第六讲作业

崔华坤

达闼科技（北京）有限公司

2018.4.2

二 LK 光流

2.1 光流文献综述

1. 光流法分为四种：Forward Additive(FA), Forward Compositional(FC)以及 Inverse Compositional(IC)算法, Inverse Additive(IA)算法。

增量方式\更新方式	forward	inverse
additive	FAIA	IAIA
compositional	FCIA	ICIA

2. Compositional 与 Additive 对比

通过增量的表示方式来区分方法. 迭代更新运动参数的时候，如果迭代的结果是在原始的值(6 个运动参数)上增加一个微小量，那么称之为 Additive，如果在仿射矩阵上乘以一个矩阵(增量运动参数形成的增量仿射矩阵)，这方法称之为 Compositional。两者理论上是等效的，而且计算量也差不多。

FCIA: warp 集合包含 identitywarp, warp 集合包含在 Composition 操作上是闭的(semi-group), 其中包括 Homograph, 3D rotation 等。

ICIA: semi-group, 另外要求增量 warp 可逆, 其中包括 Homograph, 3D rotation 等, 但不包括 piece wise affine。

warp 的物理意义是：对图像做微小的平移或者仿射变换。

3. 前向与后向对比

前向方法对于输入图像进行参数化(包括仿射变换及放射增量)。后向方法则同时参数输入图像和模板图像，其中输入图像参数化仿射变换，模板图像参数参数化仿射增量。因此后向方法的计算量显著降低。

参数化过程主要计算：图像的梯度，位置对运动参数导数，运动参数增量。前向方法中 Hessian 是运动参数的函数。提高效率的主要思想是交换模板图像和输入图像的角色。后向方法在迭代中 Hessian 是固定的。

■ 正向 FAIA 的目标函数为：

$$\underset{\Delta p}{\operatorname{argmin}} \sum_{x \in Q} \left[I\left(W(x; p + \Delta p)\right) - T(x) \right]^2$$

其中，I 为待跟踪帧，T 为参考帧，Q 表示 x 周围的一个窗口范围，假设该范围内的像素具有相同的运动，W 表示像素坐标变换关系如下：

$$W(x;p) = \begin{bmatrix} x + p1 \\ y + p2 \end{bmatrix}$$

令$\mathrm{g}(p) = I\left(W(x; p + \Delta p)\right) - T(x)$，一阶泰勒展开可得：

$$\begin{aligned} \mathrm{g}(p) &= I\left(W(x; p + \Delta p)\right) - T(x) \\ &= I\left(W(x;p)\right) + \nabla \mathrm{I} \frac{\partial W}{\partial p} \Delta p - T(x) \end{aligned}$$

其中，$\nabla \mathrm{I} = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]$为待跟踪帧在$W(x;p)$处的灰度梯度，$\frac{\partial W}{\partial p} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

■　反向 IAIA 的目标函数为：

$$\underset{\Delta p}{\operatorname{argmin}} \sum_{x \in Q} \left[T\left(W(x; \Delta p)\right) - I\left(W(x;p)\right) \right]^2$$

令$\mathrm{g}(p) = T\left(W(x; \Delta p)\right) - I\left(W(x;p)\right)$，一阶泰勒展开可得：

$$\begin{aligned} \mathrm{g}(p) &= T\left(W(x; \Delta p)\right) - I\left(W(x;p)\right) \\ &= T\left(W(x;0)\right) + \nabla \mathrm{T} \frac{\partial W}{\partial p} \Delta p - I\left(W(x;p)\right) \end{aligned}$$

$\nabla \mathrm{T} = \left[\frac{\partial T}{\partial x}, \frac{\partial T}{\partial y}\right]$为参考帧 T 在 x 处的灰度梯度，是不随迭代变化的。

■　FAIA 和 IAIA 的误差项均为：

$$e(p) = I\left(W(x;p)\right) - T(x)$$

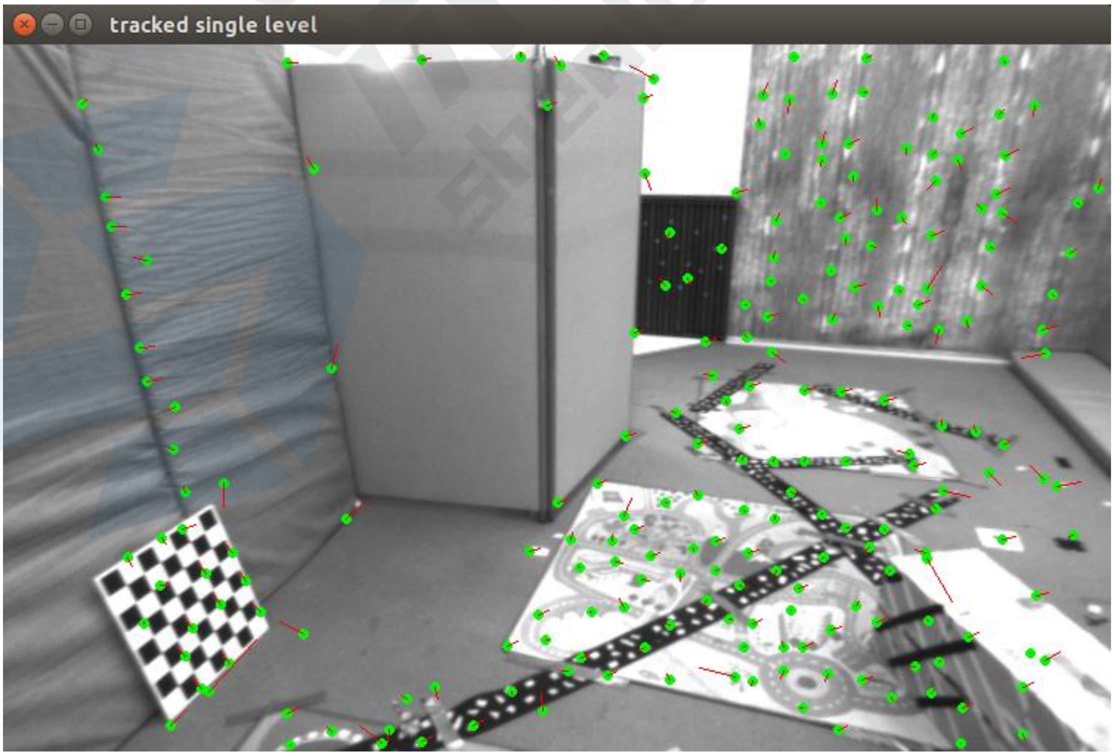
2.2 forward-addtive Gauss-Newton 光流的实现

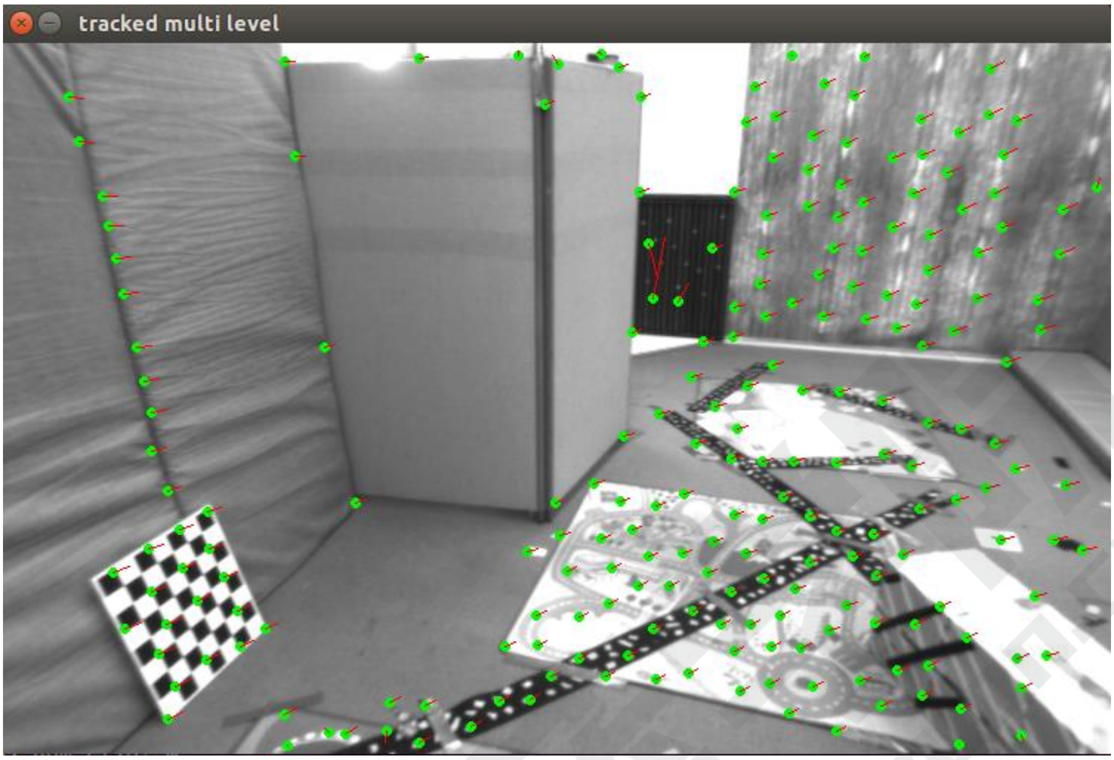
1. 从最小二乘角度来看，每个像素的误差定义为：

$$\mathrm{g}(p) = I\left(W(x; p + \Delta p)\right) - T(x)$$

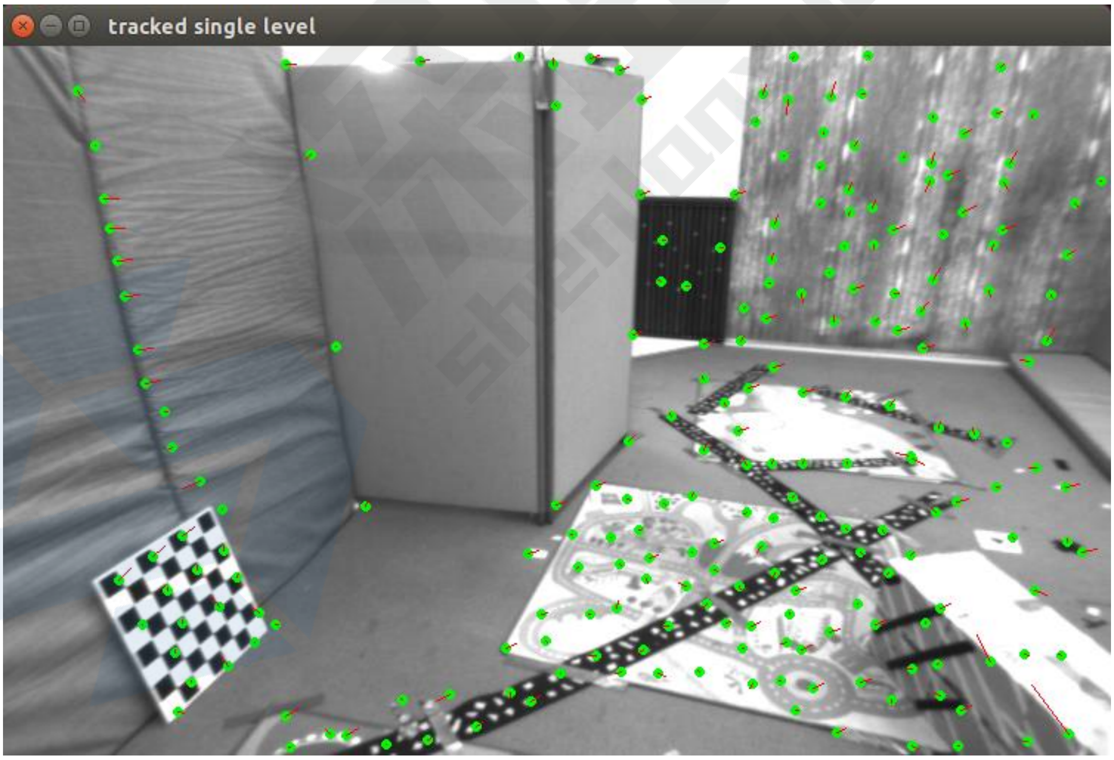
2. 误差相对于自变量的导数为：$\frac{\partial g}{\partial p} = \nabla \mathrm{I} \frac{\partial W}{\partial p} = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]$

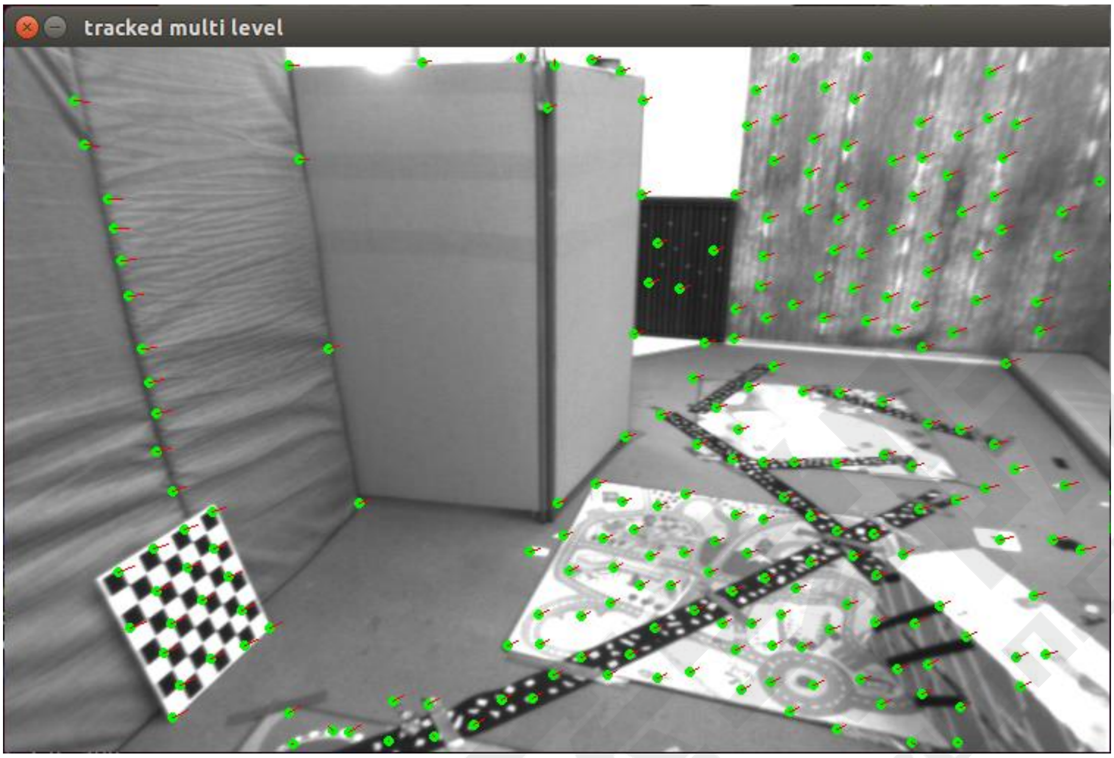
```
// compute cost and jacobian
for (int x = -half_patch_size; x < half_patch_size; x++)
    for (int y = -half_patch_size; y < half_patch_size; y++) {

        // TODO START YOUR CODE HERE (~8 lines)
        float u1 = float(kp.pt.x + x);        float v1 = float(kp.pt.y + y);
        float u2 = float(u1 + dx);            float v2 = float(v1 + dy);
        double error = 0;
        Eigen::Vector2d J;  // Jacobian
        if (inverse == false) {
            // Forward Jacobian
            J.x() = double(GetPixelValue(img2, u2 + 1, v2) - GetPixelValue(img2, u2 - 1, v2))/2;
            J.y() = double(GetPixelValue(img2, u2, v2 + 1) - GetPixelValue(img2, u2, v2 - 1))/2;
            error = double(GetPixelValue(img2, u2, v2) - GetPixelValue(img1, u1, v1));
        } else {
            // Inverse Jacobian
            // NOTE this J does NOT change when dx, dy is updated, so we can store it and only compute error
            J.x() = double(GetPixelValue(img1, u1 + 1, v1) - GetPixelValue(img1, u1 - 1, v1))/2;
            J.y() = double(GetPixelValue(img1, u1, v1 + 1) - GetPixelValue(img1, u1, v1 - 1))/2;
            error = double(GetPixelValue(img2, u2, v2) - GetPixelValue(img1, u1, v1));
        }

        // compute H, b and set cost;
        H += J * J.transpose();
        b += -J * error;
        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);
//cout<<"iter: "<<iter<<" update: "<<update.transpose()<<endl;
// TODO END YOUR CODE HERE

if (std::isnan(update[0])) {
    // sometimes occurred when we have a black or white patch and H is irreversible
    cout << "update is nan" << endl;
    succ = false;
    break;
}
if (iter > 0 && cost > lastCost) {
    //cout << "cost increased: " << cost << ", " << lastCost << endl;
    break;
}

// update dx, dy
dx += update[0];
dy += update[1];
lastCost = cost;
succ = true;
```





2.3 反向法：





2.4 推广至金字塔

光流截图如 2.2 和 2.3 所示。

1. 所谓 coarse-to-fine 只指：先跟踪金字塔的最顶层，然后用被跟踪帧在最顶层的跟踪结果，作为次顶层的跟踪初始值，再次进行跟踪，依次以至第 0 层（即原始图像）。
2. 光流法中的金字塔的目的是为了排除图像运动过快导致跟踪不上，因此可以先跟踪金字塔上层的下采样后的图像，然后层层细化。

   特征点中的金字塔是为了排除焦点距离的影响，即远处看是一个特征点，但是近看时却不是，当近看时可以用金字塔的上面几层来跟踪，来实现尺度不变性。

```
void OpticalFlowMultiLevel(
        const Mat &img1,
        const Mat &img2,
        const vector<KeyPoint> &kp1,
        vector<KeyPoint> &kp2,
        vector<bool> &success,
        bool inverse) {

    // parameters
    int pyramids = 4;
    double pyramid_scale = 0.25;
    double scales[] = {1.0, 0.25, 0.0625, 0.015625};// {1.0, 0.5, 0.25, 0.125};

    // create pyramids
    vector<Mat> pyr1, pyr2; // image pyramids
    // TODO START YOUR CODE HERE (~8 lines)
    for (int i = 0; i < pyramids; i++) {
        Mat img1_temp,  img2_temp;
        resize(img1, img1_temp, Size(img1.cols * scales[i], img1.rows * scales[i]));
        resize(img2, img2_temp, Size(img2.cols * scales[i], img2.rows * scales[i]));
        pyr1.push_back(img1_temp);
        pyr2.push_back(img2_temp);
        cout<<"Pyramid: "<<i<<" img1 size: "<<img1_temp.cols<<" "<<img1_temp.rows<<endl;
    }
    // TODO END YOUR CODE HERE
    // coarse-to-fine LK tracking in pyramids
    // TODO START YOUR CODE HEREi
    vector<KeyPoint> vkp2_now;
    vector<KeyPoint> vkp2_last;
    vector<bool> vsucc;
    for(int i = pyramids - 1; i >= 0; i--)
    {
        vector<KeyPoint> vkp1;
        for(int j = 0; j < kp1.size(); j++) {
            KeyPoint kp1_temp = kp1[j];
            kp1_temp.pt *= scales[i];
            vkp1.push_back(kp1_temp);
            if(i < pyramids - 1) {
                KeyPoint kp2_temp = vkp2_last[j];
                kp2_temp.pt /= pyramid_scale;
                vkp2_now.push_back(kp2_temp);
            }
        }
        vsucc.clear();
        OpticalFlowSingleLevel(pyr1[i], pyr2[i], vkp1, vkp2_now, vsucc, inverse);
        vkp2_last.clear();
        vkp2_last.swap(vkp2_now);
        cout<<"pyramid: "<<i<<" vkp2_last size: "<<vkp2_last.size() <<" vkp2_now size: "<<vkp2_now.size()<<endl;
    }
    kp2 = vkp2_last;
    success = vsucc;
```

2.5 讨论

1. 优化两个图像块的灰度差是否合理？

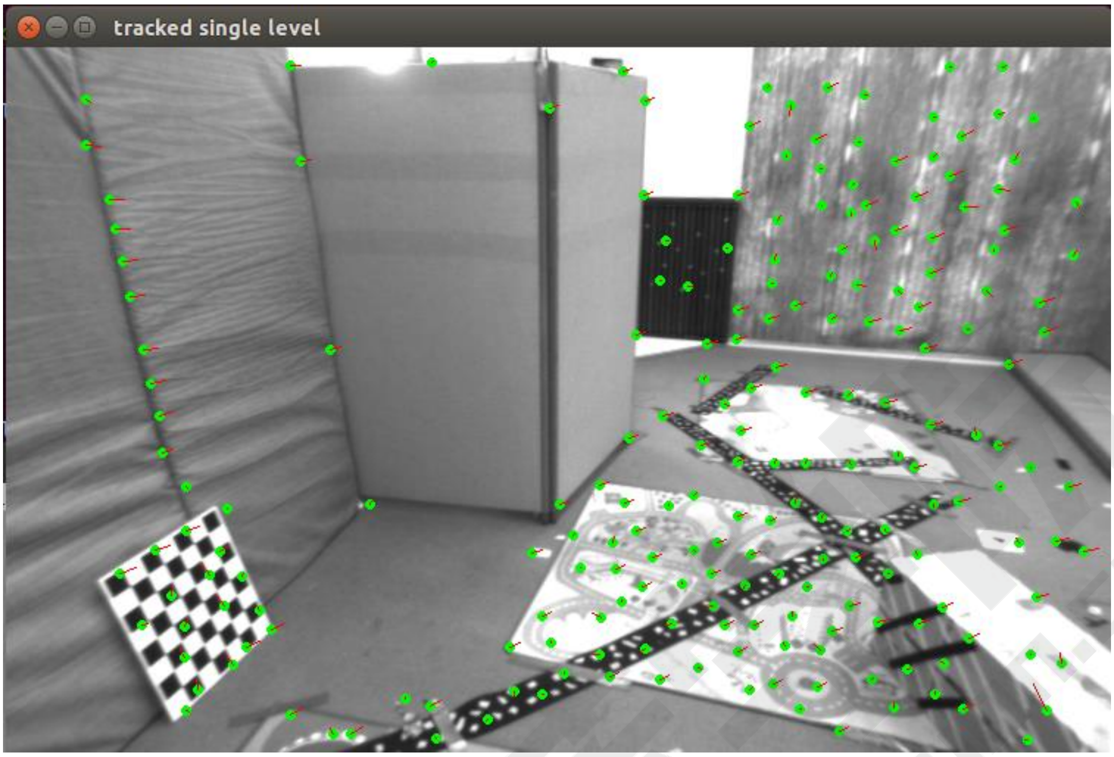
   光流法有三个假设：a)灰度不变假设，b)小运动假设，c)局部一致性假设。其中，灰度不变假设，当相机存在自动曝光、或者当物体有高光或者阴影时，灰度不变假设不在满足。

   解决方案：

   对相机进行光度模型标定，将图像校正到一致状态。

2. 图像块大小是否有明显差异？

   当图像块从 8×8 调到 16×16，单层 IAIA 效果略有改善，多层金字塔效果不明显。



3. 将金字塔数从 4 层改为 6 层，结果基本无差别；将 4 层的缩放倍率从 0.5 改为 0.25，结果基本无差别。

三、直接法

3.1 单层直接法

1. 目标函数为：

$$\underset{\xi}{\operatorname{argmin}} \frac{1}{N}\sum_{i=0}^{N}\sum_{W_i}\|I_1(p_1,i)-I_2(p_2,i)\|^2$$

误差项为：

$$g(\xi)=I_1(p_1,i)-I_2(p_2,i)$$

2. 雅可比矩阵为 1×6，为：

$$J(\xi)=\frac{\partial g}{\partial\delta\xi}=-\frac{\partial I_2}{\partial P_{uv}}\frac{\partial P_{uv}}{\partial P_c}\frac{\partial P_c}{\partial\delta\xi}$$

其中，

$$\frac{\partial P_c}{\partial\delta\xi}=[I,-P_c^\wedge]$$

$$\frac{\partial P_{uv}}{\partial P_c}=\begin{bmatrix}\frac{f_x}{z_c} & 0 & -\frac{f_x x_c}{z_c^2}\\ 0 & \frac{f_y}{z_c} & -\frac{f_y y_c}{z_c^2}\end{bmatrix}$$

$$J(\xi) = -\frac{\partial I_2}{\partial P_{uv}}\frac{\partial P_{uv}}{\partial \delta\xi} = -\frac{\partial I_2}{\partial P_{uv}}\begin{bmatrix} \frac{f_x}{z_c} & 0 & -\frac{f_x x_c}{{z_c}^2} & -\frac{f_x x_c y_c}{{z_c}^2} & f_x + \frac{f_x {x_c}^2}{{z_c}^2} & -\frac{f_x y_c}{z_c} \\ 0 & \frac{f_y}{z_c} & -\frac{f_y y_c}{{z_c}^2} & -f_y - \frac{f_y {y_c}^2}{{z_c}^2} & \frac{f_y x_c y_c}{{z_c}^2} & \frac{f_y x_c}{z_c} \end{bmatrix}$$

3. 窗口可以取单个点。

运行结果：

```
stevencui@ubuntu:~/Project/SLAMCourse/l6-3-direct/build$ ./direc
cost increased: 12697.7, 12695.4
good projection: 1000
Image 1 T_cur_ref is :
   0.999991  0.00235965  0.00339477 -0.00261845
-0.00236705    0.999995  0.00217539  0.00364224
-0.00338962  -0.0021834    0.999992    -0.72437
          0           0           0           1
good projection: 1000
Image 2 T_cur_ref is :
   0.999972   0.0012884  0.00732682  0.00501426
-0.00131622    0.999992  0.00379333   0.0021886
-0.00732187 -0.00380287    0.999966    -1.46576
          0           0           0           1
cost increased: 81240.2, 81239.7
good projection: 1000
Image 3 T_cur_ref is :
   0.999904 -2.2918e-05   0.0138574   -0.282777
-5.5171e-05    0.999984  0.00563476 -0.00690948
 -0.0138573 -0.00563498    0.999888    -1.79342
          0           0           0           1
cost increased: 153723, 153721
good projection: 1000
Image 4 T_cur_ref is :
   0.999846  0.00176662   0.0174483   -0.391273
-0.00186677    0.999982  0.00572519   0.0063952
 -0.0174379 -0.00575688    0.999831    -1.89371
          0           0           0           1
good projection: 1000
Image 5 T_cur_ref is :
  0.999664 0.00402454  0.0256015 -0.711438
-0.00414919    0.99998 0.00481769  0.0409335
-0.0255816 -0.0049223   0.999661  -2.38015
         0          0          0          1
```

Image1 跟踪结果：

Image5 跟踪结果：

单层代码：

```
for (int iter = 0; iter < iterations; iter++) {
    nGood = 0;
    goodProjection.clear();

    // Define Hessian and bias
    Matrix6d H = Matrix6d::Zero();  // 6x6 Hessian
    Vector6d b = Vector6d::Zero();  // 6x1 bias

    for (size_t i = 0; i < px_ref.size(); i++) {

        // compute the projection in the second image
        // TODO START YOUR CODE HERE
        float u = px_ref[i][0];
        float v = px_ref[i][1];
        double depth = depth_ref[i];
        double xc2_opt, yc2_opt, zc2_opt;
        float u2_opt, v2_opt;
        GetCurrFramePostionFromRefFrame(u, v, depth, T21, xc2_opt, yc2_opt, zc2_opt, u2_opt, v2_opt);
        if(!bInImage(u - half_patch_size, v - half_patch_size, img1.cols, img1.rows) &&
            !bInImage(u + half_patch_size - 1, v + half_patch_size - 1, img1.cols, img1.rows) &&
            !bInImage(u2_opt - half_patch_size, v2_opt - half_patch_size, img2.cols, img2.rows) &&
            !bInImage(u2_opt + half_patch_size - 1, v2_opt + half_patch_size - 1, img2.cols, img2.rows)) {
            continue;
        }
        nGood++;
        goodProjection.push_back(Eigen::Vector2d(u2_opt, v2_opt));

        // 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++)
            //int x = 0; int y = 0;
            {
                float u1 = float(u + x);
                float v1 = float(v + y);
                double xc2, yc2, zc2;
                float u2, v2;
                GetCurrFramePostionFromRefFrame(u1, v1, depth, T21, xc2, yc2, zc2, u2, v2);
                double error = GetPixelValue(img1, u1, v1) - GetPixelValue(img2, u2, v2);

                Matrix26d J_pixel_xi = Matrix26d::Zero();   // pixel to \xi in Lie algebra
                double xc2_2 = xc2 * xc2;  double yc2_2 = yc2 * yc2;   double zc2_2 = zc2 * zc2;
                J_pixel_xi(0,0) = fx / zc2;
                J_pixel_xi(0,2) = - fx * xc2 / zc2_2;
                J_pixel_xi(0,3) = - fx * xc2 * yc2 / zc2_2;
                J_pixel_xi(0,4) = fx + fx * xc2_2 / zc2_2;
                J_pixel_xi(0,5) = - fx * yc2 / zc2;
```

```
            J_pixel_xi(1,1) = fy / zc2;
            J_pixel_xi(1,2) = -fy * yc2 / zc2_2;
            J_pixel_xi(1,3) = -fy - fy * yc2_2 / zc2_2;
            J_pixel_xi(1,4) = fy * xc2 * yc2 / zc2_2;
            J_pixel_xi(1,5) = fy * xc2 / zc2;

            Eigen::Vector2d J_img_pixel = Vector2d::Zero();    // image gradients
            if(!bInImage(u2 - 1, v2 - 1, img2.cols, img2.rows) &&
                !bInImage(u2 + 1, v2 + 1, img2.cols, img2.rows)) {
                continue;
            }
            J_img_pixel[0] = (GetPixelValue(img2, u2 + 1, v2) - GetPixelValue(img2, u2 - 1, v2)) / 2;
            J_img_pixel[1] = (GetPixelValue(img2, u2, v2 + 1) - GetPixelValue(img2, u2, v2 - 1)) / 2 ;
            // total jacobian
            Vector6d J = -J_pixel_xi.transpose() * J_img_pixel;

            H += J * J.transpose();
            b += -error * J;
            cost += error * error;
        }
        // END YOUR CODE HERE
    }

    // solve update and put it into estimation
    // TODO START YOUR CODE HERE
    Vector6d update;
    update = H.ldlt().solve(b);
    if(DEBUG) cout<<"iter: "<<iter<<" update: "<<update.transpose()<<endl;
    T21 = Sophus::SE3::exp(update) * T21;
    // END YOUR CODE HERE
```

3.2 多层直接法

Image1 跟踪结果：

Image5 跟踪结果：

```
stevencui@ubuntu:~/Project/SLAMCourse/l6-3-direct/build$ ./direct_method
Image 1 T_cur_ref is :
   0.999991  0.00235973  0.00339463 -0.00261007
-0.00236712    0.999995  0.00217551  0.00363778
-0.00338948 -0.00218353    0.999992   -0.724384
          0           0           0           1
Image 2 T_cur_ref is :
   0.999972  0.00128793  0.00732734  0.00501752
-0.00131575    0.999992  0.00379322  0.00219306
-0.00732239 -0.00380276    0.999966    -1.46585
          0           0           0           1
Image 3 T_cur_ref is :
   0.999938  0.00150392   0.0110243  0.00952562
-0.00156267    0.999985  0.00532248 -0.00120089
 -0.0110161 -0.00533938    0.999925    -2.20375
          0           0           0           1
Image 4 T_cur_ref is :
    0.999833 -0.000349491    0.0182772    -0.224717
  0.00020477     0.999969   0.00791944   -0.0599317
  -0.0182793  -0.00791437     0.999802     -2.66913
           0            0            0            1
Image 5 T_cur_ref is :
   0.999809  0.00109498   0.0195204   0.0258622
-0.00125444    0.999966  0.00815873  -0.0517079
 -0.0195108 -0.00818166    0.999776    -3.76756
          0           0           0           1
```

3.3 延伸讨论

1. 直接法是否可类似光流法提出 inverse、compositional 概念？

答：可以。先给出直接法的误差函数：

$$g(\xi) = I_1(P_{uv1}) - I_2(P_{uv2})$$
$$= I_1(P_{uv1}) - I_2\left(\frac{1}{Z_{c2}} K T_{2\leftarrow 1} P_{c1}\right)$$

误差函数的左扰动后为：

$$g(\xi \oplus \delta\xi) = I_1(P_{uv1}) - I_2\left(\frac{1}{Z_{c2}} K exp(\delta\xi^\wedge) exp(\xi^\wedge) P_{c1}\right)$$

若按照反向光流法思想，相当于微调参考帧的点坐标，使目标函数最小化，则上式可写成：

$$g(\xi \oplus \delta\xi) = I_1\big(W(P_{uv1};\Delta p)\big) - I_2\left(\frac{1}{Z_{c2}} K exp(\xi^\wedge) P_{c1}\right)$$

则每次迭代后的被跟踪点坐标为：

$$P_{uv2} = \frac{1}{Z_{c2}} K exp(\xi^\wedge) P_{c1} = \frac{Z_{c1}}{Z_{c2}} K exp(\xi^\wedge) W(P_{uv1};\Delta p) K^{-1}$$

若再借鉴 Additive 思想，则$W(P_{uv1};\Delta p) = P_{uv1} + \Delta p$

2. 单层耗时统计：

	Imag1	Image2	Imag3	Image4	Imag5	平均(ms)
原程序	202	321	169	111	254	211.4

修改后	159	258	126	50	187	156

原程序在计算一个小窗口 8×8 内每个像素位置的 J 时，是通过如下方式计算：

$$J(\xi) = -\frac{\partial I_2}{\partial P_{uv}}\frac{\partial P_{uv}}{\partial \delta\xi} = -\frac{\partial I_2}{\partial P_{uv}}\begin{bmatrix} \frac{f_x}{z_c} & 0 & -\frac{f_x x_c}{{z_c}^2} & -\frac{f_x x_c y_c}{{z_c}^2} & f_x + \frac{f_x {x_c}^2}{{z_c}^2} & -\frac{f_x y_c}{z_c} \\ 0 & \frac{f_y}{z_c} & -\frac{f_y y_c}{{z_c}^2} & -f_y - \frac{f_y {y_c}^2}{{z_c}^2} & \frac{f_y x_c y_c}{{z_c}^2} & \frac{f_y x_c}{z_c} \end{bmatrix} \quad (1)$$

其中，$\frac{\partial P_{uv}}{\partial \delta\xi}$与被跟踪图像上的点在相机坐标系中的坐标$P_{c2}$有关，而$P_{c2}$由以下公式计算：

$$P_{c2} = T_{2\leftarrow 1}P_{c1} = T_{2\leftarrow 1}(z_{c1}P_{uv1}K^{-1}) \quad (2)$$

因此对于窗口内的每一个点都要重新计算一次P_{c2}。

修改思路是：假设小窗口内的 8×8 个像素，对应在相机坐标系下是同一个点，即小窗口内的所有像素在相机坐标系下的坐标相等。这样，我们就可以用窗口中心点的P_{c2}来表示窗口内其他点的P_{c2}，从而将$\frac{\partial P_{uv}}{\partial \delta\xi}$移出到窗口循环外部，且避免了 64 次公式(1)的局部雅可比计算及 64 次公式(2)的投影变换运算。

经过上述修改，单层直接法耗时可减小 26%！

3. 两个 patch 不变假设：窗口尺寸为 8×8 不变，灰度梯度时 3×3 不变。

4. 为什么可以取随机点？

因为直接法是基于灰度不变假设，对于任意点，在不同图像中的灰度值均相同，从而通过大量随机点的光度误差最小值，求出相机位姿。因此，无所谓是否为角点。

那些不是角点的随机点，跟踪的结果基本正确，但是在图像右上方白墙上有些点的跟踪结果有偏差。

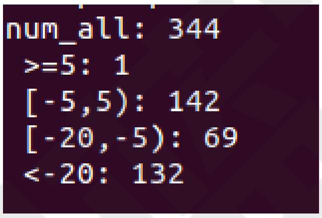
5. 直接法对比特征点法的异同和优缺点

	优点	缺点
特征点法	对光照有一定容忍度，运动过大时，只要图像中还有匹配点，则不容易丢失，有更好的鲁棒性。	当环境纹理的特征点少时易失败，特征点过于集中时易产生退化，相机运动过快，图像模糊时易失败；特征点计算耗时；
直接法	对于有亮暗变化的若纹理仍	因图像非凸，易陷入局部极

	有效，运行速度快，省掉特征点提取、描述过程，不用进行对极约束、PnP、ICP 等计算，可同时优化相机位姿、3D 点位置和匹配点位置。可产生稠密或半稠密地图。	小；相机存在自动曝光、或物体有高光阴影时易失败，相机运动过快时易失败

四、使用光流计算视差

根据多层光流法计算出左右图的匹配关系，并计算出水平视差，与 disparity.png 对比如下：



左右总共匹配了 344 个，下图中的数字表示光流法计算出来的水平视差，与 disparity.png 的差值，其中差值为>=5 有 1 个（用蓝色表示），[-5,5)有 142 个（用绿色表示），多集中在图像中心区域，视差计算基本准确，[-20,-5)有 69 个（用黄色表示），分布在绿色周围，<-20 的有 132 个（用红色表示），分布在图像四周，说明四周计算出来的视差误差较大。