2(x_i + \Delta x_i - 1, y_i + \Delta y_i)}{2} \\ \frac{\partial e_i}{\partial \Delta y_i} &= -\frac{\partial I_2}{\partial \Delta y_i} = -\frac{I_2(x_i + \Delta x_i, y_i + \Delta y_i + 1) - I_2(x_i + \Delta x_i, y_i + \Delta y_i - 1)}{2}\end{aligned}\quad (2.7)$$

```
1 // TODO START YOUR CODE HERE (~8 lines)
2 // 计算误差
```

```

3         double error = GetPixelValue(img1, kp.pt.x + x, kp.pt.y
+ y)-GetPixelValue(img2, kp.pt.x + x + dx, kp.pt.y + y + dy);
4             Eigen::Vector2d J; // Jacobian
5             if (inverse == false)
6             {
7                 // Forward Jacobian 前向雅可比 (因为是离散的, 不能
8                 // 用微分, 使用中心差分方式来进行求导)
9                 J = -1.0 * Eigen::Vector2d(
10                     0.5 * (GetPixelValue(img2,kp.pt.x + dx + x +
11                         1, kp.pt.y + dy + y)-GetPixelValue(img2, kp.pt.x + dx + x -1, kp.pt.y
12                         + dy + y)),
13                     0.5 * (GetPixelValue(img2, kp.pt.x + dx + x,
14                         kp.pt.y+ dy + y + 1)- GetPixelValue(img2, kp.pt.x + dx + x, kp.pt.y +
15                         dy + y -1))
16                     );
17             }
18             else
19             {
20                 if(iter == 0 )
21                     // Inverse Jacobian
22                     // NOTE this J does not change when dx, dy is
23                     updated, so we can store it and only compute error
24                     //反向模式, 使用I1处的梯度替换I2处的梯度, 雅可比不随
25                     //dx, dy的改变而改变, 所以不加dx, dy
26                     {
27                         J = -1.0 * Eigen::Vector2d(
28                             0.5 * (GetPixelValue(img2,kp.pt.x + x +
29                                 1, kp.pt.y + y)-
30                                     GetPixelValue(img2, kp.pt.x + x
31                                 -1, kp.pt.y + y)),
32                                     0.5 * (GetPixelValue(img2, kp.pt.x + x,
33                                         kp.pt.y + y + 1)-
34                                         GetPixelValue(img2, kp.pt.x + x,
35                                         kp.pt.y + y -1))
36                         );
37                     }
38                     // compute H, b and set cost;
39                     b += -error * J;
40                     cost += error * error;
41                     if(inverse==false || iter==0) //如果是正向或者是第一次
42                     //迭代, 就需要更新系数矩阵H
43                     {
44                         H += J * J.transpose() ; //这里的雅可比定义出来是J
45                         ^T, 直接就是向量, 求H要得是矩阵, 所以得J*J^T
46                     }

```

```

35         // TODO END YOUR CODE HERE
36     }
37     // compute update 更新
38     // TODO START YOUR CODE HERE (~1 lines)
39     Eigen::Vector2d update = H.ldlt().solve(b); // 求解方程H[dx,dy]^
40     T=b
        // TODO END YOUR CODE HERE

```

Listing 1: forward-addtive Gauss-Newton 光流的实现核心代码

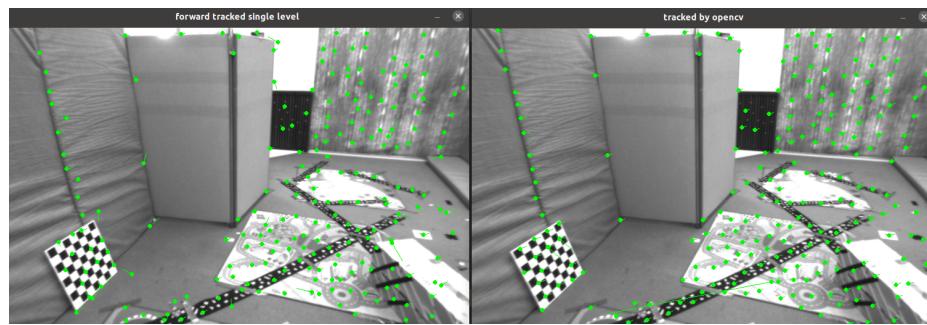


图 2.1 前向 G-N 光流与 OpenCV 光流对比

2.3 反向法

反向法的光流使用 I_1 的梯度替换了 I_2 的梯度，对应的导数为：

$$\begin{aligned}\frac{\partial I_1}{\partial x_i} &= \frac{I_2(x_i + 1, y_i) - I_2(x_i - 1, y_i)}{2} \\ \frac{\partial I_1}{\partial y_i} &= \frac{I_2(x_i, y_i + 1) - I_2(x_i, y_i - 1)}{2}\end{aligned}\quad (2.8)$$

代码部分见 Listing1，与 OpenCV 的对比见图 2.2，可见前向比反向法更好。

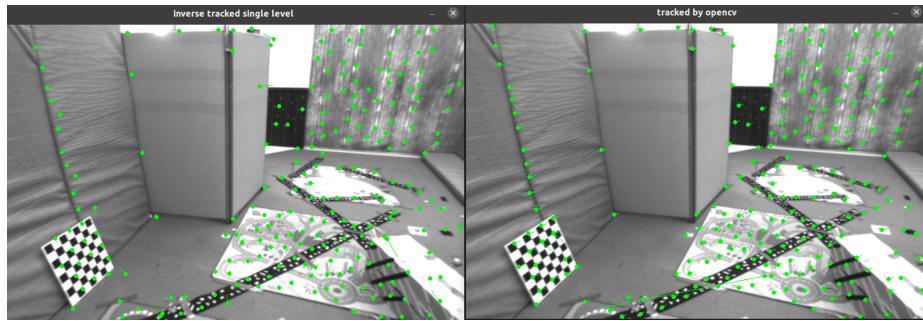


图 2.2 反向 G-N 光流与 OpenCV 光流对比

2.4 推广至金字塔

1. 所谓 coarse-to-fine 是指怎样的过程?

如图 2.3 所示, 左侧字底向上, 以原始图像作为最底层, 每往上一层, 就对下一层图像进行一定倍率的缩小, 得到一个金字塔; 计算光流时, 先将原始图像的特征点所放到左侧最顶层, 然后如右侧所示, 自顶向下逐层进行单层光流计算特征点, 并逐层放大, 本层放大后的特征点作为下层光流的初始值, 自顶向下的这个过程的特征点由大到小, 股也被称为由粗至精的过程 (coarse to fine)。

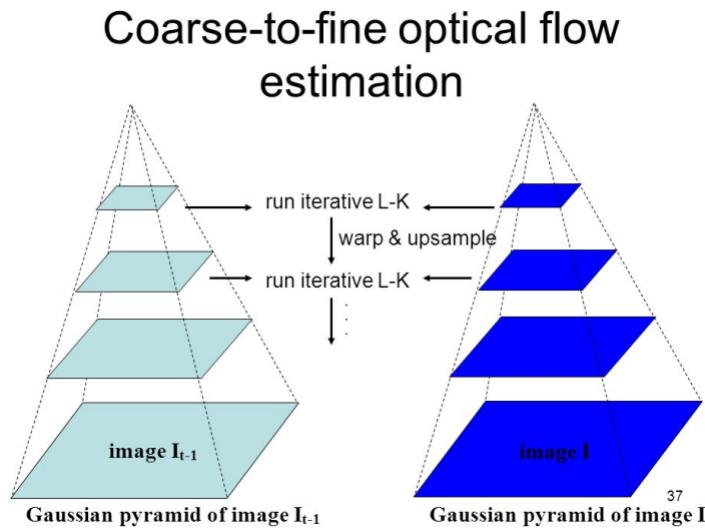


图 2.3 金字塔模型

2. 光流法中的金字塔用途和特征点法中的金字塔有何差别?

光流法金字塔，可以使得优化过程更易跳出，因为光流法作为一个优化问题，必须假设优化的初始值靠近最优值，才能在一定程度上保障算法的收敛，如果相机运动较快，两张图像差异明显，单层的光流法容易达到一个局部极小值。

特征点法金字塔，其作用是解决 Fast 角点从远处看是角点，而进到可能不是角点的尺度问题，实现 Fast 角点的尺度不变性。

金字塔代码如 Listing2 所示，正向和反向金字塔与 OpenCV 的对比如图 2.4 和 2.5 所示，可见金字塔对于正向和反向法的性能均有所提升。

```
1 void OpticalFlowMultiLevel(
2     const Mat &img1,
3     const Mat &img2,
4     const vector<KeyPoint> &kp1,
5     vector<KeyPoint> &kp2,
6     vector<bool> &success,
7     bool inverse) {
8
9     // parameters
10    int pyramids = 4; //4 层金字塔
11    double pyramid_scale = 0.5; //缩放率为 0.5
12    double scales[] = {1.0, 0.5, 0.25, 0.125};
13
14    // create pyramids
15    vector<Mat> pyr1, pyr2; // image pyramids
16    // TODO START YOUR CODE HERE (~8 lines)
17    for (int i = 0; i < pyramids; i++) {
18        if(i==0)
19        {
20            pyr1.push_back(img1);
21            pyr2.push_back(img2);
22        }
23        else
24        {
25            Mat img1_pyr, img2_pyr;
26            //自底向上缩放
27            cv::resize(pyr1[i-1], img1_pyr, cv::Size(pyr1[i-1].cols *
pyramid_scale, pyr1[i-1].rows * pyramid_scale));
28            cv::resize(pyr2[i-1], img2_pyr, cv::Size(pyr2[i-1].cols *
pyramid_scale, pyr2[i-1].rows * pyramid_scale));
29            pyr1.push_back(img1_pyr);
30            pyr2.push_back(img2_pyr);
31        }
```

```
32     }
33     // TODO END YOUR CODE HERE
34
35     // coarse-to-fine LK tracking in pyramids
36     // TODO START YOUR CODE HERE
37     vector<KeyPoint> kp1_pyr, kp2_pyr; //特征点金字塔
38
39 //    int tmp_level = pyramids;
40     for(auto &kp:kp1)
41     {
42         auto kp_top = kp;
43
44         kp_top.pt *= scales[pyramids - 1];
45         kp1_pyr.push_back(kp_top);
46         kp2_pyr.push_back(kp_top);
47     }
48
49     for(int level = pyramids-1; level>=0; level--)
50     {
51         success.clear();
52         OpticalFlowSingleLevel(pyr1[level], pyr2[level], kp1_pyr, kp2_pyr,
53         success, inverse);
54
55         if(level>0)
56         {
57             for(auto &kp: kp1_pyr) //引用， 改变源数据
58                 kp.pt /= pyramid_scale;
59             for(auto &kp: kp2_pyr)
60                 kp.pt /= pyramid_scale;
61         }
62     }
63     // don't forget to set the results into kp2
64     for(auto &kp: kp2_pyr)
65         kp2.push_back(kp);
66     // TODO END YOUR CODE HERE
67 }
```

Listing 2: 多层金字塔代码

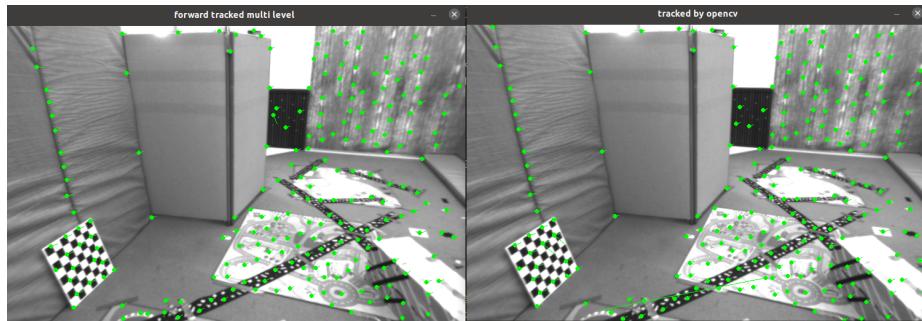


图 2.4 正向金字塔 G-N 光流与 OpenCV 光流对比

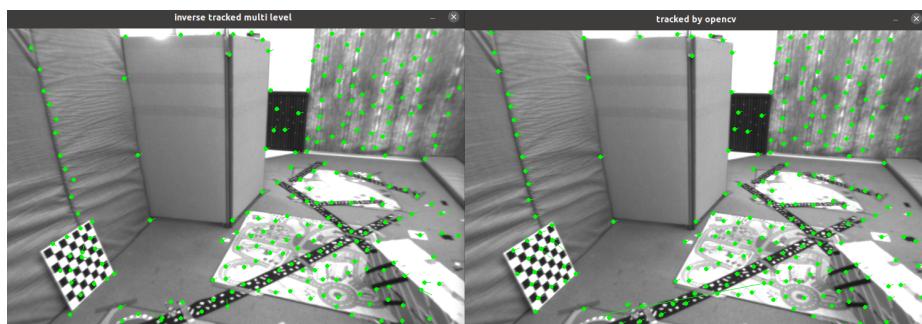


图 2.5 反向金字塔 G-N 光流与 OpenCV 光流对比

2.5 并行化

引入一个额外的 vector 用于存放下标 i，核心代码如 Listing3 所示。

```

1  vector<int> indexes;
2  for (int (i) = 0; (i) < kp1.size(); +(i))
3      indexes.push_back(i);
4  std::mutex m;
5  std::lock_guard<std::mutex> guard(m); // 代替 m.lock; m.unlock();
6  for_each(execution::par_unseq, indexes.begin(), indexes.end(),
7          [&](auto& i)
8          {
9              // 和单层光流一样
10         });

```

Listing 3: 并行化核心代码

```
/home/wrk/SLAM/DeepBlueCurriculum/VSLAM/ch6_直接法视觉里程计/Francisrk-ch6作业/Programs/T2_2/cm
方法 optical flow by SingleLevel_Forward 平均调用时间/次数: 1.01998/1 毫秒.
方法 optical flow by SingleLevelMT_Forward 平均调用时间/次数: 0.330086/1 毫秒.
方法 optical flow by MultiLevel_Forward 平均调用时间/次数: 5.80615/1 毫秒.
方法 optical flow by OpenCV 平均调用时间/次数: 0.457568/1 毫秒.
```

图 2.5 并行化对比

可以看出，光流法在并行化之后速度提升 0.7ms 左右，有较大提升。

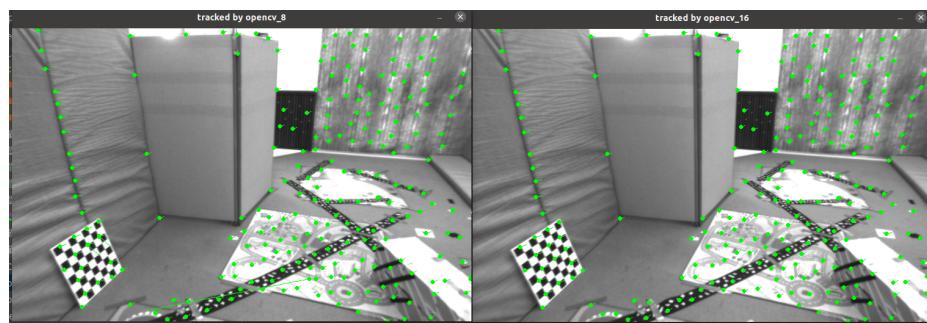
2.6 讨论

1. 我们优化两个图像块的灰度之差真的合理吗？哪些时候不够合理？你有解决办法吗？

优化两个图像块的灰度之差是在强假设“灰度不变假设”下进行的，即同一个空间点的像素灰度值，在各个图像中是固定不变的。当灰度不变假设成立时，这样优化合理，当物体材质变化较大、像素亮度发生剧烈变化时这样优化就不太合理。可以使用像素的相对变化来进行光流，对图像中的每个像素值减去均值，得到相对变化来做光流。还可以对像素值进行归一化。

2. 图像块大小是否有明显差异？取 16×16 和 8×8 的图像块会让结果发生变化吗？

使用 OpenCV 对 8×8 和 16×16 进行了对比，发现 16×16 的窗口效果要更好一些， 16×16 的基本上没有错误的方向，说明适当调整窗口大小能够优化光流追踪性能，窗口大鲁棒性好，窗口小精确性好。对比结果如图 2.6 所示。

图 2.6 8×8 和 16×16 的窗口对比

3. 金字塔层数对结果有怎样的影响？缩放倍率呢？

金字塔层数越多，追踪的准确度越好，但是层数越多，追踪到的点就越少、越密集，边缘的点容易被忽略，越容易引起误追踪，如图 2.7 是 3, 5, 7 层金字塔的对比。

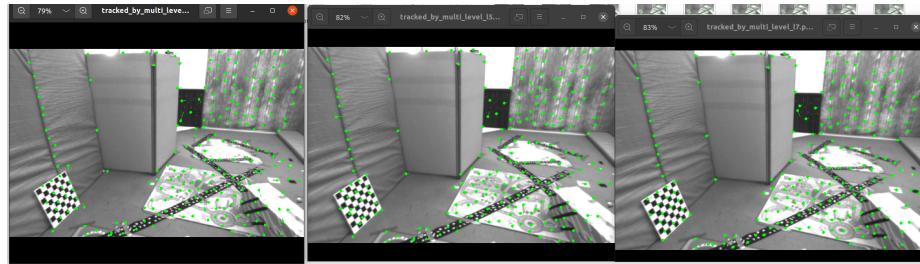


图 2.7 3、5、7 层金字塔对比

当缩放率变化时，按理来说应该变化很大（网上说倍率越小，得到的点越少，且倍率的影响比金字塔层数更大），但是经过我的实践，发现变化并不大，找了好久也没找到 bug 在哪里，图 2.8 是 0.25, 0.5, 0.75 的缩放率的对比

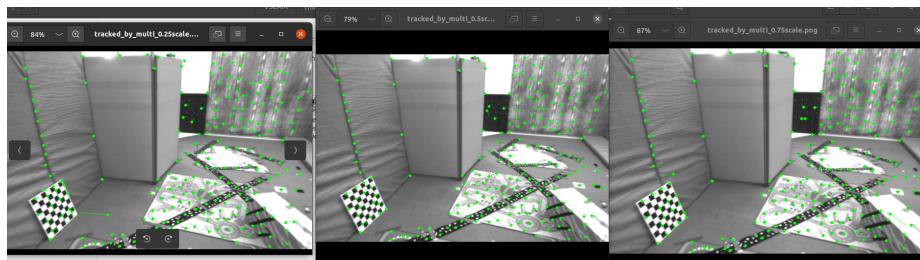


图 2.7 0.25, 0.5, 0.75 缩放率对比

3 直接法

3.1 单层直接法

1. 该问题中的误差项是什么？

在光流法中是有特征匹配的，相当于知道了 P 在两张图片上的投影像素位置 P_1, P_2 ，所以可以计算重投影的位置。而直接法事先不知道点 P_1 对应哪个点 P_2 ，根据当前对相机位置的估计来寻找点 P_2 的位置，

进而通过最小化两点之间的光度误差来调整相机的位姿，本题的误差为

$$e = I_{ref}(\pi(p_i)) - I_{cur}(\pi(p_i)) \quad (3.1)$$

2. 误差相对于自变量的雅克比维度是多少？如何求解？

参照教材 P218-220 的推导，误差相对于李代数的雅可比矩阵由两部分组成：像素梯度 ($\frac{\partial I_2}{\partial \mathbf{u}}$) 和像素坐标对李代数左扰动的梯度 ($\frac{\partial \mathbf{u}}{\partial \delta \xi}$) 前者维数为 (1,2)，后者为 (2,6)，故雅可比的维度为 (1,6)，关于像素梯度 $\frac{\partial I_2}{\partial \mathbf{u}}$ ，实际上这里简记为标量对行向量求导，分母的转置省略了，于是有

$$\begin{aligned} \frac{\partial I_2}{\partial \mathbf{u}} &= \frac{\partial I_{cur}}{\partial \mathbf{u}_i} \\ &= \left[\frac{\partial I_{cur}(u_i, v_i)}{\partial u_i} \quad \frac{\partial I_{cur}(u_i, v_i)}{\partial v_i} \right]_{1 \times 2} \end{aligned} \quad (3.2)$$

注意：(3.2) 中是标量函数对行向量的求导，这里行向量的转置符号省略。

因为像素值不连续，所以使用差分代替求导，差分有前向差分、后向差分、中心差分三种，这里选择中心差分，故：

$$\frac{\partial I_{cur}}{\partial \mathbf{u}_i} = -\frac{1}{2} \left[I_{cur}(u+1, v) - I_{cur}(u-1, v) \quad I_{cur}(u, v+1) - I_{cur}(u, v-1) \right]_{1 \times 2} \quad (3.3)$$

对于第二部分 $\frac{\partial \mathbf{u}}{\partial \delta \xi}$ ，按照教材上的推导为

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial \delta \xi} &= \frac{\partial \mathbf{u}_i}{\partial \delta \xi} \\ &= \left[\begin{matrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} & -\frac{f_x X Y}{Z^2} & f_x + \frac{f_x X^2}{Z^2} & -\frac{f_x Y}{Z} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} & -f_y - \frac{f_y Y^2}{Z^2} & \frac{f_y X Y}{Z^2} & \frac{f_y X}{Z} \end{matrix} \right]_{2 \times 6} \end{aligned} \quad (3.4)$$

所以整体的雅可比矩阵为

$$\begin{aligned} \mathbf{J} &= -\frac{\partial I_{cur}}{\partial \mathbf{u}_i} \frac{\partial \mathbf{u}_i}{\partial \delta \xi} \\ &= -\frac{1}{2} \left[I_{cur}(u+1, v) - I_{cur}(u-1, v) \quad I_{cur}(u, v+1) - I_{cur}(u, v-1) \right]_{1 \times 2} \\ &\quad * \left[\begin{matrix} \frac{f_x}{Z} & 0 & -\frac{f_x X}{Z^2} & -\frac{f_x X Y}{Z^2} & f_x + \frac{f_x X^2}{Z^2} & -\frac{f_x Y}{Z} \\ 0 & \frac{f_y}{Z} & -\frac{f_y Y}{Z^2} & -f_y - \frac{f_y Y^2}{Z^2} & \frac{f_y X Y}{Z^2} & \frac{f_y X}{Z} \end{matrix} \right]_{2 \times 6} \end{aligned} \quad (3.5)$$

核心代码如 Listing4 所示，对第一张图的直接法结果如图 3.1 所示：

```

37             (1.0 / 2) * (GetPixelValue(img2, u+x, v+1+y)-
38             GetPixelValue(img2, u+x, v-1+y));
39
40             Matrix2d J_pixel_xi; // pixel to \xi in Lie algebra
41             2*6
42             J_pixel_xi<<fx * inv_z,
43                         0,
44                         -fx * X * inv_z2,
45                         -fx * X * Y * inv_z2,
46                         fx + fx * X * X * inv_z2,
47                         -fx * Y * inv_z,
48                         0,
49                         fy * inv_z,
50                         -fy * Y * inv_z2,
51                         -fy - fy * Y * Y * inv_z2,
52                         fy * X * Y * inv_z2,
53                         fy * X * inv_z;
54
55             // total jacobian 应该是1*6的
56             Vector6d J=-1.0 * (J_img_pixel.transpose() * J_pixel_xi)
57             .transpose();
58
59             H += J * J.transpose();
60             b += -error * J;
61             cost += error * error;
62         }
63         // END YOUR CODE HERE
64     }
65
66     // solve update and put it into estimation
67     // TODO START YOUR CODE HERE
68     Vector6d update = H.ldlt().solve(b);
69     T21 = Sophus::SE3d::exp(update) * T21; // 李群更新
70     // END YOUR CODE HERE
71

```

Listing 4: 单层直接法核心代码



图 3.1 单层直接法第一张图结果

3.2 多层直接法

- 在缩放图像时，图像内参也需要跟着变化。那么，例如图像缩小一倍， fx, fy, cx, cy 应该如何变化？

相机内参也应该对应金字塔的层数做与图像对应的缩放，如 Listing5 所示

```

1 void DirectPoseEstimationMultiLayer(
2     const cv::Mat &img1,
3     const cv::Mat &img2,
4     const VecVector2d &px_ref,
5     const vector<double> depth_ref,
6     Sophus::SE3d &T21,
7     string order
8 ) {
9
10    // parameters 4层2倍金字塔
11    int pyramids = 4;
12    double pyramid_scale = 0.5;
13    double scales[] = {1.0, 0.5, 0.25, 0.125};
14
15    // create pyramids
16    vector<cv::Mat> pyr1, pyr2; // image pyramids
17    // TODO START YOUR CODE HERE 构建图像金字塔
18    for(int i=0; i<pyramids; i++)
19    {
20        if(i==0)
21        {

```

```

22         pyr1.push_back(img1);
23         pyr2.push_back(img2);
24     }
25     else
26     {
27         Mat img1_pyr, img2_pyr;
28         //自底向上缩放
29         cv::resize(pyr1[i-1], img1_pyr, cv::Size(pyr1[i-1].cols *
30             pyramid_scale, pyr1[i-1].rows * pyramid_scale));    //Size(width,
31             height)
32         cv::resize(pyr2[i-1], img2_pyr, cv::Size(pyr2[i-1].cols *
33             pyramid_scale, pyr2[i-1].rows * pyramid_scale));
34         pyr1.push_back(img1_pyr);
35         pyr2.push_back(img2_pyr);
36     }
37 }
38
39 // END YOUR CODE HERE
40 //构建特征点金字塔
41 double fxG = fx, fyG = fy, cxG = cx, cyG = cy; // backup the old
42 values
43 for (int level = pyramids - 1; level >= 0; level--) {
44     VecVector2d px_ref_pyr; // set the keypoints in this pyramid
45     level
46     for (auto &px: px_ref) {
47         px_ref_pyr.push_back(scales[level] * px);
48     }
49
50     // TODO START YOUR CODE HERE
51     // scale fx, fy, cx, cy in different pyramid levels
52     fx = fxG * scales[level];
53     fy = fyG * scales[level];
54     cx = cxG * scales[level];
55     cy = cyG * scales[level];
56     // END YOUR CODE HERE
57     DirectPoseEstimationSingleLayer(pyr1[level], pyr2[level],
58         px_ref_pyr, depth_ref, T21, order);
59 }

```

Listing 5: DirectPoseEstimationMultiLayer 函数

第 5 张图的结果如图 3.6 所示



图 3.2 多层直接法, 图 5 第 1 层



图 3.3 多层直接法, 图 5 第 2 层

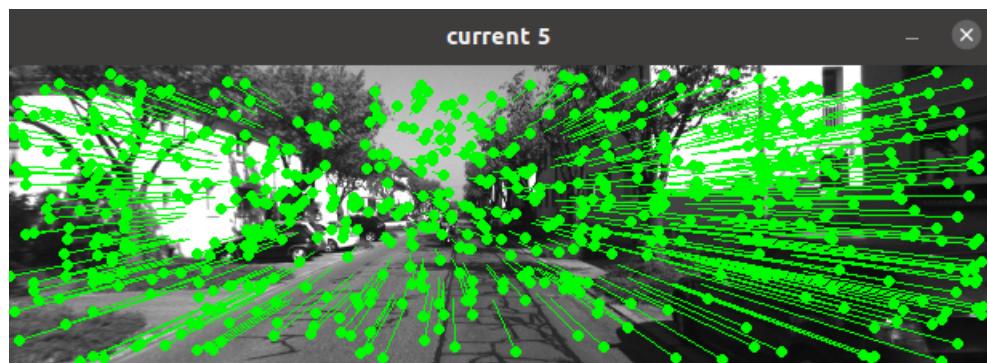


图 3.4 多层直接法, 图 5 第 3 层



图 3.5 多层直接法，图 5 第 4 层

```
T21 =
    0.999803  0.00120276  0.0198247  0.0188679
-0.00133237  0.999978  0.00652592 -0.0102066
-0.0198164 -0.00655105  0.999782  -3.79297
        0          0          0          1
```

图 3.6 估计结果

3.3 并行化

使用 `for_each` 和 `execution` 进行改进，在遍历 `ref` 中的特征点处使用多线程并发，注意在对有序容器进行插入前需要加锁，核心代码如下：

```

1 // Define Hessian and bias
2 Matrix6d H = Matrix6d::Zero(); // 6x6 Hessian
3 Vector6d b = Vector6d::Zero(); // 6x1 bias
4
5     vector<int> ref_index;
6     for(int i=0;i<px_ref.size();++i)
7         ref_index.push_back(i);
8
9     std::mutex m;
10    for_each(execution::par_unseq, ref_index.begin(), ref_index.end(),
11              [&](auto& i)
12  {
13      // compute the projection in the second image
14      // TODO START YOUR CODE HERE

```

```

15     Eigen::Vector3d point_ref = depth_ref[i] * Eigen::
16     Vector3d((px_ref[i][0]-cx)/fx, (px_ref[i][1]-cy)/fy, 1); //ref中的3D点
17     //坐标
18     Eigen::Vector3d point_cur = T21 * point_ref; //ref中的
19     //3D点转换到cur中的3D点
20     if (point_cur[2] >= 0) // depth invalid
21     {
22         float u = fx * point_cur[0]/point_cur[2] + cx, v =
23             fy * point_cur[1]/point_cur[2] + cy;
24         if(u>=half_patch_size && u+half_patch_size<=img2.
25             cols && v>=half_patch_size && v+half_patch_size<=img2.rows) //变换到
26             cur中若越界则不优化
27         {
28             double X = point_cur[0], Y = point_cur[1], Z =
29             point_cur[2], inv_z = 1.0 / Z, inv_z2 = inv_z * inv_z; //cur中的3D坐标
30             X'Y'Z'
31             nGood++;
32             std::lock_guard<std::mutex> guard(m); //代替m.
33             lock; m.unlock();
34             //记录投影前后的uv坐标
35             goodProjection.push_back(Eigen::Vector2d(u, v))
36             ;
37             GoodRefIndex.push_back(Eigen::Vector2d(px_ref[i
38                 ][0], px_ref[i][1]));

```

Listing 6: DirectPoseEstimationMultiLayer 函数

选择两张 000001.png 和 000002.png 分别与单进程进行对比，普通单层和并行化单层运行时间比较如图 3.7-3.8 所示

```

good projection: 989
T21 =
0.999991 0.00242132 0.00337216 -0.00184408
-0.00242871 0.999995 0.00218895 0.0026733
-0.00336684 -0.00219713 0.999992 -0.725126
0 0 0 1
方法 直接法 平均调用时间/次数: 178.85/1 毫秒.
cost = 137911, good = 1000

```

```
good projection: 989
T21 =
    0.999991  0.00242132  0.00337216 -0.00184408
   -0.00242871    0.999995  0.00218895   0.0026733
   -0.00336684 -0.00219713    0.999992   -0.725126
        0          0          0          1
方法 直接法MT 平均调用时间/次数: 467.574/1 毫秒.
cost = 133313, good = 989
```

图 3.7 估计结果 _1

```
cost increased: 29782.5, 29782.5
good projection: 928
T21 =
    0.999972  0.0013728  0.00728926  0.00740609
   -0.00140114    0.999991  0.00388439 -0.00131596
   -0.00728387 -0.00389449    0.999966      -1.4707
        0          0          0          1
方法 直接法 平均调用时间/次数: 217.528/1 毫秒.
cost = 133313, good = 989
```

```
good projection: 926
T21 =
    0.999972  0.00137274  0.00728994  0.0073672
   -0.00140108    0.999991  0.00388405 -0.00130557
   -0.00728455 -0.00389415    0.999966      -1.47066
        0          0          0          1
方法 直接法MT 平均调用时间/次数: 653.143/1 毫秒.
cost = 135273, good = 928
cost = 130781, good = 927
```

图 3.8 估计结果 _2

可以看出，并行化之后速度略微有些变慢，具体是什么原因呢？

3.4 延伸讨论

1. 直接法是否可以类似光流，提出 inverse, compositional 的概念？它们有意义吗？

可以提出 inverse,compositional 的概念，但是没有意义。因为，直接法必须计算两张图片整体的光度误差，而不是估计图像中像素的相对运动。

2. 请思考上面算法哪些地方可以缓存或加速？

在遍历每个选择的特征点时可以使用并行加速遍历。另外，窗口的大小可以适当调整，不是越大越好，窗口越大计算量越大，速度越慢。

3. 在上述过程中，我们实际假设了哪两个 patch 不变？

1. 灰度值不变（同一空间点在各个视角下成像的灰度不变）
2. 同一窗口内的深度信息不变

如下列代码所示：

```
1 double error = GetPixelValue(img1, px_ref[i][0]+x, px_ref[i][1]+y) -  
    GetPixelValue(img2, u+x, v+y);
```

4. 为何可以随机取点？而不用取角点或线上的点？那些不是角点的地方，投影算对了吗？

因为直接法对计算光度误差进行优化，雅可比表示当像素梯度不为 0 即对优化有贡献，所以不是角点的地方，只要保证有像素梯度即可。

5. 请总结直接法相对于特征点法的异同与优缺点。

优点：

1. 可以省去计算特征点、描述子的时间。
2. 只要求有像素梯度即可，不需要特征点。
3. 可以构建半稠密乃至稠密的地图，这点是特征点无法做到的。

缺点：

1. 非凸性。
2. 单个像素没有区分度。
3. 灰度值不变是很强的假设。具体在十四讲第二版 P230-231。

4 使用光流计算视差

使用 LK 光流计算 left 中的 GFTT 特征点在 right 中的对应，根据坐标的对应关系计算出水平视差，计算视差的核心代码如 Listing7 所示，计算结果如图 4.1 所示，OpenCV 的光流法之后的视差平均误差为-24.5884。

```
1 //OpenCV
2 Mat img2_CV;
3 cv::cvtColor(img2, img2_CV, CV_GRAY2BGR);
4 for (int i = 0; i < pt2.size(); i++) {
5     if (status[i]) {
6         cv::circle(img2_CV, pt2[i], 2, cv::Scalar(0, 250, 0), 2);
7         cv::line(img2_CV, pt1[i], pt2[i], cv::Scalar(0, 250, 0));
8         dis = kp1[i].pt.x - kp2_single[i].pt.x;
9         cost.push_back(dis - GetPixelValue(disparity_img, kp1[i].pt.x,
10                                         kp1[i].pt.y));
11    }
12 }
13 cost_sum = accumulate(cost.begin(), cost.end(), 0.0); //求和
14 cout<<"OpenCV 光流平均误差: "<<cost_sum/cost.size()<<endl;
15 cost.clear();
```

Listing 7: 计算视差的核心代码



图 4.1 视差计算结果

参考文献

- [1] <https://zhuanlan.zhihu.com/p/101092663>
- [2] <https://blog.csdn.net/jiachang98/article/details/121269827?spm=1001.2014.3001.5502>