

2018 年第 5 次课习题答案

——Vance 吴方熠

2 ORB 特征点 (4 分, 约 2.5 小时)

ORB(Oriented FAST and BRIEF) 特征是 SLAM 中一种很常用的特征, 由于其二进制特性, 使得它可以非常快速地提取与计算 [1]。下面, 你将按照本题的指导, 自行书写 ORB 的提取、描述子的计算以及匹配的代码。代码框架参照 computeORB.cpp 文件, 图像见 1.png 文件和 2.png。

2.1 ORB 提取

ORB 即 Oriented FAST 简称。它实际上是 FAST 特征再加上一个旋转量。本习题将使用 OpenCV 自带的 FAST 提取算法, 但是你要完成旋转部分的计算。旋转的计算过程描述如下 [2]:

在一个小图像块中, 先计算质心。质心是指以图像块灰度值作为权重的中心。

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:

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

实际上只需计算 m_{01} 和 m_{10} 即可。习题中取图像块大小为 16×16 , 即对于任意点 (u, v) , 图像块从 $(u-8, v-8)$ 取到 $(u+7, v+7)$ 即可。请在习题的 computeAngle 中, 为所有特征点计算这个旋转角。

解:

计算思路在题中已经给出, 特征点及其方向的可视化结果如下图所示:



图 1 特征点及其方向的可视化结果

特征点旋转角计算的自编代码部分如下所示：

```
... ..
115 // compute the angle
116 void computeAngle(const cv::Mat &image, vector<cv::KeyPoint>
    &keypoints) {
117     int half_patch_size = 8;
118     for (auto &kp : keypoints) {
119         // START YOUR CODE HERE (~7 lines)
120         int m_10 = 0, m_01 = 0;
121         for (int u = -half_patch_size; u <= half_patch_size-1;
++u) { // col
122             for (int v = -half_patch_size; v <= half_patch_size-
1; ++v) { // row
123                 int u1 = cvRound(kp.pt.x) + u, v1 =
cvRound(kp.pt.y) + v; // u1, kp.pt.x(col); v1, kp.pt.y(row)
124                 if (u1 >= 0 && u1 < image.cols && v1 >= 0 && v1 <
image.rows) {
125                     m_10 += u * image.at<uchar>(v1, u1);
126                     m_01 += v * image.at<uchar>(v1, u1);
127                 } else continue;
128             }
129         }
130         kp.angle = cv::fastAtan2((float)m_01, (float)m_10);
131         // END YOUR CODE HERE
132     }
133     return;
134 }
... ..
```

编程过程的心得：

- 1) 质心计算以特征点为中心，故在计算过程中任意点 (u, v) 的坐标不需要加上特征点的坐标；
- 2) 注意 `image.at<uchar>(y, x)` 的使用， x, y 容易搞反。
- 3) `cv::fastAtan2` 可直接返回 $0\sim 360^\circ$ 的值。

2.2 ORB 描述

ORB 描述即带旋转的 BRIEF 描述。所谓 BRIEF 描述是指一个 0-1 组成的字符串（可以取 256 位或 128 位），每一个 bit 表示一次像素间的比较。算法流程如下：

1. 给定图像 I 和关键点 (u, v) ，以及该点的转角 θ 。以 256 位描述为例，那么最终描述子

$$\mathbf{d} = [d_1, d_2, \dots, d_{256}].$$

2. 对任意 $i = 1, \dots, 256$ ， d_i 的计算如下。取 (u, v) 附近任意两个点 \mathbf{p}, \mathbf{q} ，并按照 θ 进行旋转：

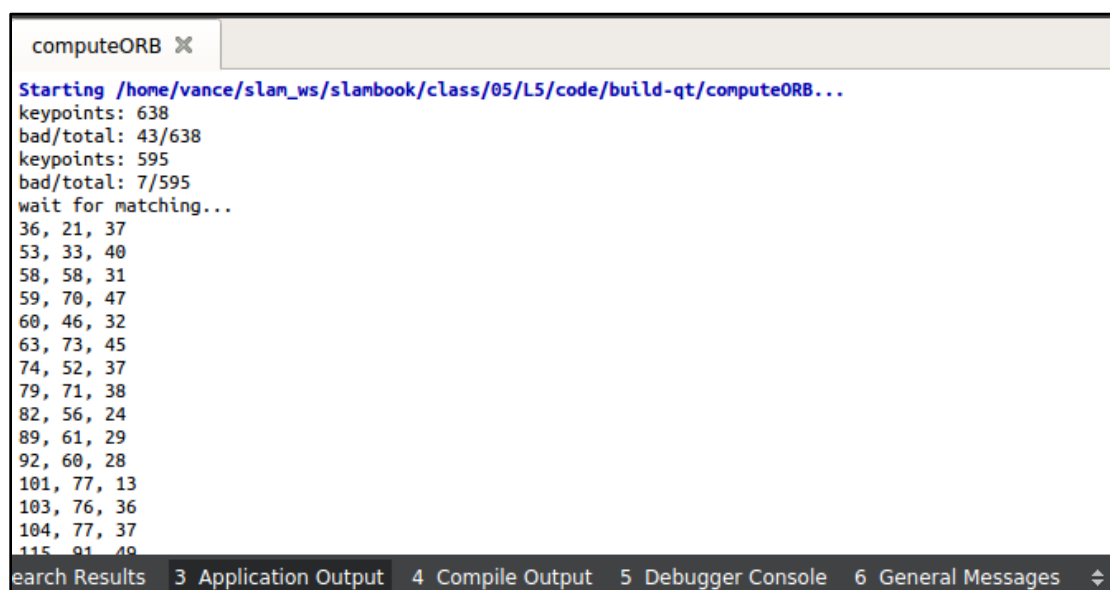
$$\begin{bmatrix} u_p' \\ v_p' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} u_p \\ v_p \end{bmatrix}. \quad (1)$$

其中 u_p, v_p 为 \mathbf{p} 的坐标，对 \mathbf{q} 亦然。记旋转后的 \mathbf{p}, \mathbf{q} 为 \mathbf{p}', \mathbf{q}' ，那么比较 $I(\mathbf{p}')$ 和 $I(\mathbf{q}')$ ，若前者大，记 $d_i = 0$ ，反之记 $d_i = 1$ 。

这样我们就得到了 ORB 的描述。我们在程序中用 256 个 bool 变量表达这个描述^[9]。请你完成 `computeORBDesc` 函数，实现此处计算。注意，通常会固定 \mathbf{p}, \mathbf{q} 的取法（称为 ORB 的 pattern），否则每次都重新随机选取，会使得描述不稳定。我们在全局变量 `ORB_pattern` 中定义了 \mathbf{p}, \mathbf{q} 的取法，格式为 u_p, v_p, u_q, v_q 。请你根据给定的 pattern 完成 ORB 描述的计算。

解：

特征点描述坏点计算结果截图如下所示：



```
computeORB X
Starting /home/vance/slam_ws/slambook/class/05/L5/code/build-qt/computeORB...
keypoints: 638
bad/total: 43/638
keypoints: 595
bad/total: 7/595
wait for matching...
36, 21, 37
53, 33, 40
58, 58, 31
59, 70, 47
60, 46, 32
63, 73, 45
74, 52, 37
79, 71, 38
82, 56, 24
89, 61, 29
92, 60, 28
101, 77, 13
103, 76, 36
104, 77, 37
115, 91, 40
```

图 2 特征点描述坏点计算结果截图

计算出两幅图的坏点/特征点数量分别为：43/638，7/595，这些坏点产生的原因是因为该特征坏点接近于图像的边界，故其 pattern 容易超出图像边界，按程序中的处理，其描述子设为空，将其视为坏点。

特征点描述的自编代码部分如下所示：

```
... ..
398 // compute the descriptor
399 void computeORBDesc(const cv::Mat &image, vector<cv::KeyPoint>
&keypoints, vector<DescType> &desc) {
400     for (auto &kp : keypoints) {
401         DescType d(256, false);
402         for (int i = 0; i < 256; i++) {
403             // START YOUR CODE HERE (~7 lines)
404             int up = ORB_pattern[4*i];
405             int vp = ORB_pattern[4*i+1];
406             int uq = ORB_pattern[4*i+2];
407             int vq = ORB_pattern[4*i+3];
408             double angle = kp.angle*pi/180;
409             int u_p = cvRound( up*cos(angle) - vp*sin(angle) );
410             int v_p = cvRound( up*sin(angle) + vp*cos(angle) );
411             int u_q = cvRound( uq*cos(angle) - vq*sin(angle) );
412             int v_q = cvRound( uq*sin(angle) + vq*cos(angle) );
413             if (kp.pt.x+u_p < 0 || kp.pt.x+u_p > image.cols-1 ||
kp.pt.x+u_q < 0 || kp.pt.x+u_q > image.cols-1 || kp.pt.y+v_p <
0 || kp.pt.y+v_p > image.rows-1 || kp.pt.y+v_q < 0 ||
kp.pt.y+v_q > image.rows-1) {
414                 d.clear(); // if kp goes outside, set d.clear()
415                 break; // next keypoint
416             } else if ( image.at<uchar>(cvRound(kp.pt.y)+v_p,
cvRound(kp.pt.x)+u_p) <= image.at<uchar>(cvRound(kp.pt.y)+v_q,
cvRound(kp.pt.x)+u_q) ) {
417                 d[i] = 1;
418             }
419         }
420         // END YOUR CODE HERE
421         desc.push_back(d);
422     }
423 ... ..
```

注意：对于 pattern 中取的点对，应先旋转，后相加，否则旋转中心将会变成像素坐标系的(0, 0)点。

2.3 暴力匹配

在提取描述子之后，我们需要根据描述子进行匹配。暴力匹配是一种简单粗暴的匹配方法，在特征点不多时很有用。下面你将根据习题指导，书写暴力匹配算法。

所谓暴力匹配思路很简单。给定两组描述子 $\mathbf{P} = [p_1, \dots, p_M]$ 和 $\mathbf{Q} = [q_1, \dots, q_N]$ 。那么，对 \mathbf{P} 中任意一个点，找到 \mathbf{Q} 中对应最小距离点，即算一次匹配。但是这样做会对每个特征点都找到一个匹配，所以我们通常还会限制一个距离阈值 d_{max} ，即认作匹配的特征点距离不应该大于 d_{max} 。下面请你根据上述描述，实现函数 `bfMatch`，返回给定特征点的匹配情况。实践中取 $d_{max} = 50$ 。

解：

暴力匹配的结果如图 3，图 4 所示。

共匹配到 99 个特征点，总共花费时长约 4.2 秒。

```
computeORB x
608, 566, 41
613, 540, 40
614, 534, 37
615, 535, 44
621, 577, 48
622, 552, 39
631, 586, 41
632, 587, 37
633, 588, 35
634, 588, 44
matches: 99
matching time cost: 4.21488 seconds.
done.
/home/vance/slam_ws/slambook/class/05/L5/code/build-qt/computeORB exited with code 0
```

图 3 暴力匹配运行结果截图



图 4 暴力匹配效果可视化（阈值为 50）

代码编写心得：

- 1) 要注意对空描述子的跳过处理；
- 2) 循环内部的中间变量要及时保存在容器中，同样在每个特征点处理结束后又要及时清空容器。

暴力匹配的自编代码部分如下所示:

```
... ..
432 // brute-force matching
433 void bfMatch(const vector<DescType> &desc1, const
vector<DescType> &desc2, vector<cv::DMatch> &matches) {
434     int d_max = 50;
435
436     // START YOUR CODE HERE (~12 lines)
437     // find matches between desc1 and desc2.
438     vector<int> ham_dis, ham_dis_index;
439     cv::DMatch match;
440     for (int i=0; i<desc1.size(); i++) {
441         if (desc1[i].empty()) continue;
442
443         for (int j=0; j<desc2.size(); j++) {
444             if (desc2[j].empty()) continue;
445
446             int d = 0;
447             for (int k=0; k<256; k++) {
448                 if (desc1[i][k] != desc2[j][k]) d++; // 计算汉明距离
449             }
450             ham_dis.push_back(d); // desc1 中的第 i 个有效描述子与
desc2 中所有有效描述子的汉明距离
451             ham_dis_index.push_back(j); // 对应的索引
452         }
453
454         vector<int>::iterator d_min =
min_element(ham_dis.begin(), ham_dis.end()); // 最小值
455         int n = distance(ham_dis.begin(), d_min); // 最小值索引位置
456         if (*d_min <= d_max) {
457             match.queryIdx = i;
458             match.trainIdx = ham_dis_index[n];
459             match.distance = *d_min;
460             matches.push_back(match);
461         }
462
463         ham_dis.clear();
464         ham_dis_index.clear();
465     }
466     // END YOUR CODE HERE
... ..
```


最后，请结合实验，回答下面几个问题：

1. 为什么说 ORB 是一种二进制特征？
2. 为什么在匹配时使用 50 作为阈值，取更大或更小值会怎么样？
3. 暴力匹配在你的机器上表现如何？你能想到什么减少计算量的匹配方法吗？

答：

1. 主要是因为 ORB 特征采用的描述子为带旋转的 BRIEF 描述。所谓 BRIEF 描述是指由 0-1 组成的字符串，即二进制串的特征描述符，故有此说。

2. 若取更小值（如图 5，取为 30），则会出现匹配的点数过于稀少，不利于后期的相机位姿变换的计算；若取更大值（如图 6，取为 70），则会出现误匹配现象，这对后续的计算更加不利。当然，做完特征点的匹配后，我们还可以通过 RANSAC 方法计算透视变换矩阵来筛选符合相同透视的特征点，这样做可以去除很多错误的匹配。

3. 在本人机器人匹配 99 个点用时 4.2 秒，感觉时间有点长。对于减少计算量的匹配方法，可以使用 FLANN 快速近似最近邻算法来寻找匹配关系。

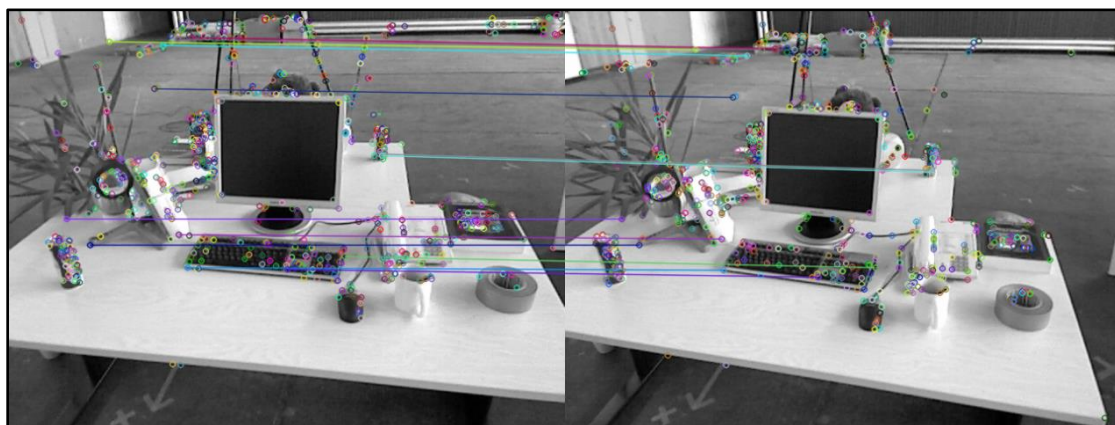


图 5 阈值为 30 时的匹配效果

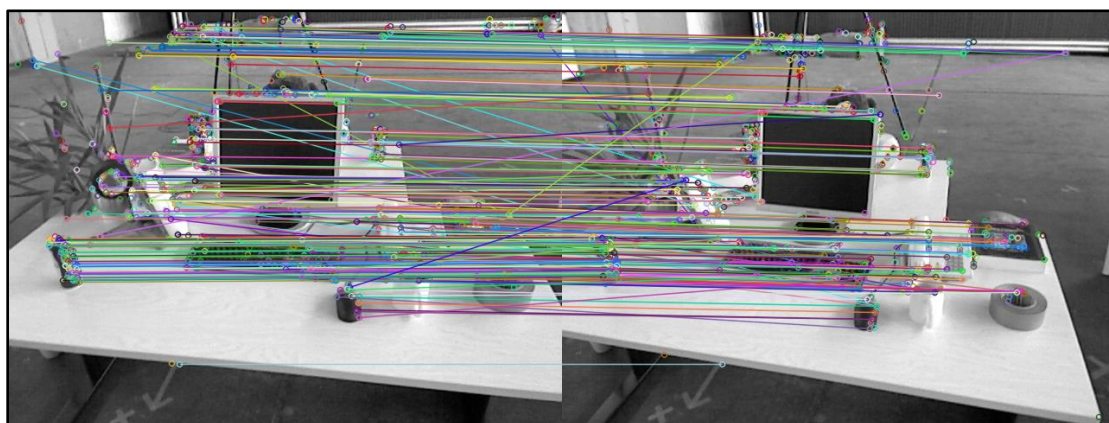


图 6 阈值为 70 时的匹配效果

3 从 \mathbf{E} 恢复 \mathbf{R}, \mathbf{t} (3 分, 约 1 小时)

我们在书中讲到了单目对极几何部分, 可以通过本质矩阵 \mathbf{E} , 得到旋转和平移 \mathbf{R}, \mathbf{t} , 但那时直接使用了 OpenCV 提供的函数。本题中, 请你根据数学原理, 完成从 \mathbf{E} 到 \mathbf{R}, \mathbf{t} 的计算。程序框架见 code/E2Rt.cpp。

设 Essential 矩阵 \mathbf{E} 的取值为 (与书上实验数值相同):

$$\mathbf{E} = \begin{bmatrix} -0.0203618550523477 & -0.4007110038118445 & -0.03324074249824097 \\ 0.3939270778216369 & -0.03506401846698079 & 0.5857110303721015 \\ -0.006788487241438284 & -0.5815434272915686 & -0.01438258684486258 \end{bmatrix}$$

· 请计算对应的 \mathbf{R}, \mathbf{t} , 流程如下:

1. 对 \mathbf{E} 作 SVD 分解:

$$\mathbf{E} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T.$$

2. 处理 $\mathbf{\Sigma}$ 的奇异值。设 $\mathbf{\Sigma} = \text{diag}(\sigma_1, \sigma_2, \sigma_3)$ 且 $\sigma_1 \geq \sigma_2 \geq \sigma_3$, 那么处理后的 $\mathbf{\Sigma}$ 为:

$$\mathbf{\Sigma} = \text{diag}\left(\frac{\sigma_1 + \sigma_2}{2}, \frac{\sigma_1 - \sigma_2}{2}, 0\right). \quad (2)$$

3. 共存在四个可能的解:

$$\begin{aligned} \mathbf{t}_1^\wedge &= \mathbf{U}\mathbf{R}_Z\left(\frac{\pi}{2}\right)\mathbf{\Sigma}\mathbf{U}^T, & \mathbf{R}_1 &= \mathbf{U}\mathbf{R}_Z^T\left(\frac{\pi}{2}\right)\mathbf{V}^T \\ \mathbf{t}_2^\wedge &= \mathbf{U}\mathbf{R}_Z\left(-\frac{\pi}{2}\right)\mathbf{\Sigma}\mathbf{U}^T, & \mathbf{R}_2 &= \mathbf{U}\mathbf{R}_Z^T\left(-\frac{\pi}{2}\right)\mathbf{V}^T. \end{aligned} \quad (3)$$

其中 $\mathbf{R}_Z(\frac{\pi}{2})$ 表示沿 Z 轴旋转 90 度得到的旋转矩阵。同时, 由于 $-\mathbf{E}$ 和 \mathbf{E} 等价, 所以对任意一个 \mathbf{t} 或 \mathbf{R} 取负号, 也会得到同样的结果。因此, 从 \mathbf{E} 分解到 \mathbf{t}, \mathbf{R} 时, 一共存在四个可能的解。请打印这四个可能的 \mathbf{R}, \mathbf{t} 。

提示: 用 AngleAxis 或 Sophus::SO3 计算 $\mathbf{R}_Z(\frac{\pi}{2})$ 。

注: 实际当中, 可以利用深度值判断哪个解是真正的解, 不过本题不作要求, 只需打印四个可能的解即可。同时, 你也可以验证 $\mathbf{t}^\wedge \mathbf{R}$ 应该与 \mathbf{E} 只差一个乘法因子, 并且与书上的实验结果亦只差一个乘法因子。

解:

解题思路已由题中给出, 程序运行结果如下所示, 可以看出 SVD 分解结果正确, $\mathbf{t}^\wedge \mathbf{R}$ 乘上一个乘法因子 s 后与 \mathbf{E} 几乎一致。

```
E2Rt X
Starting /home/vance/slam_ws/slambook/class/05/L5/code/build-qt/E2Rt...
U*D*V = -0.0203619 -0.400711 -0.0332407
  0.393927 -0.035064 0.585711
-0.00678849 -0.581543 -0.0143826
R1 = -0.365887 -0.0584576 0.928822
-0.00287462 0.998092 0.0616848
  0.930655 -0.0198996 0.365356
R2 = -0.998596 0.0516992 -0.0115267
-0.0513961 -0.99836 -0.0252005
  0.0128107 0.0245727 -0.999616
t1 = -0.581301 -0.0231206 0.401938
t2 = 0.581301 0.0231206 -0.401938
t^R = -0.0203619 -0.400711 -0.0332407
  0.393927 -0.035064 0.585711
-0.00678849 -0.581543 -0.0143826
s * t^R = -0.0203619 -0.400711 -0.0332407
  0.393927 -0.035064 0.585711
-0.00678849 -0.581543 -0.0143826
/home/vance/slam_ws/slambook/class/05/L5/code/build-qt/E2Rt exited with code 0
```

图 7 由 \mathbf{E} 恢复 \mathbf{R}, \mathbf{t} 的程序运行结果截图

自编代码部分如下所示:

```
... ..
16 #include <algorithm>
17
18 bool compare(double a, double b)
19 {
20     return a > b;
21 }
22
23 int main(int argc, char **argv) {
24
25     // 给定 Essential 矩阵
26     Matrix3d E;
27     E << -0.0203618550523477, -0.4007110038118445, -
0.03324074249824097,
28         0.3939270778216369, -0.03506401846698079,
0.5857110303721015,
29         -0.006788487241438284, -0.5815434272915686, -
0.01438258684486258;
30
31     // 待计算的 R, t
32     Matrix3d R;
33     Vector3d t;
34
35     // SVD and fix singular values
36     // START YOUR CODE HERE
37     JacobiSVD<Matrix3d> svd(E, ComputeFullV | ComputeFullU );
38     MatrixXd U = svd.matrixU(), V = svd.matrixV(), sv =
svd.singularValues();
39
40     double DD[3] = {sv(0), sv(1), sv(2)};
41     sort(DD, DD+2, compare);
42     Matrix3d D = Matrix3d::Zero();
43     D(0) = (DD[0]+DD[1])/2;
44     D(4) = D(0);
45     cout << "U*D*V = " << U*D*V.transpose() << endl;
46     // END YOUR CODE HERE
47
48     // set t1, t2, R1, R2
49     // START YOUR CODE HERE
50     Matrix3d t_wedge1;
51     Matrix3d t_wedge2;
52     Matrix3d R1;
53     Matrix3d R2;
```

```

54     Matrix3d R_z1 = AngleAxisd(M_PI/2,
Vector3d(0,0,1)).toRotationMatrix();
55     Matrix3d R_z2 = AngleAxisd(-M_PI/2,
Vector3d(0,0,1)).toRotationMatrix();
56
57     R1 = U * R_z1.transpose() * V.transpose();
58     R2 = U * R_z2.transpose() * V.transpose();
59     t_wedge1 = U * R_z1 * D * U.transpose();
60     t_wedge2 = U * R_z2 * D * U.transpose();
61     // END YOUR CODE HERE
62
63     cout << "R1 = " << R1 << endl;
64     cout << "R2 = " << R2 << endl;
65     cout << "t1 = " << Sophus::SO3d::vee(t_wedge1).transpose()
<< endl;
66     cout << "t2 = " << Sophus::SO3d::vee(t_wedge2).transpose()
<< endl;
67
68     // check t^R=E up to scale
69     Matrix3d tR = t_wedge1 * R1;
70     cout << "t^R = " << tR << endl;
71     cout << "s * t^R = " << E(0)/tR(0) * tR << endl;
72
73     return 0;
74 }
... ..

```

4 用 G-N 实现 Bundle Adjustment (3 分, 约 2 小时)

Bundle Adjustment 并不神秘, 它仅是一个目标函数为重投影误差的最小二乘。我们演示了 Bundle Adjustment 可以由 Ceres 和 g2o 实现, 并可用于 PnP 当中的位姿估计。本题, 你需要自己书写一个高斯牛顿法, 实现用 Bundle Adjustment 优化位姿的功能, 求出相机位姿。严格来说, 这是 Bundle Adjustment 的一部分, 因为我们仅考虑了位姿, 没有考虑点的更新。完整的 BA 需要用到矩阵的稀疏性, 我们留到第七节课介绍。

假设一组点的 3D 坐标为 $\mathbf{P} = \{\mathbf{p}_i\}$, 它们在相机中的坐标为 $\mathbf{U} = \{\mathbf{u}_i\}$, $\forall i = 1, \dots, n$ 。在文件 p3d.txt 和 p2d.txt 中给出了这两组点的值。同时, 设待估计的位姿为 $\mathbf{T} \in \text{SE}(3)$, 内参矩阵为:

$$\mathbf{K} = \begin{bmatrix} 520.9 & 0 & 325.1 \\ 0 & 521.0 & 249.7 \\ 0 & 0 & 1 \end{bmatrix}.$$

请你根据上述条件, 用 G-N 法求出最优位姿, 初始估计为 $\mathbf{T}_0 = \mathbf{I}$ 。程序 GN-BA.cpp 文件提供了大致的框架, 请填写剩下的内容。

在书写程序过程中, 回答下列问题:

1. 如何定义重投影误差?
2. 该误差关于自变量的雅可比矩阵是什么?
3. 解出更新量之后, 如何更新至之前的估计上?

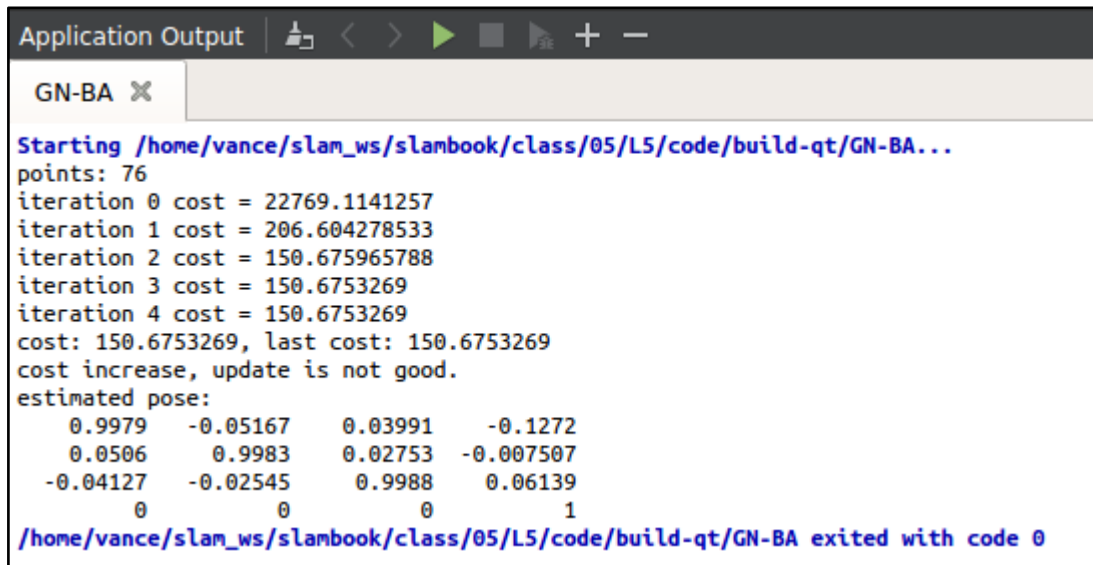
作为验证, 最后估计得到的位姿应该接近:

$$\mathbf{T}^* = \begin{bmatrix} 0.9978 & -0.0517 & 0.0399 & -0.1272 \\ 0.0506 & 0.9983 & 0.0274 & -0.007 \\ -0.0412 & -0.0253 & 0.9977 & 0.0617 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

这和书中使用 g2o 优化的结果很接近^[1]。

答:

运行结果截图如下所示:



```
Application Output | [Icons]
GN-BA x
Starting /home/vance/slam_ws/slambook/class/05/L5/code/build-qt/GN-BA...
points: 76
iteration 0 cost = 22769.1141257
iteration 1 cost = 206.604278533
iteration 2 cost = 150.675965788
iteration 3 cost = 150.6753269
iteration 4 cost = 150.6753269
cost: 150.6753269, last cost: 150.6753269
cost increase, update is not good.
estimated pose:
    0.9979    -0.05167    0.03991    -0.1272
    0.0506     0.9983     0.02753    -0.007507
   -0.04127   -0.02545     0.9988     0.06139
         0         0         0         1
/home/vance/slam_ws/slambook/class/05/L5/code/build-qt/GN-BA exited with code 0
```

图 8 BA 优化的结果截图

1. 重投影误差是将像素坐标（观测值，这里即 `p2d.txt` 中的数据）与 3D 点（这里即 `p3d.txt` 中的数据）按当前位姿（这里即每次迭代更新的 T ）估计出的投影相比较得到的误差。在程序中的第 62 行定义了重投影误差（这里是一个二维向量），在第 76 行对其进行了赋值。

2. 误差关于自变量的雅克比矩阵，可以参照《视觉 SLAM 十四讲》中公式(7.36) 及 (7.45)，是误差函数（这里是重投影误差 e ）关于自变量增量（这里是相机位姿 T ）的一阶变化关系。

3. 关于更新量的更新问题，由于更新量计算出来是一个 6 维向量，即李代数的左乘扰动，将其转换到李群上进行左乘即可更新位姿。代码实现位于第 124 行。

因程序代码过长，此处不再显示，详见附件“GN-BA.cpp”文档。

5 * 用 ICP 实现轨迹对齐 (2 分, 约 2 小时)

在实际当中, 我们经常需要比较两条轨迹之间的误差。第三节课习题中, 你已经完成了两条轨迹之间的 RMSE 误差计算。但是, 由于 ground-truth 轨迹与相机轨迹很可能不在一个参考系中, 它们得到的轨迹并不能直接比较。这时, 我们可以用 ICP 来计算两条轨迹之间的相对旋转与平移, 从而估计出两个参考系之间的差异。

设真实轨迹为 \mathbf{T}_g , 估计轨迹为 \mathbf{T}_e , 二者皆以 \mathbf{T}_{WC} 格式存储。但是真实轨迹的坐标原点定义于外部某参考系中 (取决于真实轨迹的采集方式, 如 Vicon 系统可能以某摄像头中心为参考系, 见图 8), 而估计轨迹则以相机出发点为参考系 (在视觉 SLAM 中很常见)。由于这个原因, 理论上的真实轨迹点与估计轨迹点应满足:

$$\mathbf{T}_{g,i} = \mathbf{T}_{ge} \mathbf{T}_{e,i} \quad (4)$$

其中 i 表示轨迹中的第 i 条记录, $\mathbf{T}_{ge} \in \text{SE}(3)$ 为两个坐标系之间的变换矩阵, 该矩阵在整个轨迹中保持不变。 \mathbf{T}_{ge} 可以通过两条轨迹数据估计得到, 但方法可能有若干种:

1. 认为初始化时两个坐标系的差异就是 \mathbf{T}_{ge} , 即:

$$\mathbf{T}_{ge} = \mathbf{T}_{g,1} \mathbf{T}_{e,1}^{-1}. \quad (5)$$

2. 在整条轨迹上利用最小二乘计算 \mathbf{T}_{ge} :

$$\mathbf{T}_{ge} = \arg \min_{\mathbf{T}_{ge}} \sum_{i=1}^n \left\| \log (\mathbf{T}_{g,i}^{-1} \mathbf{T}_{ge} \mathbf{T}_{e,i})^\vee \right\|_2. \quad (6)$$

3. 把两条轨迹的平移部分看作点集, 然后求点集之间的 ICP, 得到两组点之间的变换。

其中第三种也是实践中用的最广的一种。现在请你书写 ICP 程序, 估计两条轨迹之间的差异。轨迹文件在 compare.txt 文件中, 格式为:

$$\text{time}_e, \mathbf{t}_e, \mathbf{q}_e, \text{time}_g, \mathbf{t}_g, \mathbf{q}_g,$$

其中 \mathbf{t} 表示平移, \mathbf{q} 表示单位四元数。请计算两条轨迹之间的变换, 然后将它们统一到一个参考系, 并画在 pangolin 中。轨迹的格式与先前相同, 即以时间, 平移, 旋转四元数方式存储。

本题不提供代码框架, 你可以利用之前的作业完成本题。图 9 显示了对准前与对准后的两条轨迹。

解:

因程序代码过长, 此处不再显示, 详见附件 “ICP_trajectory.cpp” 文档。

程序思路如下:

- 1) 读取两个文档的数据;
- 2) 用 SVD 求解 ICP;
- 3) 依照求解出变换关系配准位姿;
- 4) 画图;

运行结果截图如下:

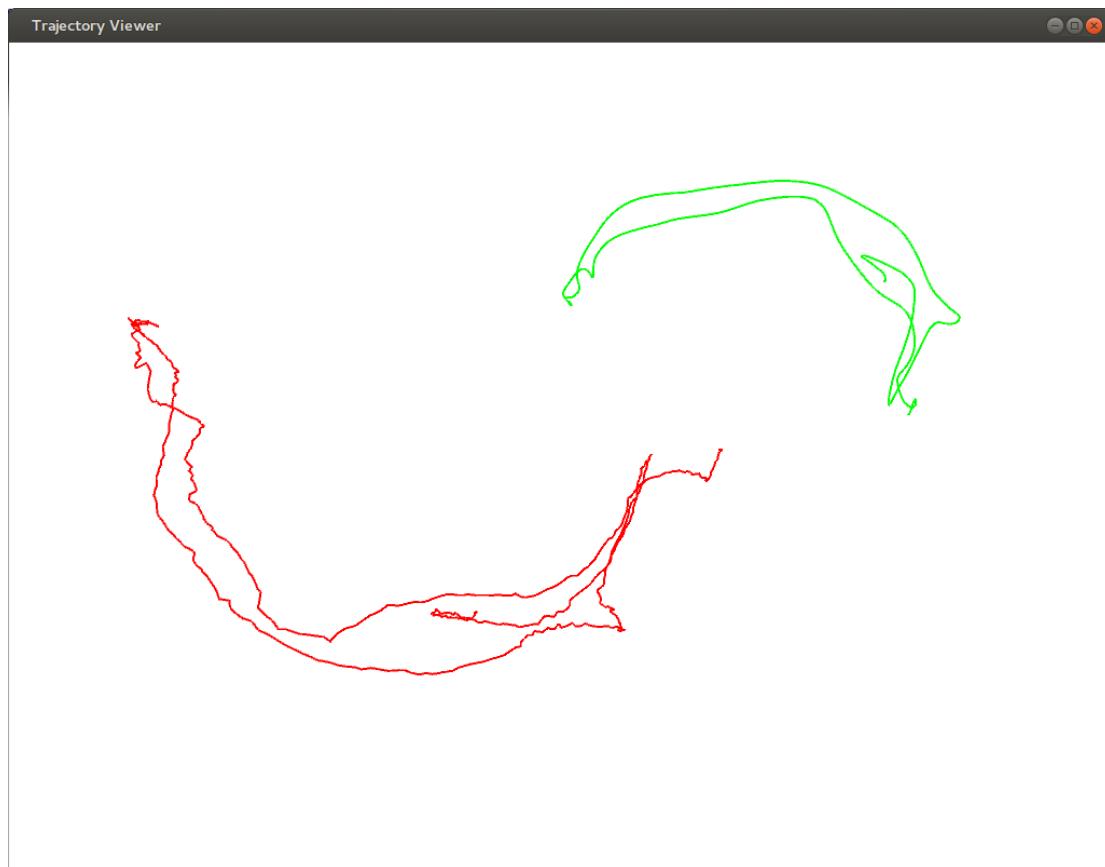


图 9 两轨迹对齐前的相对位置（红：groundtruth，绿：estimate）



图 10 用 ICP 对齐轨迹后的效果（红：groundtruth，绿：estimate）