2.1 光流文献综述 (1 分)

1. 按此文的分类,光流法可分为哪几类?

Additive， compositional ， forwards or inverse的四种组合

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?

因为 compositional 和 additive 的更新方式不一样。Warp 是对图像的一种变换，比如 affine 变换，homograph 变换或者 rotation 变换。这种方式的更新量通常是一个矩阵乘在另一个矩阵上，类似 T 乘以 delta T。（而不是 T 加 delta T）。

3. forward 和 inverse 有何差别?

Forward 中图像梯度放在图像 2 上面计算，每次迭代都要重新算。

Inverse 法图像梯度放在图像 1 上面计算，不随迭代而改变。

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

1. 从最小二乘角度来看,每个像素的误差怎么定义?

我们需要找到运动dx, dy使得同一个特征点在两幅图像中像素值之差最小。
则这里的误差即是图像 1 中(x,y)的像素值和图像 2 中(x+dx,y+dy)的像素值之差。

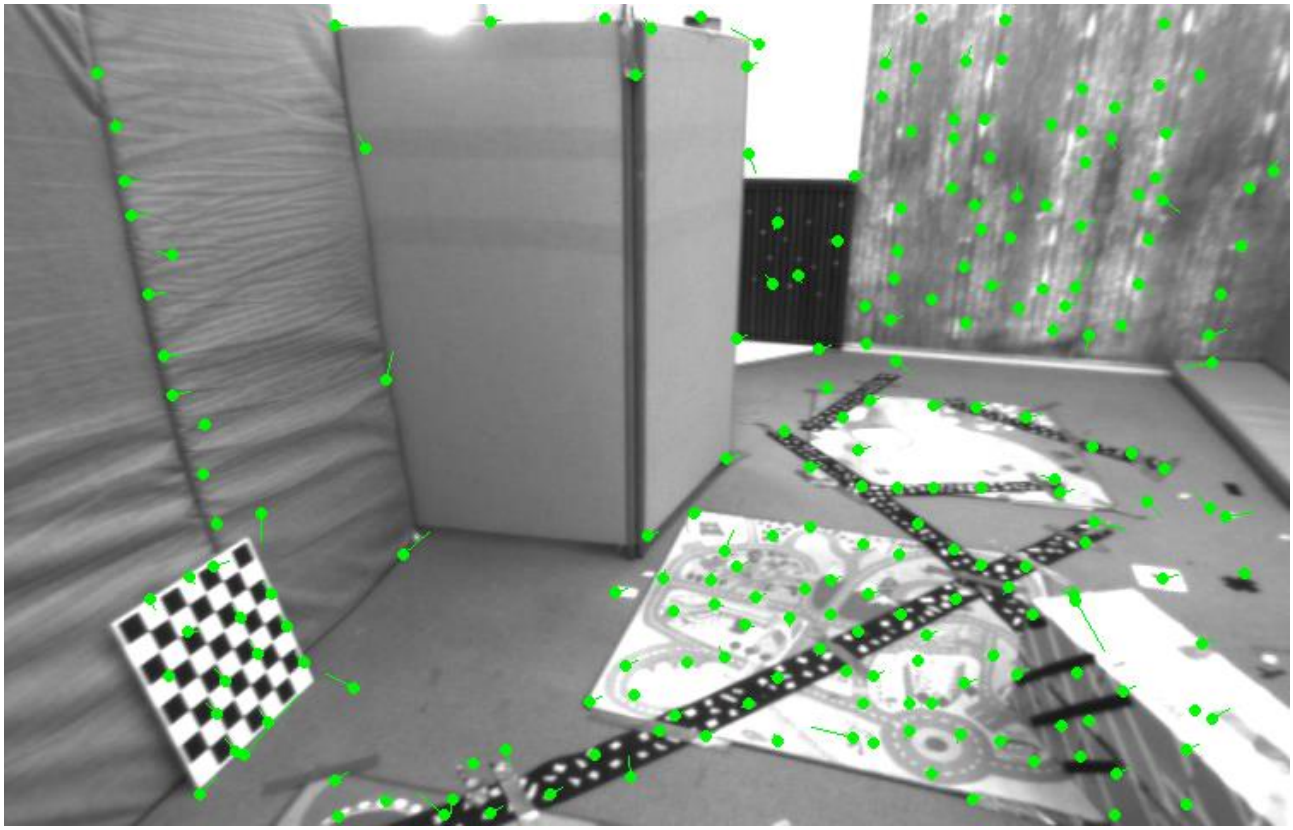
2. 误差相对于自变量的导数如何定义

像素值相对于 dx 对 dy 的偏导其实就是像素值在这两个方向上的梯度，按照课程视频介绍我们用右边一个像素减去左边一个像素值再除以二来近似 X 方向的梯度。Y 方向同理。
同时还有其他求梯度的算法比如 Laplacian。

代码

```
        // TODO START YOUR CODE HERE (~8 lines)
        double error = 0;
        error = GetPixelValue(img1, x: kp.pt.x + x, y: kp.pt.y + y) -
                GetPixelValue(img2, x: kp.pt.x + x + dx, y: kp.pt.y + y + dy);
        Eigen::Vector2d J;  // Jacobian
        if (inverse == false) {
            // Forward Jacobian
            J = -1.0 * Eigen::Vector2d( x: 0.5 * (GetPixelValue(img2, x: kp.pt.x + dx + x + 1, y: kp.pt.y + dy + y) -
                                                  GetPixelValue(img2, x: kp.pt.x + dx + x - 1, y: kp.pt.y + dy + y)),
                                        y: 0.5 * (GetPixelValue(img2, x: kp.pt.x + dx + x, y: kp.pt.y + dy + y + 1) -
                                                  GetPixelValue(img2, x: kp.pt.x + dx + x, y: kp.pt.y + dy + y - 1)));
        } else {
            // Inverse Jacobian
            // NOTE this J does not change when dx, dy is updated, so we can store it and only compute error
            J = -1.0 * Eigen::Vector2d( x: 0.5 * (GetPixelValue(img1, x: kp.pt.x + dx + x + 1, y: kp.pt.y + dy + y) -
                                                  GetPixelValue(img1, x: kp.pt.x + dx + x - 1, y: kp.pt.y + dy + y)),
                                        y: 0.5 * (GetPixelValue(img1, x: kp.pt.x + dx + x, y: kp.pt.y + dy + y + 1) -
                                                  GetPixelValue(img1, x: kp.pt.x + dx + x, y: kp.pt.y + dy + y - 1)));
        }

        // compute H, b and set cost;
        if (inverse == false || 0 == iter) {
            H += J * J.transpose();
        }
        b += -error * J;
        cost += error * error;
        // TODO END YOUR CODE HERE
    }

    // compute update
    // TODO START YOUR CODE HERE (~1 lines)
    Eigen::Vector2d update;
    update = H.ldlt().solve(b);
    // TODO END YOUR CODE HERE
```

结果，可见有一些点的运动明显不合群。



2.3 反向法(1 分)

代码见上图 inverse == true 部分。

2.4 推广至金字塔(2 分)

1. 所谓 coarse-to-fine 是指怎样的过程?

即粗到精过程，从最小的缩放图做光流，逐步到在原始图像上做光流。
因为光流法要假设相机运动较小，特征点的亮度不变特性只在局部生效，所以遇到大运动的情况需要“缩放”图像，相当于在更大的局部范围中找到 dxdy 的运动。

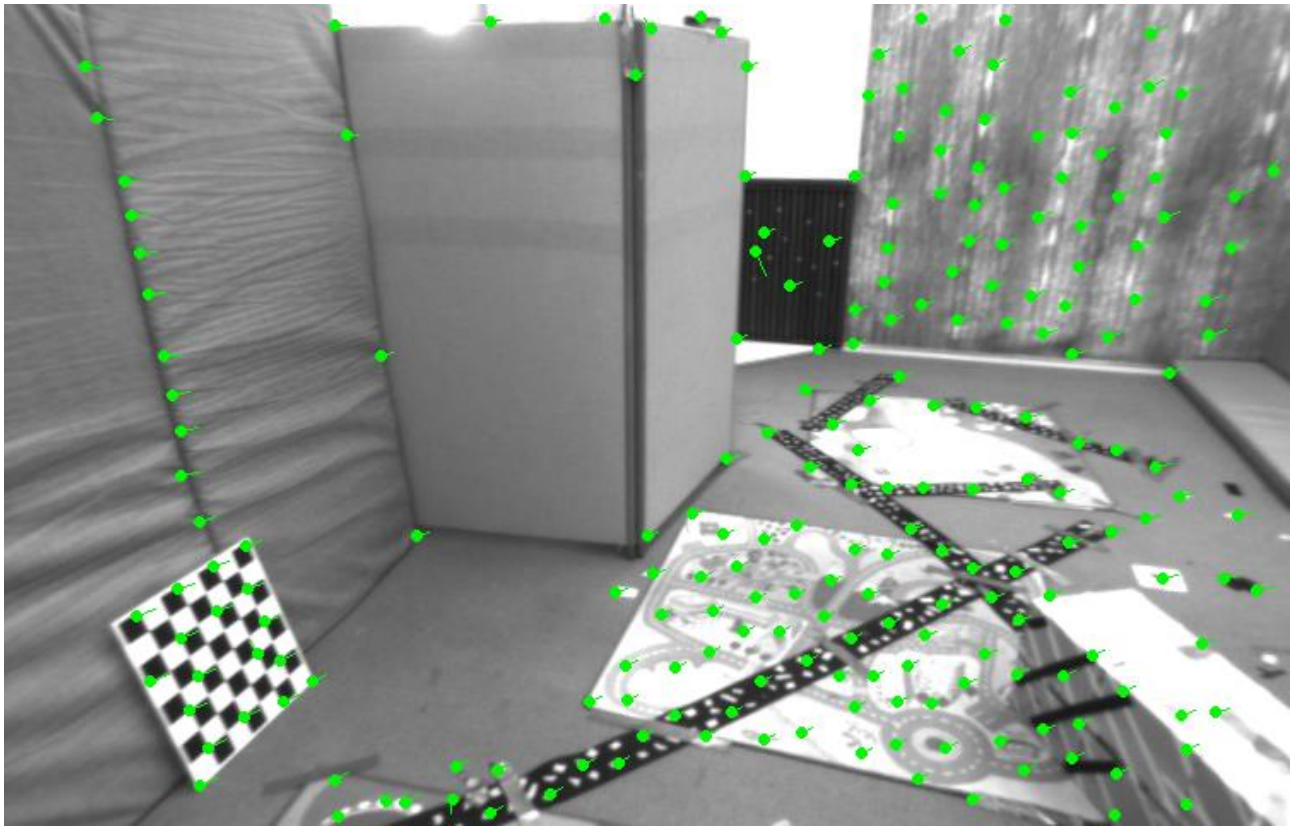
2. 光流法中的金字塔用途和特征点法中的金字塔有何差别?

光流法的金字塔是通过缩放图像应对大运动。光流法要在同一个 level 的金字塔图像中进行。
特征点法金字塔是通过缩放图像来应对尺度变化。特征点可以跨 level 寻找。
但是思想上是类似的，都是通过金字塔来把让一些利用局部信息的算法更鲁棒。

代码

```cpp
vector<Mat> pyr1, pyr2; // image pyramids
// TODO START YOUR CODE HERE (~8 lines)
for (int i = 0; i < pyramids; i++) {
    if (i == 0) {
        pyr1.push_back(img1);
        pyr2.push_back(img2);
    } else {
        Mat img1_pyr, img2_pyr;
        cv::resize( src: pyr1[i - 1], img1_pyr,
                   dsize: cv::Size( _width: pyr1[i - 1].cols * pyramid_scale,  _height: pyr1[i - 1].rows * pyramid_scale));
        cv::resize( src: pyr2[i - 1], img2_pyr,
                   dsize: cv::Size( _width: pyr2[i - 1].cols * pyramid_scale,  _height: pyr2[i - 1].rows * pyramid_scale));
        pyr1.push_back(img1_pyr);
        pyr2.push_back(img2_pyr);TODO
    }
}
// TODO END YOUR CODE HERE

// coarse-to-fine LK tracking in pyramids
// TODO START YOUR CODE HERE
vector<KeyPoint> kp1_pyr, kp2_pyr;
for (int i = 0; i < kp1.size(); i++) {
    KeyPoint thisKeyPoint = kp1[i];
    thisKeyPoint.pt = thisKeyPoint.pt * scales[pyramids - 1];
    kp1_pyr.push_back(thisKeyPoint);
    kp2_pyr.push_back(thisKeyPoint);
}

for (int level = pyramids - 1; level >= 0; level--) {
    // from coarse to fine
    success.clear();
    OpticalFlowSingleLevel(pyr1[level], pyr2[level], kp1_pyr,  &: kp2_pyr,  &: success);
    if (level > 0) {
        for (auto &kp: kp1_pyr)
            kp.pt /= pyramid_scale;
        for (auto &kp: kp2_pyr)
            kp.pt /= pyramid_scale;
    }
}

for (auto &kp: kp2_pyr)
    kp2.push_back(kp);

// TODO END YOUR CODE HERE
// don't forget to set the results into kp2
```

结果，明现多数点的运动向着一个方向。



2.5 讨论

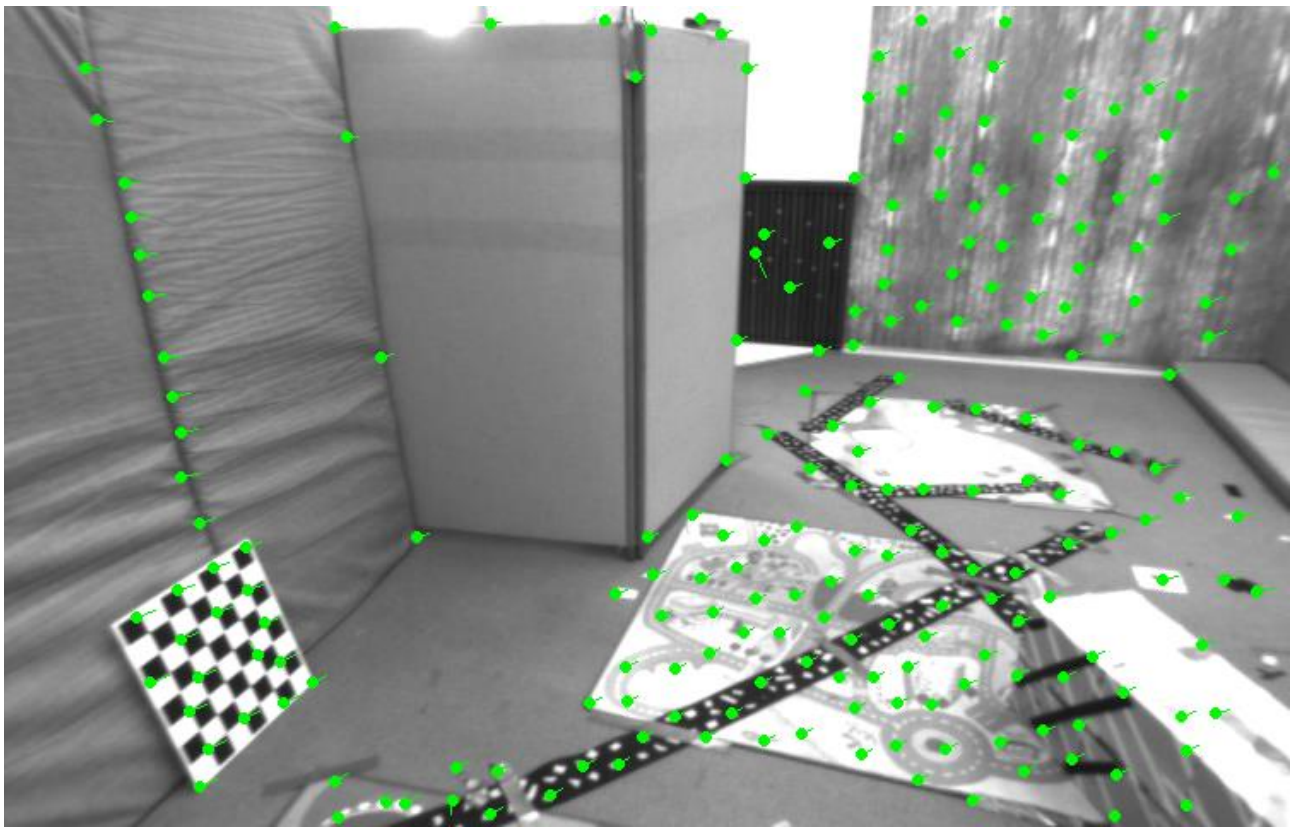
• 我们优化两个图像块的灰度之差真的合理吗?哪些时候不够合理?你有解决办法吗?

图像块的灰度之差本身就是怕只用单个特征点的像素差不够鲁棒。但是图像块也有自己不够鲁棒的时候，比如光照变化剧烈，反光之类的。可以试试除以灰度均值，得到一个去除亮度变化的结果。

• 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?

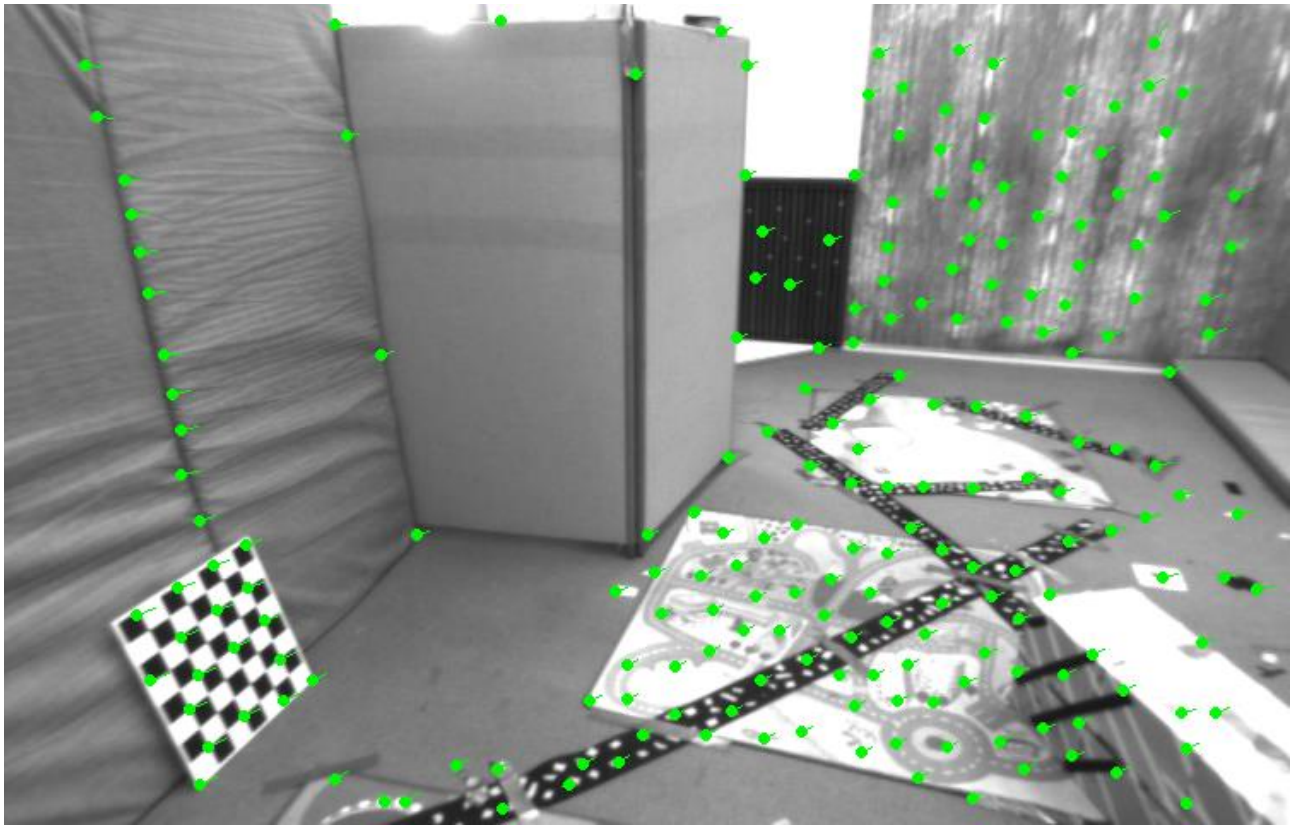
一些特殊位置的点的运动结果有差异。

下图为多层金字塔 Batch8 结果



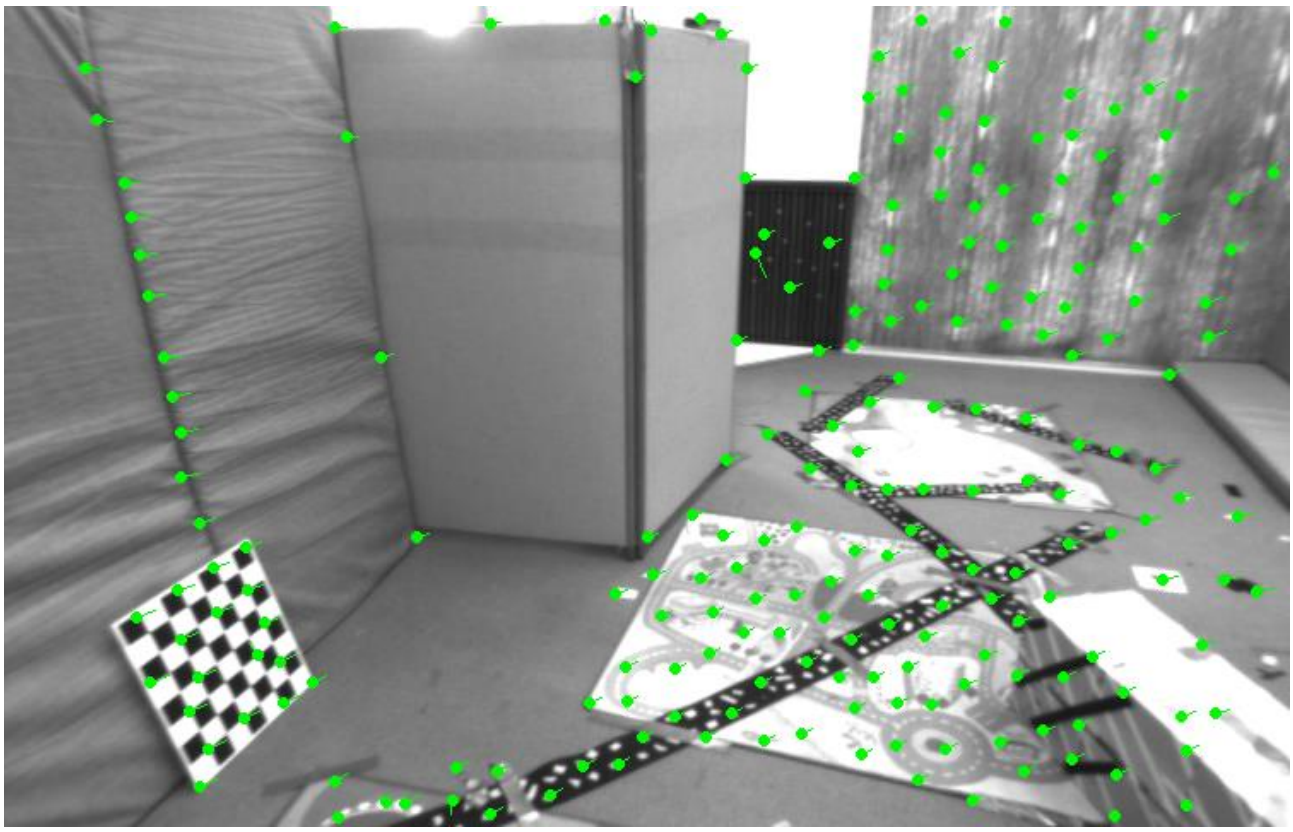
下图为多层金字塔 batch 为 16 的结果

Batch 增大之后会把更多的周围像素纳入进计算，有些特殊位置的特征点的运动结果就改变了（也可能有些特征点就匹配失败了）。比如上图中央部分那块黑色的物体上有一个特征点的方向是朝右下，batch 扩大之后那个点的方向就看起来更正确了。



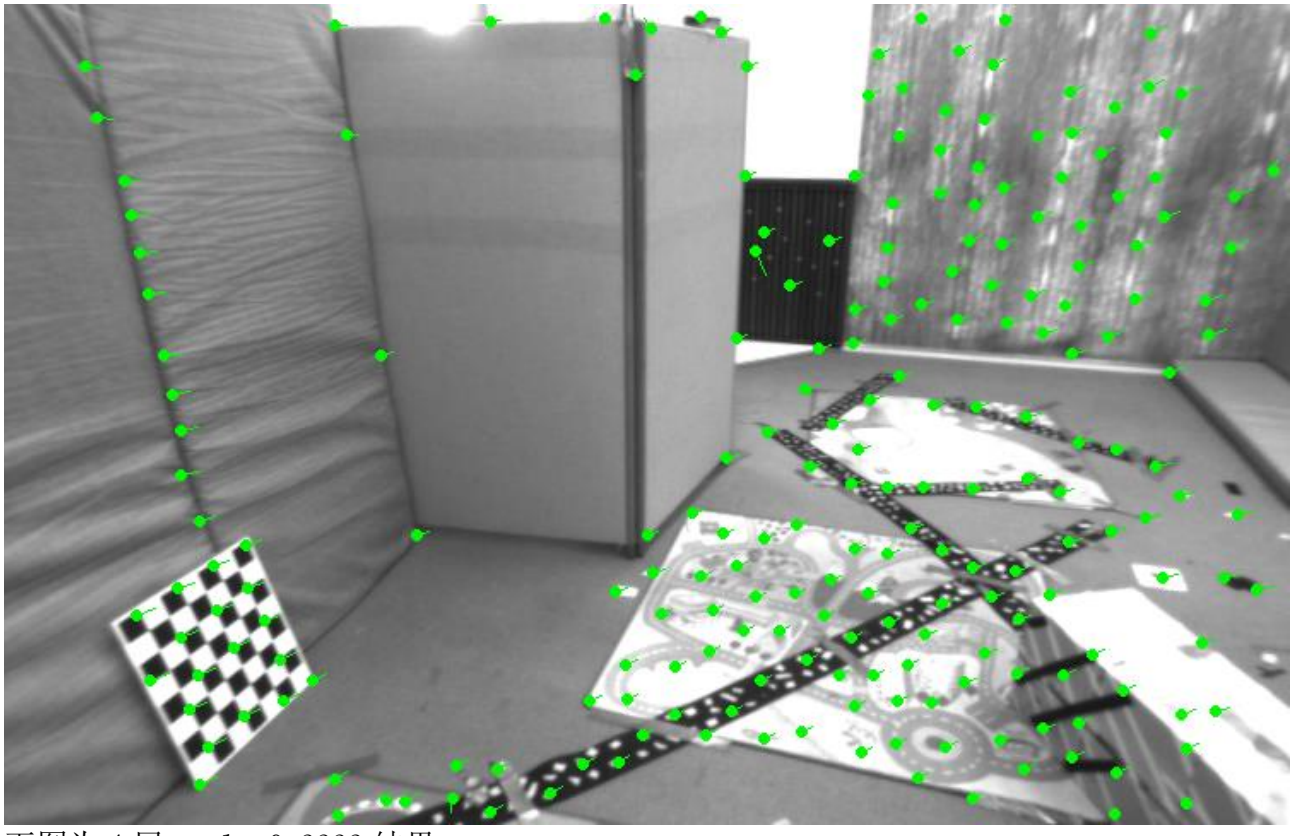
- 金字塔层数对结果有怎样的影响?缩放倍率呢?

下图为原始的 4 层 0.5 倍率结果



下图为 8 层 0.5 倍率结果。

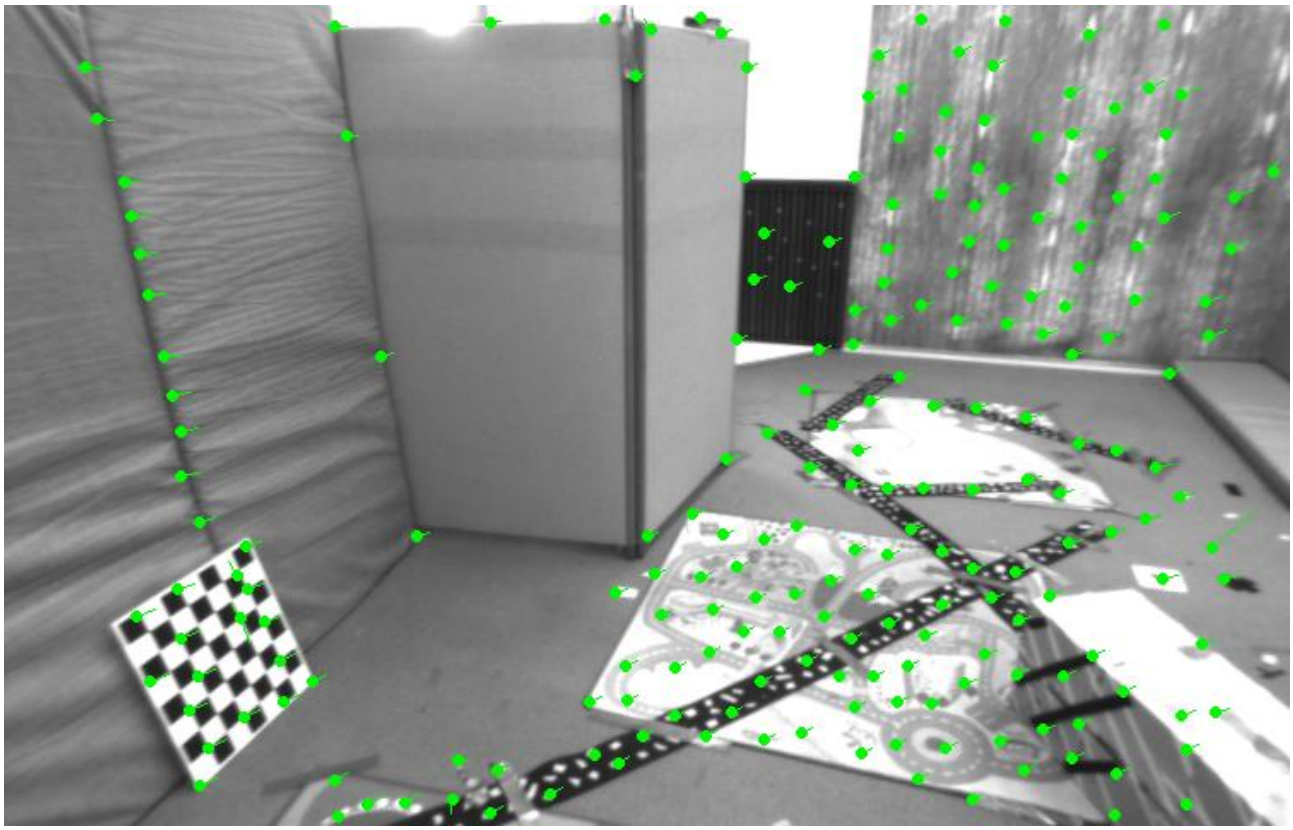
理论上 8 层可能更精细一些，对于大运动的鲁棒性更强一点，不过实验中可能相机运动比较小，体现不出来。



下图为 4 层 scale=0.3333 结果。

可见图中央黑色块状物有一个特征点的方向发生了变化（类似上题 batch 改变的结果）

同时左下角棋盘格有一个特征点的运动看起来不太正确，地板右边边缘处有一个特征点的运动幅度变大。



3 直接法 (5 分,约 2.5 小时)

1. 该问题中的误差项是什么?

类似光流法，是光度误差。

2. 误差相对于自变量的雅可比维度是多少?如何求解?

   误差 e 是 1 维，自变量 T 是 6 维度，则 J 为 1 行 6 列。

   其中$\frac{\partial \mathrm{e}}{\partial \mathrm{T}} = \frac{\partial I_2}{\partial u}\frac{\partial u}{\partial q}\frac{\partial q}{\partial \delta \xi} = 1\ 行 2\ 列 * 2\ 行 3\ 列 * 3\ 行 6\ 列 = 1\ 行 6\ 列$

3. 窗口可以取多大?是否可以取单个点

   课程中窗口为 3x3. 理论上可以取单个点，鲁棒性降低而已。还有就是没有像素变化的单个像素点就不对结果产生贡献。所以还是不要只取一个点。

代码（主要是构建 error，J 和 updates）

```cpp
    // TODO START YOUR CODE HERE
    float u = 0, v = 0;
    //back-project from image 1 to space
    Eigen::Vector3d point_1 =
            Eigen::Vector3d( x: (px_ref[i][0] - cx) / fx,  y: (px_ref[i][1] - cy) / fy,  z: 1) * depth_ref[i];
    //project to ref2
    Eigen::Vector3d point_2 = T21 * point_1;
    if (point_2[2] < 0) continue; //Z less Than 0
    //project to image2
    u = fx * point_2[0] / point_2[2] + cx;
    v = fy * point_2[1] / point_2[2] + cy;
    if (u < half_patch_size || u > img2.cols - half_patch_size || v < half_patch_size ||
        v > img2.rows - half_patch_size)
        continue; //out of image boundary
    nGood++;
    goodProjection.push_back(Eigen::Vector2d(u, v));

    // and compute error and jacobian
    for (int x = -half_patch_size; x < half_patch_size; x++)
        for (int y = -half_patch_size; y < half_patch_size; y++) {
            double error = 0;
            error = GetPixelValue(img1,  x: px_ref[i][0] + x,  y: px_ref[i][1] + y) - GetPixelValue(img2,  x: u + x,  y: v + y);
            Matrix26d J_pixel_xi;   // pixel to \xi in Lie algebra
            Eigen::Vector2d J_img_pixel;    // image gradients
            J_pixel_xi( row: 0,  col: 0) = fx / point_2[2];
            J_pixel_xi( row: 0,  col: 1) = 0;
            J_pixel_xi( row: 0,  col: 2) = -fx * point_2[0] / (point_2[2] * point_2[2]);
            J_pixel_xi( row: 0,  col: 3) = -fx * point_2[0] * point_2[1] / (point_2[2] * point_2[2]);
            J_pixel_xi( row: 0,  col: 4) = fx + fx * point_2[0] * point_2[0] / (point_2[2] * point_2[2]);
            J_pixel_xi( row: 0,  col: 5) = -fx * point_2[1] / point_2[2];
            J_pixel_xi( row: 1,  col: 0) = 0;
            J_pixel_xi( row: 1,  col: 1) = fy / point_2[2];
            J_pixel_xi( row: 1,  col: 2) = -fy * point_2[1] / (point_2[2] * point_2[2]);
            J_pixel_xi( row: 1,  col: 3) = -fy - fy * point_2[1] * point_2[1] / (point_2[2] * point_2[2]);
            J_pixel_xi( row: 1,  col: 4) = fy * point_2[0] * point_2[1] / (point_2[2] * point_2[2]);
            J_pixel_xi( row: 1,  col: 5) = fy * point_2[0] / point_2[2];

            J_img_pixel = Eigen::Vector2d(
                    x: 0.5 * (GetPixelValue(img2,  x: u + x + 1,  y: v + y) - GetPixelValue(img2,  x: u + x - 1,  y: v + y)),
                    y: 0.5 * (GetPixelValue(img2,  x: u + x,  y: v + y + 1) - GetPixelValue(img2,  x: u + x,  y: v + y - 1)));
            // total jacobian
            Vector6d J = -1 * (J_img_pixel.transpose() * J_pixel_xi).transpose();
            H += J * J.transpose();
            b += -error * J;

        }
    // END YOUR CODE HERE
}

// solve update and put it into estimation
// TODO START YOUR CODE HERE
Vector6d update;
update = H.ldlt().solve(b);
T21 = Sophus::SE3::exp(update) * T21;
// END YOUR CODE HERE
```

下图为 0001 图的结果，可见 cost 一直在下降。（0002-0005 图略）

```
xin@ubuntu16:~/VSLAM-course/vSLAM-course/Ch6/SLAM[illegible]/L6/code/cmake-build-debug$ ./direct_method
cost = 137911, good = 1000
cost = 126184, good = 1000
cost = 120182, good = 1000
cost = 114124, good = 1000
cost = 109089, good = 1000
cost = 104817, good = 1000
cost = 100996, good = 1000
cost = 96390.8, good = 1000
cost = 91361.1, good = 1000
cost = 86790.9, good = 1000
cost = 81702.3, good = 1000
cost = 77537.8, good = 1000
cost = 73685.1, good = 1000
cost = 70262.9, good = 1000
cost = 66961.9, good = 997
cost = 64466.8, good = 996
cost = 60753.7, good = 994
cost = 57802.8, good = 992
cost = 54855, good = 990
cost = 52927.3, good = 990
cost = 50701.3, good = 990
cost = 48647.2, good = 990
cost = 46225.6, good = 990
cost = 43841.1, good = 990
cost = 41382.7, good = 990
cost = 38888.4, good = 990
cost = 36050.6, good = 990
cost = 32233.8, good = 990
cost = 27514.9, good = 990
cost = 22244.2, good = 990
cost = 16565.3, good = 990
cost = 13523.3, good = 990
cost = 13087.5, good = 989
cost = 13041.7, good = 989
cost = 13036.6, good = 989
cost increased: 13036.6, 13036.6
good projection: 989
T21 =
   0.999991  0.00242132  0.00337215 -0.00184407
-0.00242871    0.999995  0.00218895   0.0026733
-0.00336684 -0.00219713    0.999992   -0.725126
          0           0           0           1
```

3.2 多层直接法(2 分)

主要就是把图像 resize 到不同的 level。同理 resize 内参。
在不同的 level 上跑单层直接法
代码

```cpp
// create pyramids
vector<cv::Mat> pyr1, pyr2; // image pyramids
// TODO START YOUR CODE HERE
for(int i=0;i<pyramids;i++)
{
    if (0 == i) {
        pyr1.push_back(img1);
        pyr2.push_back(img2);
    }
    else {
        cv::Mat upper_level_img1;
        cv::Mat upper_level_img2;
        cv::resize(img1, upper_level_img1, dsize: cv::Size( _width: img1.cols * scales[i], _height: img1.rows * scales[i]));
        cv::resize(img2, upper_level_img2, dsize: cv::Size( _width: img2.cols * scales[i], _height: img2.rows * scales[i]));
        pyr1.push_back(upper_level_img1);
        pyr2.push_back(upper_level_img2);
    }
}
// END YOUR CODE HERE

double fxG = fx, fyG = fy, cxG = cx, cyG = cy;  // backup the old values
for (int level = pyramids - 1; level >= 0; level--) {
    VecVector2d px_ref_pyr; // set the keypoints in this pyramid level
    for (auto &px: px_ref) {
        px_ref_pyr.push_back(scales[level] * px);
    }

    // TODO START YOUR CODE HERE
    // scale fx, fy, cx, cy in different pyramid levels
    fx = fxG * scales[level];
    fy = fyG * scales[level];
    cx = cxG * scales[level];
    cy = cyG * scales[level];
    // END YOUR CODE HERE
    DirectPoseEstimationSingleLayer(pyr1[level], pyr2[level], px_ref_pyr, depth_ref, &: T21);
}
```

结果

0001

Z 位移-0.697 接近参考值-0.725

```
xin@ubuntu16:~/VSLAM-course/vSLAM-course/Ch6/SLAM[illegible]/L6/code/cmake-build-debug$ ./direct_method
cost = 75389.2, good = 1000
cost = 52910.2, good = 986
cost = 36591.7, good = 965
cost = 30582.2, good = 946
cost = 29419.2, good = 941
cost = 29346.1, good = 939
cost = 29331.1, good = 939
cost = 29328, good = 939
cost = 29327.4, good = 939
cost = 29327.3, good = 939
cost = 29327.2, good = 939
cost = 29327.2, good = 939
cost increased: 29327.2, 29327.2
good projection: 939
T21 =
   0.999994  0.00176948   0.0028233  0.00351091
-0.00177647    0.999995  0.00247482  0.00191224
-0.00281891 -0.00247982    0.999993   -0.697176
          0           0           0           1
```

0005

Z 运动-3.79 接近参考-3.99。

```
cost = 93942.5, good = 672
cost = 93520.5, good = 672
cost = 93193.8, good = 672
cost = 93015.3, good = 672
cost = 92801.4, good = 672
cost = 92389.3, good = 673
cost = 92036.6, good = 673
cost = 91590.8, good = 674
cost = 91351.2, good = 674
cost = 90924.8, good = 676
cost = 90850.5, good = 676
cost = 90809.3, good = 676
cost = 90783.8, good = 676
cost = 90722.7, good = 676
cost = 90563.6, good = 677
cost = 90545, good = 677
cost = 90539.5, good = 677
cost increased: 90543.3, 90539.5
good projection: 677
T21 =
   0.999803   0.0011997   0.0198228   0.0189706
-0.00132935    0.999978  0.00652881  -0.0102958
 -0.0198145 -0.00655388    0.999782    -3.79305
          0           0           0           1
```

3.3* 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?

不行，直接法怎样都需要把第一帧图像的点投影到第二帧图像上面。链式 jacobian 中就包括一项第二张图像的像素梯度。而上述思想避免了求第二张图像的像素梯度，这跟直接法的 jacobian 定义相悖。

2. 请思考上面算法哪些地方可以缓存或加速?

雅可比矩阵可以并行计算，反正是每个点的结果累加。

3. 在上述过程中,我们实际假设了哪两个 patch 不变?

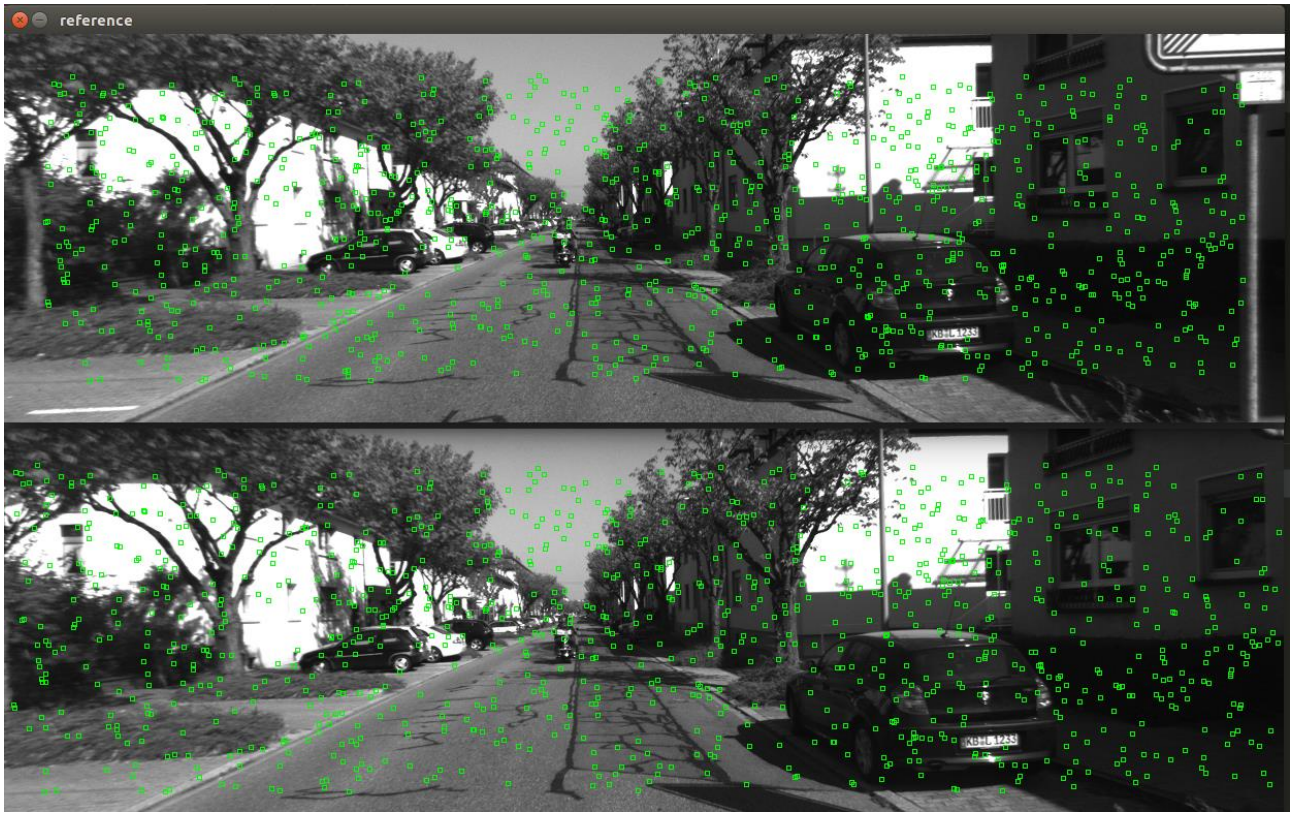
除了定义灰度不变外，同一个 patch 内我们用的是同一个深度，即深度也不变。

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?

直接法不需要寻找特征点的匹配，它计算的是像素前后帧的运动。不需要像素本身有特征性质。

下图可以看到，中间天空部分的点都找对了，而这些点显然不是特征点。

同理还有建筑物墙壁点，地面点。



5. 请总结直接法相对于特征点法的异同与优缺点。

都是通过像素在前一帧和后一帧的对应关系来计算两帧图像之间的运动。

相同点

特征点法是找到特征点对，构建的误差是 x-y 坐标的误差。

直接法找的是同一个像素，构建的误差是像素灰度误差。

不同点/优缺点

因为像素亮度不会真的一成不变，所以特征点法应该是更精确一些。

不过特征点法要找特征点，计算描述子，还要匹配特征点，比较耗时。

直接法省去了这些步骤，更快一些。而且可以在低特征的场景工作。

同时如果重建场景，特征点只能还原特征点在场景中的 3d 位置，也就是稀疏点云。

直接法在算力足够的情况可以计算所有像素的运动，从而重构稠密场景。

4 * 使用光流计算视差(2 分,约 1 小时)

即用第一张图片的特征点 kp1 的 x 坐标跟第二张图的特征点 kp2 的 x 坐标相减得到视差。

代码

```
//calculate dispartiy from optical flow
cv::Mat disparity = cv::imread(disparity_file,  flags: 0);
cv::Mat disparity_optical_flow = Mat::zeros(disparity.rows, disparity.cols, CV_8U);
vector<double> origin_disparity;
Eigen::VectorXd errors = Eigen::VectorXd(kp1.size());
double max_disparity = -999; double min_disparity = 999;
for(int i = 0; i< kp1.size();i++)
{
    //dispartiy = feature point1 .x - feature point2 .x
    double thisD = kp1[i].pt.x - kp2_multi[i].pt.x;
    double thisError = thisD - (int)disparity.at<uchar>(kp1[i].pt.y, kp1[i].pt.x);
    //cout<<"my d "<<thisD<<" ref D "<<(int)disparity.at<uchar>(kp1[i].pt.y, kp1[i].pt.x)<<" error "<<thisError<<endl;
    errors[i]=thisError;


}
```

为了方便显示，我把得到的视察数据（假设范围是-100 到 100）通过一个 scalar 变换到了(0,255)的范围。

显示结果如下。大致可以看出来视察点的位置和特征点是对应的。