2018 年第 6 次课习题答案

——Vance 吴方熠

2 LK 光流 (5 分，约 3 小时)

2.1 光流文献综述 (1 分)

我们课上演示了 Lucas-Kanade 稀疏光流，用 OpenCV 函数实现了光流法追踪特征点。实际上，光流法有很长时间的研究历史，直到现在人们还在尝试用 Deep learning 等方法对光流进行改进 [1, 2]。本题将指导你完成基于 Gauss-Newton 的金字塔光流法。首先，请阅读文献 [3] (paper 目录下提供了 pdf)，回答下列问题。

问题:

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?

答：

1. 图像配准的方法包括图片梯度下降法，差分分解法和线性回归法等，其中梯度下降法是一个标准的解决方案。对于梯度下降法来说，又因是否估计参数的累加增量，或估计的增量变换（wrap）是否与当前的变换估值相组合，而有所不同。另一个区别是算法是否在每个梯度下降步骤中是否执行了高斯牛顿法（Gauss-Newton）、牛顿法（Newton），最速下降法（steepest-descent）或列文伯格马夸尔特（Levenberg-Marquardt）近似法。

本文中以 wrap 的更新规则作为分类，将图像配准的方法分为了正向（forward）和反向（inverse）、加性（additive）和组合（compositional）两两匹配出的四种方法。

2. 在组合法（compositional）中，参数更新增量$\Delta\mathbf{p}$引起的增量变化$\mathbf{W}(\mathbf{x},\Delta\mathbf{p})$必须组合到当前的变化估计$\mathbf{W}(\mathbf{x},\mathbf{p})$中（即$\mathbf{W}(\mathbf{x},\mathbf{p}) \leftarrow \mathbf{W}(\mathbf{x},\mathbf{p}) \circ \mathbf{W}(\mathbf{x},\Delta\mathbf{p})$），而不是简单地作参数加性更新（即$\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$），故在 compositional 中需要做原始图像的 wrap.

在前面提到的更新规则里有两个要求：①这组 wrap 必须包含“幺元”；②这组 wrap 对其“compositional 运算”是封闭的。因此这组 wrap 运算必须形成一种“半群”（semi-groups），这就是其物理意义。

3. forward 的方法在每次迭代更新时，计算要追踪图像上的特征点的灰度梯度；而 inverse 的方法，可直接使用原图像中特征点的灰度梯度进行迭代，不需要在每次迭代过程中重新计算。

2.2 forward-addtive Gauss-Newton 光流的实现（1 分）

接下来我们来实现最简单的光流，即上文所说的 forward-addtive。我们先考虑单层图像的 LK 光流，然后再推广至金字塔图像。按照教材的习惯，我们把光流法建模成一个非线性优化问题，再使用 Gauss-Newton 法迭代求解。设有图像 1.png，2.png，我们在 1.png 中提取了 GFTT 角点 [4]，然后希望在 2.png 中追踪这些关键点。设两个图分别为 I_1, I_2，第一张图中提取的点集为 $\mathbf{P} = \{\mathbf{p}_i\}$，其中 $\mathbf{p}_i = [x_i, y_i]\mathrm{T}$ 为像素坐标值。考虑第 i 个点，我们希望计算 $\Delta x_i, \Delta y_i$，满足：

$$\min_{\Delta x_i, \Delta y_i} \sum_{W} \left\| I_1(x_i, y_i) - I_2(x_i + \Delta x_i, y_i + \Delta y_i) \right\|_2^2. \tag{1}$$

即最小化二者灰度差的平方，其中 $\sum_W$ 表示我们在某个窗口（Window）中求和（而不是单个像素，因为问题有两个未知量，单个像素只有一个约束，是欠定的）。实践中，取此 window 为 8×8 大小的小块，即从 $x_i - 4$ 取到 $x_i + 3$，y 坐标亦然。显然，这是一个 forward-addtive 的光流，而上述最小二乘问题可以用 Gauss-Newton 迭代求解。请回答下列问题，并根据你的回答，实现 code/optical_flow.cpp 文件中的 OpticalFlowSingleLevel 函数。

1. **从最小二乘角度来看，每个像素的误差怎么定义？**
2. **误差相对于自变量的导数如何定义？**

下面是有关实现过程中的一些提示：

1. 同上一次作业，你仍然需要去除那些提在图像边界附近的点，不然你的图像块可能越过边界。
2. 该函数称为单层的光流，下面我们要基于这个函数来实现多层的光流。在主函数中，我们对两张图像分别测试单层光流、多层光流，并与 OpenCV 结果进行对比。作为验证，正向单层光流结果应该如图 1 所示，它结果不是很好，但大部分还是对的。
3. 在光流中，关键点的坐标值通常是浮点数，但图像数据都是以整数作为下标的。之前我们直接取了浮点数的整数部分，即把小数部分归零。但是在光流中，通常的优化值都在几个像素内变化，所以我们还用浮点数的像素插值。函数 GetPixelValue 为你提供了一个**双线性插值**方法（这也是常用的图像插值法之一），你可以用它来获得浮点的像素值。

答：

1. 每个像素的误差定义为两幅图像对应点之间的**光度误差**，其中对应像素是根据对应特征点上取相同的 patch 上对应位置的像素。

2. 误差相对于自变量的导数定义为该点在 x 和 y 方向上的梯度，是一个二维向量。

正向光流效果与参考答案的对比如下图所示：

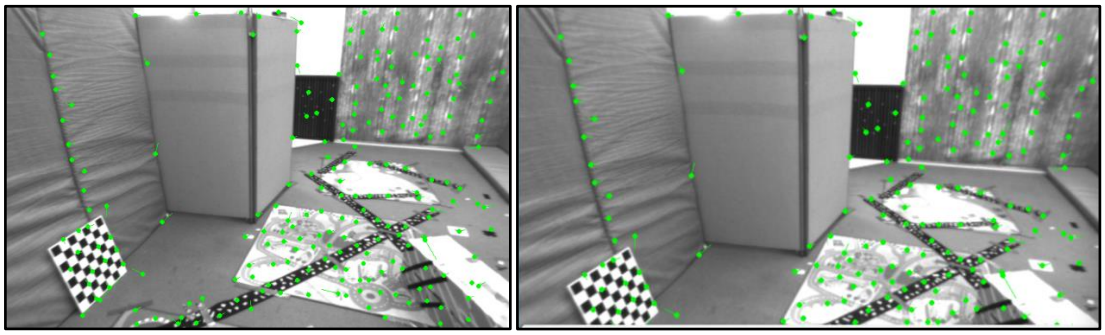

图 1 正向单层光流法效果（左）与参考答案（右）的对比

2.3 反向法（1 分）

在你实现了上述算法之后，就会发现，在迭代开始时，Gauss-Newton 的计算依赖于 I_2 在 (x_i, y_i) 处的梯度信息。然而，角点提取算法仅保证了 $I_1(x_i, y_i)$ 处是角点（可以认为角度点存在明显梯度），但对于 I_2，我们并没有办法假设 I_2 在 x_i, y_i 处亦有梯度，从而 Gauss-Newton 并不一定成立。反向的光流法（inverse）则做了一个巧妙的技巧，即用 $I_1(x_i, y_i)$ 处的梯度，替换掉原本要计算的 $I_2(x_i + \Delta x_i, y_i + \Delta y_i)$ 的梯度。这样做的好处有：

- $I_1(x_i, y_i)$ 是角点，梯度总有意义；
- $I_1(x_i, y_i)$ 处的梯度不随迭代改变，所以只需计算一次，就可以在后续的迭代中一直使用，节省了大量计算时间。

我们为 OpticalFlowSingleLevel 函数添加一个 bool inverse 参数，指定要使用正常的算法还是反向的算法。请你根据上述说明，完成反向的 LK 光流法。

解：

反向光流效果如下图所示：

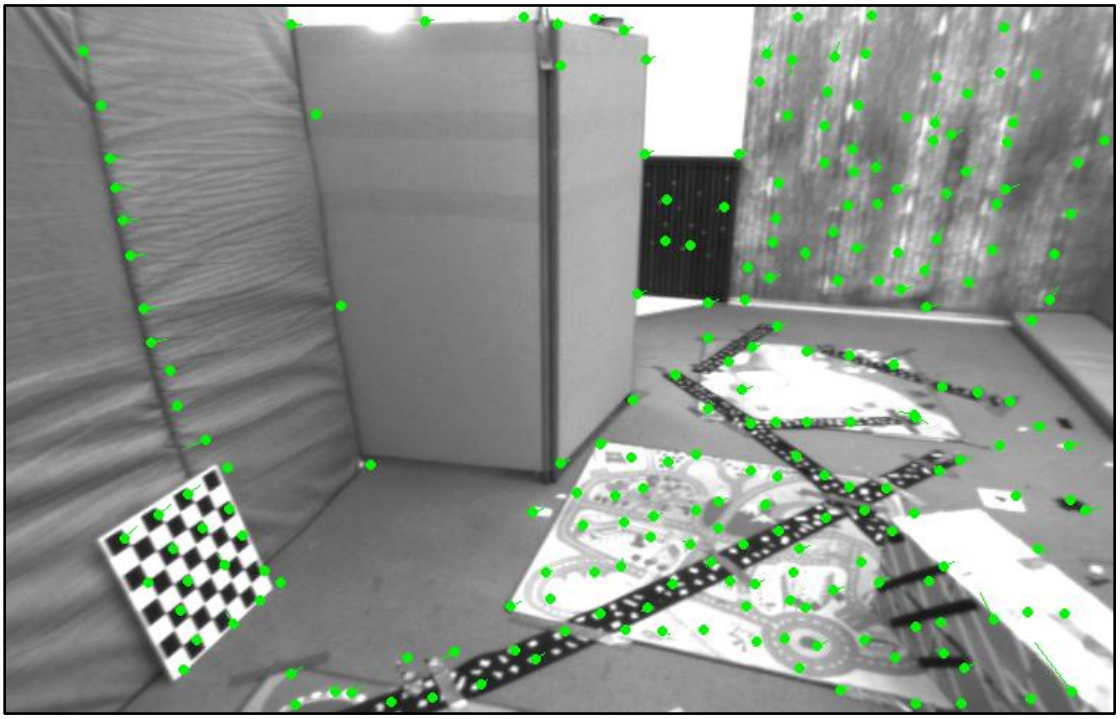

图 2 反向单层光流法效果

2.4 推广至金字塔（2 分）

通过实验，可以看出光流法通常只能估计几个像素内的误差。如果初始估计不够好，或者图像运动太大，光流法就无法得到有效的估计（不像特征点匹配那样）。但是，使用图像金字塔，可以让光流对图像运动不那么敏感。下面请你使用缩放倍率为 2，共四层的图像金字塔，实现 coarse-to-fine 的 LK 光流。函数在 OpticalFlowMultiLevel 中。

实现完成后，给出你的光流截图（正向、反向、金字塔正向、金字塔反向），可以和 OpenCV 作比较。然后回答下列问题：

1. 所谓 coarse-to-fine 是指怎样的过程?
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?

提示：你可以使用上面写的单层光流来帮助你实现多层光流。

答：

1. 由于光流法要求相机的运动是微小的，它对长距离的移动表现不够好，这时可以用 coarse-to-fine 的方法来提高其健壮性。所谓的 coarse-to-fine 的过程是指为图像构建图像金字塔，并通过在每层金字塔上进行光流来摆脱小运动的约束。如果构造了一个 n 层的金字塔，那么顶层中一个像素可以代表底层2^{n-1}个像素的距离。

2. 光流法中的金字塔用途，是为了解决其“微小运动假设”的约束，使得光流法在相机进行较大尺度运动的情况下任可以起作用。特征点法中的金字塔，是为了让特征点具有尺度不变性，加强特征点的“可识别性”和“独特性”，为后续的特征点匹配搭建一个更好的条件。

金字塔正、反向光流效果如下图所示：

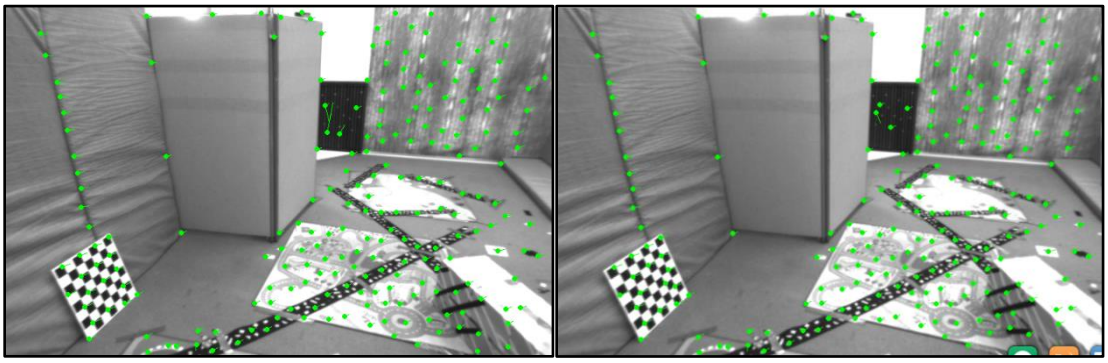

图 3 正向多层光流法效果（左）与参考答案（右）的对比

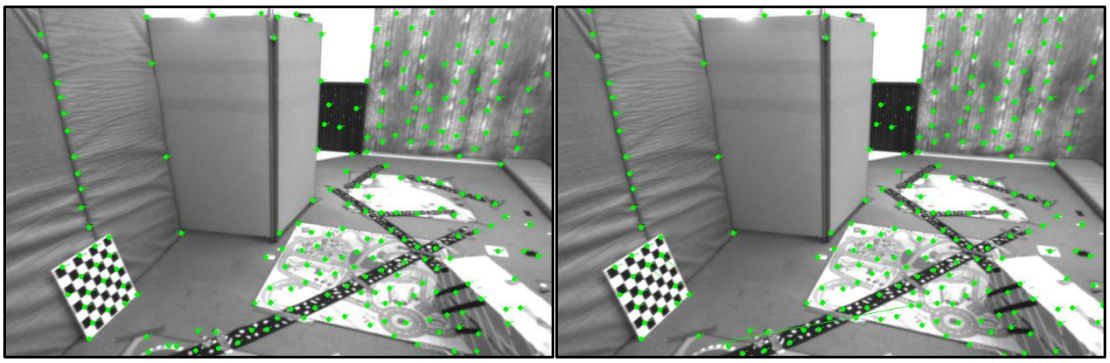

图 4 反向多层光流法效果（左）和使用 OpenCV 计算的多层光流法效果（右）

2.5 讨论

现在你已经自己实现了光流，看到了基于金字塔的 LK 光流能够与 OpenCV 达到相似的效果（甚至更好）。根据光流的结果，你可以和上讲一样，计算对极几何来估计相机运动。下面针对本次实验结果，谈谈你对下面问题的看法：

- 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？
- 图像块大小是否有明显差异？取 16x16 和 8x8 的图像块会让结果发生变化吗？
- 金字塔层数对结果有怎样的影响？缩放倍率呢？

答：

1.不太合理。在实际应用中，由于遮挡性、多光源、透明性和噪声等原因，使得光流的“灰度守恒假设条件”不能满足，所以优化两图像块的灰度之差在实际应用一般不会被对精度要求比较高的系统采用。我的解决思路：可以考虑分通道计算光流。

2.patch 的大小对多层光流法最后的结果从程序上来看没有太明显的差异，只是随着 patch 的增大，图像边界上的特征点会被逐渐剔除掉，这是因为我们程序中对 patch 超出边界的特征点选择了删除处理。patch 对单层光流法的效果影响相对较大，大的 patch 会提高单层光流的精度。图 5 是正向单层光流法在 patch 取 32x32 和 100x100 时效果。

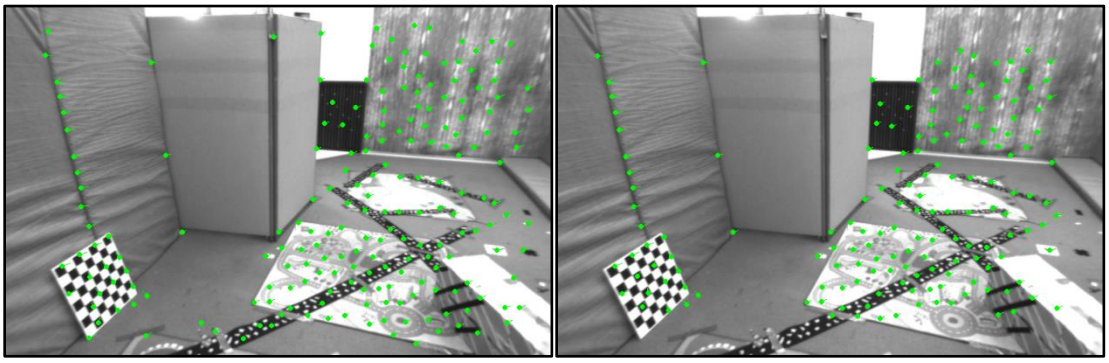

图 5 正向单层光流法 patch-32x32 效果（左）
和正向单层光流法 patch-100x100 效果（右）

但在实际应用中，一般都使用多层的光流法。OpenCV 里计算的 patch 默认尺寸为 21x21，而在本例程序中 OpenCV 计算的光流图像块仅取到 8x8 大小。patch 的大小对多层结果影响不大，但计算时间会随着 patch 增大而增加。另一方面，patch 取太大的话也会违反其“空间一致性”的假设，故实际使用时 patch 不必取的太大。

3.金字塔层数越多，缩放倍率约小，就意味着更精细的光流效果和更大的计算量。金字塔层数一般取 3-4 层即可，缩放倍率一般为 2。

3 直接法 (5 分，约 2.5 小时)

3.1 单层直接法（3 分）

我们说直接法是光流的直观拓展。在光流中，我们估计的是每个像素的平移（在 additive 的情况下）。而在直接法当中，我们最小化光流误差，来估计相机的旋转和平移（以李代数的形式）。现在我们将使用和前一个习题非常相似的做法来实现直接法，请同学体现二者之间的紧密联系。

本习题中，你将使用 Kitti 数据集中的一些图像。给定 left.png 和 disparity.png，我们知道，通过这两个图可以得到 left.png 中任意一点的 3D 信息。现在，请你使用直接法，估计图像 000001.png 至 000005.png 的相机位姿。我们称 left.png 为**参考图像**（reference，简称 ref），称 000001.png -000005.png 中任意一图为**当前图像**（current，简称 cur），如图2 所示。设待估计的目标为 $\mathbf{T}_{\text{cur,ref}}$，那么在 ref 中取一组点 $\{\mathbf{p}_i\}$，位姿可以通过最小化下面的目标函数求解：

$$\mathbf{T}_{\text{cur,ref}} = \frac{1}{N}\sum_{i=1}^{N}\sum_{W_i}\left\|I_{\text{ref}}(\mathbf{p}_i) - I_{\text{cur}}\left(\pi\left(\mathbf{T}_{\text{cur,ref}}\,\pi^{-1}(\mathbf{p}_i)\right)\right)\right\|_2^2. \tag{2}$$

其中 N 为点数，π 函数为针孔相机的投影函数 $\mathbb{R}^3 \mapsto \mathbb{R}^2$，$\pi^{-1}$ 为反投影函数，W_i 为第 i 个点周围的小窗口。同光流法，该问题可由 Gauss-Newton 函数求解。请回答下列问题，然后实现 code/direct_method.cpp 中的 DirectPoseEstimationSingleLayer 函数。

1. 该问题中的误差项是什么?
2. 误差相对于自变量的雅可比维度是多少？如何求解?
3. 窗口可以取多大？是否可以取单个点?

下面是一些实现过程中的提示：

1. 这次我们在参考图像中随机取 1000 个点，而不是取角点。请思考为何不取角点，直接法也能工作。
2. 由于相机运动，参考图像中的点可能在投影之后，跑到后续图像的外部。所以最后的目标函数要对投影在内部的点求平均误差，而不是对所有点求平均。程序中我们以 good 标记出投影在内部的点。

答：

1. 误差项是对应像素点的**光度误差**：$e = I_1(p_1) - I_2(p_2)$

2. 误差项是常量，自变量是 6 维李代数，故误差相对于自变量的雅克比维度是 6 维的，即：

$$J = -\frac{\partial I_2}{\partial u}\cdot\frac{\partial u}{\partial \delta\xi} = -\frac{\partial I_2}{\partial u}\cdot\frac{\partial u}{\partial q}\cdot\frac{\partial q}{\partial \delta\xi}$$

其中$\boldsymbol{q} = [x \quad y \quad z]^{\mathrm{T}}$为投影点在相机坐标系下的三维坐标。参考《十四讲》8.4.1 小节可知：

$$\frac{\partial I_2}{\partial u} = \begin{bmatrix}\dfrac{I_2(u+1,v) - I_2(u-1,v)}{2} & \dfrac{I_2(u,v+1) - I_2(u,v-1)}{2}\end{bmatrix} \tag{3.1}$$

$$\frac{\partial u}{\partial q} = \begin{bmatrix}\dfrac{f_x}{z} & 0 & -\dfrac{f_x x}{z^2} \\ 0 & \dfrac{f_y}{z} & -\dfrac{f_y y}{z^2}\end{bmatrix} \tag{3.2}$$

$$\frac{\partial q}{\partial \delta\xi} = [I \quad -\boldsymbol{q}^{\wedge}] \tag{3.3}$$

式（3.2）与式（3.3）相乘后得到的$\frac{\partial u}{\partial \delta\xi}$即是《十四讲》中的式（8.15）。

3. ①窗口的大小不要取得太大，一方面大窗口会让计算量增加，另一方面一个大窗口内可能会包含多个要计算的像素点，这样就产生了很多重复的计算。②不可以取单个点。由于程序初始迭代时传入的相机位姿初始值为“幺元”变换（即无变换），如果窗口仅取单个点，那么每次迭代时求出的位姿更新量均为 0，两幅图像间的匹配全部变成了相同像素位置的匹配了，见图 6.（由于 1000 个点太过密集，这里大致画出了 1/3 的匹配点）

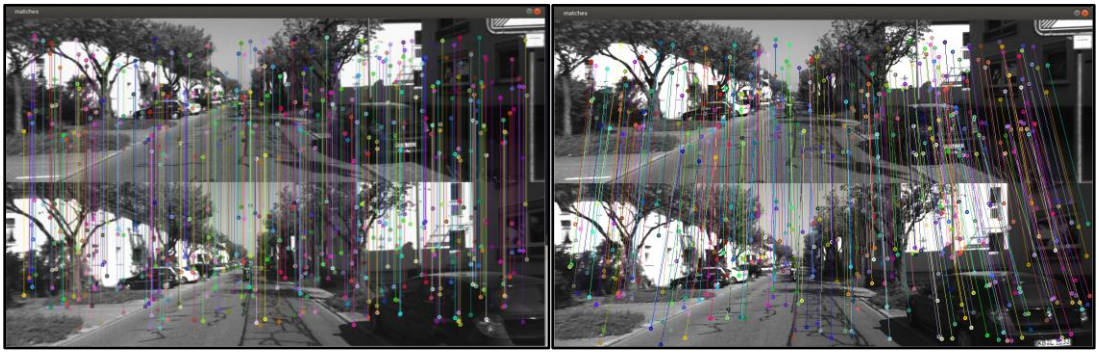

图 6 单层直接法窗口取 1 个点（左）和取 64 个点（右）时两帧间的匹配情况

4. 直接法是根据当前相机的位姿估计值来寻找对应的像素点，它可以对所有的像素进行处理，所以并不需要提取特征点也可以工作。

单层直接法的匹配情况见图 6 右侧，计算输出结果见图 7.

direct_method

```
Starting /home/vance/slam_ws/slambook/class/L6/code/build/direct_method...
cost increased in iteration 35: 13049.4 --> 13049.3
good projection: 988
Image 1 processing time cost = 16737.8 ms. T_cur_ref is
   0.999991  0.00242139  0.00337219 -0.00183998
-0.00242878    0.999995  0.00218908   0.0026696
-0.00336687 -0.00219725    0.999992    -0.72514
          0           0           0           1
cost increased in iteration 60: 29834.5 --> 29834.5
good projection: 926
Image 2 processing time cost = 16642 ms. T_cur_ref is
   0.999972  0.00136972  0.00729044  0.00737278
-0.00139808    0.999991  0.00388642 -0.00136252
-0.00728505  -0.0038965    0.999966    -1.47081
          0           0           0           1
cost increased in iteration 33: 86060.5 --> 85990
good projection: 872
Image 3 processing time cost = 9890.76 ms. T_cur_ref is
   0.999913 0.000475498   0.0132178   -0.219988
-0.000548504    0.999985  0.00552023 -0.00368286
  -0.013215   -0.005527    0.999897     -1.8877
          0           0           0           1
cost increased in iteration 9: 157974 --> 157845
good projection: 864
Image 4 processing time cost = 3579.53 ms. T_cur_ref is
    0.99986  0.00249654   0.0165647   -0.284996
-0.00258327    0.999983  0.00521636   0.0201918
 -0.0165514 -0.00525842    0.999849    -2.01654
          0           0           0           1
cost increased in iteration 68: 168582 --> 168543
good projection: 760
Image 5 processing time cost = 18509 ms. T_cur_ref is
   0.999736  0.00194354   0.0228752   -0.408856
-0.00204569    0.999988  0.00444333   0.0640692
 -0.0228663 -0.00448895    0.999728    -2.94519
          0           0           0           1
/home/vance/slam_ws/slambook/class/L6/code/build/direct_method exited with code 0
```

图 7 单层直接法计算输出结果

3.2 多层直接法（2 分）

下面，类似于光流，我们也可以把直接法以 coarse-to-fine 的过程，拓展至多层金字塔。多层金字塔的直接法允许图像在发生较大运动时仍能追踪到所有点。下面我们使用缩放倍率为 2 的四层金字塔，实现金字塔上的直接法。请实现 DirectPoseEstimationMultiLayer 函数，下面是一些提示：

1. 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，f_x, f_y, c_x, c_y 应该如何变化？
2. 根据 coarse-to-fine 的过程，上一层图像的位姿估计结果可以作为下一层图像的初始条件。
3. 在调试期间，可以画出每个点在 ref 和 cur 上的投影，看看它们是否对应。若准确对应，则说明位姿估计是准确的。

作为验证，图像 000001 和 000005 的位姿平移部分应该接近：

$$\begin{aligned} \mathbf{t}_1 &= [0.005876, -0.01024, -0.0725]^T \\ \mathbf{t}_5 &= [0.0394, -0.0592, -3.9907]^T \end{aligned} \tag{3}$$

可以看出车辆基本是笔直向前开的。

答：

当图像缩小一倍时，即$\boldsymbol{P_u}$的值缩小了一倍，由于点的三维空间坐标$\boldsymbol{P}$是不变的，根据针孔相机投影模型：

$$\boldsymbol{ZP_u = KP}$$

可知相机内参矩阵缩小了一倍，故相机的内参值也缩小了一倍。

多层直接法的计算输出结果如图 8 所示。

```
Starting /home/vance/slam_ws/slambook/class/L6/code/build/direct_method...
cost increased in iteration 12: 29888.8 --> 29888.8
good projection: 900
cost increased in iteration 2: 17509.2 --> 17470.4
good projection: 969
cost increased in iteration 4: 10973.3 --> 10972.9
good projection: 984
cost increased in iteration 6: 13049.5 --> 13049.5
good projection: 988
Image 1 processing time cost = 2.41063e+07 s. T_cur_ref is
   0.999991  0.00242148  0.00337211 -0.00183395
-0.00242887    0.999995   0.0021892  0.00266532
-0.00336679 -0.00219737    0.999992   -0.725152
          0           0           0           1
cost increased in iteration 19: 42415.7 --> 42415.7
good projection: 797
cost increased in iteration 15: 29731.4 --> 29731.4
good projection: 879
cost increased in iteration 10: 23963.6 --> 23963.6
good projection: 911
cost increased in iteration 7: 29834.5 --> 29834.5
good projection: 926
Image 2 processing time cost = 3.33536e+07 s. T_cur_ref is
   0.999972  0.00136972  0.00729048  0.00737005
-0.00139808    0.999991  0.00388638 -0.00136157
 -0.0072851 -0.00389647    0.999966    -1.47081
          0           0           0           1
cost increased in iteration 7: 57250.2 --> 57204.8
good projection: 720
cost increased in iteration 21: 44501.6 --> 44501.6
good projection: 792
cost increased in iteration 10: 41688.3 --> 41688.2
good projection: 820
cost increased in iteration 14: 50367.4 --> 50367.3
good projection: 836
```

```
Image 3 processing time cost = 2.00136e+07 s. T_cur_ref is
   0.999937  0.00161543   0.0111185  0.00709328
-0.00167204    0.999986  0.00508431  0.00377221
 -0.0111102 -0.00510258    0.999925    -2.20888
          0           0           0           1
cost increased in iteration 23: 73964.8 --> 73774.2
good projection: 627
cost increased in iteration 2: 59933.7 --> 59770.5
good projection: 709
cost increased in iteration 19: 56029.7 --> 56029.7
good projection: 743
cost increased in iteration 5: 68137.9 --> 68125
good projection: 753
Image 4 processing time cost = 1.47226e+07 s. T_cur_ref is
   0.999874 0.000357878   0.0158768  0.00924876
-0.000448983    0.999983  0.00573504  0.00292326
 -0.0158745 -0.00574144    0.999858    -2.99718
          0           0           0           1
cost increased in iteration 11: 95516.6 --> 95146.7
good projection: 560
cost increased in iteration 5: 77253.5 --> 77198.3
good projection: 626
cost increased in iteration 4: 76489.7 --> 76455.6
good projection: 654
cost increased in iteration 17: 91237.4 --> 91233.8
good projection: 671
Image 5 processing time cost = 1.15692e+07 s. T_cur_ref is
   0.999803  0.00119796   0.0198234   0.0189657
-0.00132764    0.999978    0.00653  -0.0103282
 -0.0198152 -0.00655503    0.999782    -3.79311
          0           0           0           1
/home/vance/slam_ws/slambook/class/L6/code/build/direct_method exited with code 0
```

图 8 多层直接法计算输出结果

3.3 * 延伸讨论

现在你已经实现了金字塔上的 Gauss-Newton 直接法。你可以调整实验当中的一些参数，例如图像点数、每个点周围小块的大小等等。请思考下面问题：

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？
2. 请思考上面算法哪些地方可以缓存或加速？
3. 在上述过程中，我们实际假设了哪两个 patch 不变？
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？
5. 请总结直接法相对于特征点法的异同与优缺点。

答：

1. inverse 的概念对直接法来说没有意义。光流提出的 inverse 是为了保证估计点的梯度总是很明显（因为角点处总有明显的梯度），且降低计算量。而直接法的取点并不用取特征点，所以就算是 inverse 也无法保证点有明显的梯度。

由于直接法完全依靠优化来求解相机位姿，这里的位姿由一个 6 维的李代数来表示，其位姿的更新规则采用 compositional 会比较合理。

2. 暂未发现。

3. 实际假设了两帧间对应像素点的 patch 不变，均为 8x8 的大小。

4. 直接法是根据当前相机的位姿估计值来寻找对应的像素点，它可以对所有的像素进行处理，所以并不需要提取特征点也可以工作。投影依然可以算对，投影情况见图 6 右侧。

5. 见下表：

表 1 直接法和特征点法的异同与优缺点统计表

方法	直接法	特征点法
相同点	目的都是为了找到匹配的像素点，进而求解相机运动	
不同点	● 特征点法要计算特征点和描述子，直接法不要 ● 特征点法通过最小化重投影误差来优化相机运动，直接法通过最小化光度误差 ● 特征点法智能构建稀疏地图，直接法可以构建半稠密甚至稠密地图	
优点	● 节省了计算特征点和描述子的时间 ● 只要求有像素梯度，不需要特征点 ● 可构建半稠密甚至稠密地图	● 运行稳定 ● 对光照、动态物体不敏感 ● 特征信息可做回环检测的依据
缺点	● 非凸性，对大运动效果不好 ● 单个像素没有区分度 ● 灰度值不变假设太强，容易受相机暗角、模糊、曝光等因素影响	● 计算特征点和描述子耗时 ● 忽略了特征点以外的信息 ● 在特征缺失的地方无法工作

4 * 使用光流计算视差（2 分，约 1 小时）

请注意本题为附加题。

在上一题中我们已经实现了金字塔 LK 光流。光流有很多用途，它给出了两个图像中点的对应关系，所以我们可以用光流进行位姿估计，或者计算双目的视差。回忆第四节课的习题中，我们介绍了双目可以通过视差图得出点云，但那时直接给出了视差图，而没有进行视差图的计算。现在，给定图像 left.png, right.png，请你使用上题的结果，计算 left.png 中的 GFTT 点在 right.png 中的（水平）视差，然后与 disparity.png 进行比较。这样的结果是一个稀疏角点组成的点云。请计算每个角点的水平视差，然后对比视差图比较结果。

本程序不提供代码框架，请你根据之前习题完成此内容。

由于时间有限，本题暂未解决。