Jindong_Shi
2021-10-23

第五节课习题

2. ORB 特征点

2.1 ORB 提取

源文件 computeORB.cpp 位于文件夹 computeORB 内。

对两个图像进行 FAST 关键点提取，再计算每个关键点的旋转部分。部分代码如下:

```
// compute the angle
void computeAngle(const cv::Mat &image, vector<cv::KeyPoint> &keypoints) {
  int half_patch_size = 8;
  for (auto &kp : keypoints) {
    // START YOUR CODE HERE (~7 lines)
    // 提取关键点的坐标
    int x = cvRound(kp.pt.x);                    // 这里使用cvRound对关键点的坐标进行四舍五入，转换为整数
    int y = cvRound(kp.pt.y);
    // 对关键点的位置做边界检查，如果该关键点跑出图像外，就跳到下一次循环，处理下一个关键点
    if( ((x - half_patch_size) < 0) ||  ((x + half_patch_size) > image.cols) || ((y-half_patch_size) < 0) || ((y + half_patch_size) > image.rows) )
        continue;

    float m01 = 0;
    float m10 = 0;
    for(int i=-half_patch_size; i<half_patch_size; i++)                //行的递增
        for(int j=-half_patch_size; j<half_patch_size; j++) {          //列的递增
            int v = y + i;
            int u = x + j;
            m01 += i * (float)image.at<uchar>(v, u);                  // !!!注意: image.at<uchar>(v, u) 内的参数 行在前, 列在后
            m10 += j * (float)image.at<uchar>(v, u);
        }
    // 计算关键点的旋转角
    kp.angle = atan2(m01, m10) * 180.0f / pi;                          // 因为 m01 和 m10 均为正数, 故在这里 atan 和 atan2 没有差异
    // kp.angle = atan(m01 / m10) * 180.0f / pi;
    // END YOUR CODE HERE
  }

  return;
}
```

第一个图像提取的关键点如下:

2.2 ORB 描述

计算各个关键点的带旋转的 BRIEF 描述子。部分代码如下:

```
// compute the descriptor
void computeORBDesc(const cv::Mat &image, vector<cv::KeyPoint> &keypoints,
                    vector<DescType> &desc) {
  for (auto &kp : keypoints) {
    DescType d(256, false);                                   // 初始化256个初始值为false的元素
    for (int i = 0; i < 256; i++) {
      // START YOUR CODE HERE (~7 lines)
        // 取得 p 和 q相对于关键点的相对坐标
        int delta_up = ORB_pattern[4*i];
        int delta_vp = ORB_pattern[4*i + 1];
        int delta_uq = ORB_pattern[4*i + 2];
        int delta_vq = ORB_pattern[4*i + 3];

        float theta = kp.angle * pi / 180.0f;                 // cos和sin输入的是弧度而不是角度，这里将theta转化为弧度
        // 利用关键点的旋转角对 p 和 q 进行旋转
        cv::Point2f delta_p((float)(cos(theta) * delta_up - sin(theta) * delta_vp), (float)(sin(theta) * delta_up + cos(theta) * delta_vp));
        cv::Point2f delta_q((float)(cos(theta) * delta_uq - sin(theta) * delta_vq), (float)(sin(theta) * delta_uq + cos(theta) * delta_vq));
        // 计算 p 和 q 在图像中的绝对坐标
        cv::Point2f p(kp.pt + delta_p);
        cv::Point2f q(kp.pt + delta_q);

        // 对p, q做边界检查，如果该描述子附近取的某个任意点p, q跑出图像外，就设这个描述子为空，并跳出当前for循环，计算下一个关键点
        if(p.x < 0 || p.x > (image.cols-1) || p.y < 0 || p.y > (image.rows-1) || q.x < 0 || q.x > (image.cols-1) || q.y < 0 || q.y > (image.rows-1) ){
            d.clear();                                        // d 的类型为 vector<bool>, 对vector进行清零
            break;
        }
       // 计算 256位 BRIEF描述子, 由 0 和 1 组成
       d[i] = image.at<uchar>(p) > image.at<uchar>(q) ? false : true;
      // END YOUR CODE HERE
    }
    desc.push_back(d);
  }
```

2.3 暴力匹配

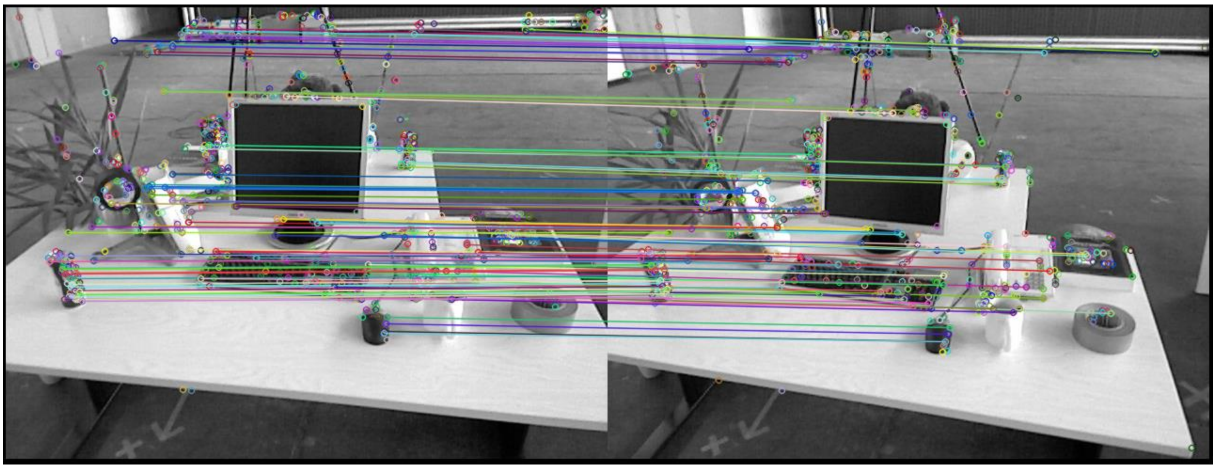
对特征点进行暴力匹配，部分代码如下:

```
// brute-force matching
void bfMatch(const vector<DescType> &desc1, const vector<DescType> &desc2,
             vector<cv::DMatch> &matches) {
  int d_max = 50;                   // d_max 为 两个特征点认作匹配的距离阈值

  // START YOUR CODE HERE (~12 lines)
  // find matches between desc1 and desc2.
  for(int i = 0; i < desc1.size(); i++){
      if(desc1[i].empty())
          continue;
      DescType d1 = desc1[i];
      int d_min = 256;                // d_min 为 最短汉明距离
      int d2_match_index = -1;        // d2_match_index 为 d1 在 desc2 中匹配的 特征点的序号

      for(int j = 0; j < desc2.size(); j++){
         if(desc2[j].empty())
             continue;
         int d = 0;                  // d 为 汉明距离，并在这里初始化
         DescType d2 = desc2[j];
         for(int k=0; k<d2.size(); k++){
             if(d1[k]!= d2[k])
                 d += 1;
         }
         if(d < d_min){
             d_min = d;
             d2_match_index = j;
         }
      }
      // 如果计算得到的匹配点的汉明距离小于等于d_max, 就被认作匹配
      if(d_min <= d_max){
          matches.push_back(cv::DMatch(i, d2_match_index, float(d_min)));
          // DMath定义: CV_WRAP DMatch(int _queryIdx, int _trainIdx, float _distance);
      }
  }
  // END YOUR CODE HERE
```

匹配结果如下:



2.4 多线程 ORB

C++17 标准引入了 STL 并行算法库，仅仅在原有的 STL 算法中添加一个处理策略参数 std::execution::par，就可以让其具备并行计算的能力[1]。GCC 9.1.0 以后的版本支持 C++ 17 STL 并行算法库，但需要安装 Intel TBB 库。

TBB 库安装失败，故未能运行并行运算。

回答问题:

1. 因为 ORB 采用的 BRIEF 描述子是二进制描述子
2. 取更大的阈值会得到更多的匹配点，但同时误匹配的概率也更高; 取更小的阈值，得到匹配点更少，误匹配概率更低
3. 暴力匹配用时如下:

   ```
   方法 bf match 平均调用时间/次数: 14577.7/1 毫秒.
   matches: 98
   ```

   为了减少计算量可以使用快速近似最邻点(FLANN)算法，该算法只会尝试去匹配特征点附近局部区域的特征点。
4. 单线程计算角点方向和描述子用时如下:

   ```
   jindong@jindong-virtual-machine:~/SLAM/Chap5/L5_code/ComputeORB/build$ ./computeORB
   keypoints: 638
   方法 compute angle 平均调用时间/次数: 2.9823/1 毫秒.
   bad/total: 44/638
   方法 compute orb descriptor 平均调用时间/次数: 74.9353/1 毫秒.
   keypoints: 595
   bad/total: 7/595
   ```

3. 从 E 恢复 R, t

源文件 E2Rt.cpp 位于文件夹 E2Rt 内，四个可能的 R, t 如下:

```
jindong@jindong-virtual-machine:~/SLAM/Chap5/L5_code/E2Rt/build$ ./E2Rt
sigma =
0.707107        0        0
       0 0.707107        0
       0        0        0
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
E =
 -0.0203619   -0.400711  -0.0332407
   0.393927   -0.035064    0.585711
-0.00678849   -0.581543  -0.0143826
```

我们可以看到计算得到的 t^R 与这里给出的 E 相等。

4. 用 G-N 实现 Bundle Adjustment 中的位姿估计

4.1 如何定义重投影误差

重投影误差就是将观测到的像素坐标和 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:

$$e_i = u_i - \frac{1}{s_i} K\exp(\xi^\wedge)P_i$$

我们将所有观测点的重投影误差求和，构建最小二乘问题:

$$\xi^* = arg\min_{\xi} \frac{1}{2}\sum_{i=1}^{n}\left\|u_i - \frac{1}{s_i}K\exp(\xi^\wedge)P_i\right\|_2^2$$

注意这里P_i为齐次坐标，按照位姿估计的空间点的坐标P'应当取前三维:

$$P' = (\exp(\xi^\wedge)P_i)_{1:3} = [X', Y', Z']^T$$

4.2 误差关于自变量的雅可比矩阵是什么?

为了通过 G-N, L-M 等优化算法进行求解，我们需要知道每个误差项关于优化变量的导数J, 对误差项进行线性化:

$$e(\xi + \Delta\xi) = e(\xi) + J\Delta\xi$$

误差项e对优化变量ξ的求导，可以根据求导链式法则，先求e对投影点P'的导数:

$$\frac{\partial e}{\partial P'} = -\begin{bmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} \end{bmatrix}$$

再求投影点P'对位姿T的求导, 我们这里对T左乘一个扰动量$\delta\xi$:

$$\begin{aligned}
\frac{\partial(TP)}{\partial\delta\xi} &= \lim_{\delta\xi\to 0}\frac{\exp(\delta\xi^\wedge)\exp(\xi^\wedge)P - \exp(\xi^\wedge)P}{\delta\xi} \\
&= \lim_{\delta\xi\to 0}\frac{(I+\delta\xi^\wedge)\exp(\xi^\wedge)P - \exp(\xi^\wedge)P}{\delta\xi} \\
&= \lim_{\delta\xi\to 0}\frac{\delta\xi^\wedge\exp(\xi^\wedge)P}{\delta\xi} \\
&= \lim_{\delta\xi\to 0}\frac{\begin{bmatrix}\delta\phi^\wedge & \delta\rho \\ 0^T & 0\end{bmatrix}\begin{bmatrix}RP+t \\ 1\end{bmatrix}}{\delta\xi} \\
&= \lim_{\delta\xi\to 0}\frac{\begin{bmatrix}\delta\phi^\wedge(RP+t)+\delta\rho \\ 0\end{bmatrix}}{\delta\xi} \\
&= \begin{bmatrix} I & -(RP+t)^\wedge \\ 0^T & 0^T \end{bmatrix} \\
&= \begin{bmatrix} I & -P'^\wedge \\ 0^T & 0^T \end{bmatrix} \triangleq (TP)^\odot
\end{aligned}$$

我们取出前三维:

$$\frac{\partial P'}{\partial \delta\xi} = [I, \ -P'^{\wedge}] = \begin{bmatrix} 1 & 0 & 0 & 0 & Z' & -Y' \\ 0 & 1 & 0 & -Z' & 0 & X' \\ 0 & 0 & 1 & Y' & -X' & 0 \end{bmatrix}$$

将$\frac{\partial e}{\partial P'}$和$\frac{\partial P'}{\partial \delta\xi}$相乘得到 2x6 的雅可比矩阵:

$$J = \frac{\partial e}{\partial \delta\xi} = -\begin{pmatrix} \frac{f_x}{Z'} & 0 & -\frac{f_x X'}{Z'^2} & -\frac{f_x X' Y'}{Z'^2} & f_x + \frac{f_x X'^2}{Z'^2} & -\frac{f_x Y'}{Z'} \\ 0 & \frac{f_y}{Z'} & -\frac{f_y Y'}{Z'^2} & -f_y - \frac{f_y Y'^2}{Z'^2} & \frac{f_y X' Y'}{Z'^2} & \frac{f_y X'}{Z'} \end{pmatrix}$$

4.3 解出更新量之后，如何更新至之前的估计上?

我们有以下循环过程:

对于第k次迭代, 我们需要寻找下降矢量$\Delta\xi_k$，使得$\|e(\xi + \Delta\xi)\|_2^2$达到最小, 并将更新量加到之前的估计位姿上。

1. 对于所有 3D 坐标点 P_i:
   a. 利用当前估计的位姿变换矩阵 T_k 计算 3D 坐标点 P_i 在相机坐标系下的空间点估计坐标$P_i' = T_k P_i$, 这里需将 3x1 的坐标P_i转换为 4x1 的齐次坐标。
   b. 然后可以利用P_i'计算该 3D 坐标点 P_i的估计投影位置 $u_i' = \frac{1}{s} K P_i'$，其中$s$为投影$KP_i'$的深度。
   c. 计算该 3D 坐标点的重投影误差为: $e_i = u_i - u_i'$。
   d. 求出当前 3D 坐标点的雅可比矩阵J_i, 从而计算矩阵$H_i = J_i^T J_i$, $b_i = -J_i^T e_i$
2. 将第k次迭代内所有点P_i计算得到的矩阵H_i和b_i进行累加，求出该次迭代最终的矩阵H和b。对误差e_i进行累加得到该次迭代的误差和: $cost_k = \frac{1}{2}\sum_i e_i^T e_i$
3. 求解增量方程: $H\Delta\xi_k = g$, 其中$H = J(\xi_k)^T J(\xi_k)$, $g = -J(\xi_k)^T e(\xi_k)$
4. 将更新量加到之前的估计上: $T_{k+1} = \exp(\Delta\xi^{\wedge}) T_k$

若 $\Delta\xi_k$ 足够小，或者该次迭代累计得到的误差和 $cost_k$ 小于上一次，那么停止迭代。

源文件 GN_BA.cpp 位于文件夹 GN_BA 内，计算结果如下:

```
jindong@jindong-virtual-machine:~/SLAM/Chap5/L5_code/GN_BA/build$ ./GN_BA
points: 76
iteration 0 cost=64718971.15893
iteration 1 cost=1215944.4619731
iteration 2 cost=12158.238860397
iteration 3 cost=12150.675364918
iteration 4 cost=12150.675326948
cost: 150.675326951, last cost: 150.675326948
estimated pose:
  0.997866179253  -0.051672715387  0.0399126394478  -0.127226355864
 0.0505962018322   0.998339757615  0.0275273087145 -0.0075066938777
-0.0412687855807 -0.0254491424109   0.998823922665  0.0613860934409
               0                0                0                1
```

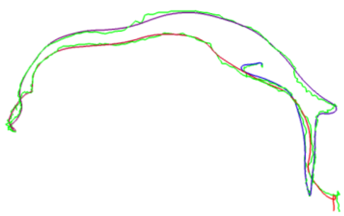
5. *用 ICP 实现轨迹的对齐

源文件 ICP_Compare.cpp 位于文件夹 ICP_Compare 内，由计算得到的变换矩阵T_{ge}，可得到从估计轨迹坐标系到真实轨迹坐标系的旋转矩阵R_{ge}和平移矩阵t_{ge}如下:

```
jindong@jindong-virtual-machine:~/SLAM/Chap5/L5_code/ICP_Compare/build$ ./ICP_Compare
Rge =
 0.923062  0.133592 -0.360707
 0.369046 -0.571969  0.732568
-0.108448 -0.809323 -0.577265
tge =
  1.5394
0.932636
 1.44618
```

两条轨迹在对齐之前如下, 其中由红到蓝的曲线为真实轨迹，绿色曲线为估计轨迹:

将两条轨迹对齐后如下, 其中由红到蓝的曲线为真实轨迹，绿色曲线由估计轨迹变换到真实轨迹的坐标系下后得到:



参考文献:

[1] Ubuntu 16.04 系统中使用 GCC 9.1 及 Intel TBB 库运行 C++17 STL 并行算法库
https://blog.csdn.net/davidhopper/article/details/98309966