Jindong_Shi
2021-10-27

第 6 节课习题

2. LK 光流

2.1 光流文献综述

1) 按文献中的分类，光流法可以分为 the additive approach 和 the compositional approach, 或者分为 forwards 和 inverse。

2) **The additive approach:**

in Eq. (3), the Lucas-Kanade algorithm assumes that a current estimate of $\mathbf{p}$ is known and then iteratively solves for increments to the parameters $\Delta\mathbf{p}$; i.e. the following expression is (approximately) minimized:

$$\sum_{\mathbf{x}}[I(\mathbf{W}(\mathbf{x};\mathbf{p}+\Delta\mathbf{p}))-T(\mathbf{x})]^2 \qquad (4)$$

with respect to $\Delta\mathbf{p}$, and then the parameters are updated:

$$\mathbf{p} \leftarrow \mathbf{p}+\Delta\mathbf{p}. \qquad (5)$$

These two steps are iterated until the estimates of the parameters $\mathbf{p}$ converge. Typically the test for convergence is whether some norm of the vector $\Delta\mathbf{p}$ is below a threshold ϵ; i.e. $\|\Delta\mathbf{p}\| \le \epsilon$.

The compositional approach:

3.1.1. Goal of the Compositional Algorithm. The compositional algorithm, used most notably by Shum and Szeliski (2000), approximately minimizes:

$$\sum_{\mathbf{x}}[I(\mathbf{W}(\mathbf{W}(\mathbf{x};\Delta\mathbf{p});\mathbf{p}))-T(\mathbf{x})]^2 \qquad (12)$$

with respect to $\Delta\mathbf{p}$ in each iteration and then updates the estimate of the warp as:

$$\mathbf{W}(\mathbf{x};\mathbf{p}) \leftarrow \mathbf{W}(\mathbf{x};\mathbf{p})\circ\mathbf{W}(\mathbf{x};\Delta\mathbf{p}), \qquad (13)$$

文献[1]对此已经说明地十分清楚。The compositional approach 更新的是 incremental warp $\mathbf{W}(\mathbf{x};\Delta\mathbf{p})$, 而非 additive update to the parameter $\Delta\mathbf{p}$。对原始图像做 wrap 相当于做一个仿射(affine)变换。

3) 我们分两种情况，

首先是 forward-additive 和 inverse-additive 算法的不同:

gorithm is the same as the Lucas-Kanade algorithm; i.e. to minimize $\sum_{\mathbf{x}}[\,I(\mathbf{W}(\mathbf{x};\mathbf{p}+\Delta\mathbf{p}))-T(\mathbf{x})\,]^2$ with respect to $\Delta\mathbf{p}$ and then update the parameters $\mathbf{p} \leftarrow \mathbf{p}+\Delta\mathbf{p}$. Rather than changing variables like in Section 3.2.5, the roles of the template and the image are switched as follows. First the Taylor expansion is performed, just as in Section 2.1:

$$\sum_{\mathbf{x}}\left[I(\mathbf{W}(\mathbf{x};\mathbf{p}))+\nabla I\frac{\partial\mathbf{W}}{\partial\mathbf{p}}\Delta\mathbf{p}-T(\mathbf{x})\right]^2. \qquad (41)$$

The template and the image are then switched by deriving the relationship between ∇I and ∇T. In Hager and Belhumeur (1998) it is assumed that the current estimates of the parameters are approximately correct: i.e.

$$I(\mathbf{W}(\mathbf{x};\mathbf{p})) \approx T(\mathbf{x}) \qquad (42)$$

然后是 forward-composional 和 inverse-composional 的不同:

Section 3.2.5. The result is that the inverse compositional algorithm minimizes:

$$\sum_{\mathbf{x}}[\,T(\mathbf{W}(\mathbf{x};\Delta\mathbf{p}))-I(\mathbf{W}(\mathbf{x};\mathbf{p}))\,]^2 \qquad (31)$$

with respect to $\Delta\mathbf{p}$ (note that the roles of I and T are reversed) and then updates the warp:

$$\mathbf{W}(\mathbf{x};\mathbf{p}) \leftarrow \mathbf{W}(\mathbf{x};\mathbf{p})\circ\mathbf{W}(\mathbf{x};\Delta\mathbf{p})^{-1}. \qquad (32)$$

The only difference from the update in the forwards compositional algorithm in Eq. (13) is that the incremental warp $\mathbf{W}(\mathbf{x};\Delta\mathbf{p})$ is *inverted* before it is composed with the current estimate. For example, the parameters

2.2 forward-additive Gauss-Newton 光流的实现

1) 对于任意一个特征点像素 $p_i = [x_i,\ y_i]^T$ 的误差可定义为该像素点所在 8x8 大小的窗口内所有像素灰度差的平方:

$$e_i = \frac{1}{2}\sum_{x=x_i-4}^{x_i+3}\sum_{y=y_i-4}^{y_i+3}\|I_1(x,y) - I_2(x+\Delta x, y+\Delta y)\|^2$$

2) 求误差相对于自变量的导数

可将 1)中的误差项 $f(\mathbf{x},\Delta\mathbf{x}) = I_1(\mathbf{x}) - I_2(\mathbf{x}+\Delta\mathbf{x}) = I_1(\mathbf{x}) - I_2(\mathbf{W}(\mathbf{x};\Delta\mathbf{x}))$，其中 $\mathbf{W}(\mathbf{x};\Delta\mathbf{x}) = \mathbf{x}+\Delta\mathbf{x} = \begin{pmatrix} x+\Delta x \\ y+\Delta y \end{pmatrix}$为像素点经运动之后得到的坐标，进行泰勒展开，得

$$f(\mathbf{x},\Delta\mathbf{x}+\Delta\mathbf{u}) = I_1(\mathbf{x}) - I_2(\mathbf{x}+\Delta\mathbf{x}) \underbrace{-\frac{\partial I_2}{\partial \mathbf{W}}\frac{\partial \mathbf{W}}{\partial \Delta\mathbf{x}}}_{J(\Delta\mathbf{x})}\Delta\mathbf{u}$$

这里 $\Delta\mathbf{u}$ 是 $\Delta\mathbf{x}$ 的更新量。

那么误差项 e_i 相对于自变量 $\Delta\mathbf{x}$ 的导数为

$$J(\Delta\mathbf{x}) = -\frac{\partial I_2}{\partial \mathbf{W}}\frac{\partial \mathbf{W}}{\partial \Delta\mathbf{x}}$$

其中 $\frac{\partial I_2}{\partial \mathbf{W}} = \left(\frac{I_2(W_x+1,W_y)-I_2(W_x-1,W_y)}{2},\ \frac{I_2(W_x,W_y+1)-I_2(W_x,W_y-1)}{2}\right)$ 为 $\mathbf{W}$ 处的的像素梯度。

由于$\frac{\partial \mathbf{W}}{\partial \Delta\mathbf{x}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$为单位矩阵，故 $J(\Delta\mathbf{x}) = -\frac{\partial I_2}{\partial \mathbf{W}}$

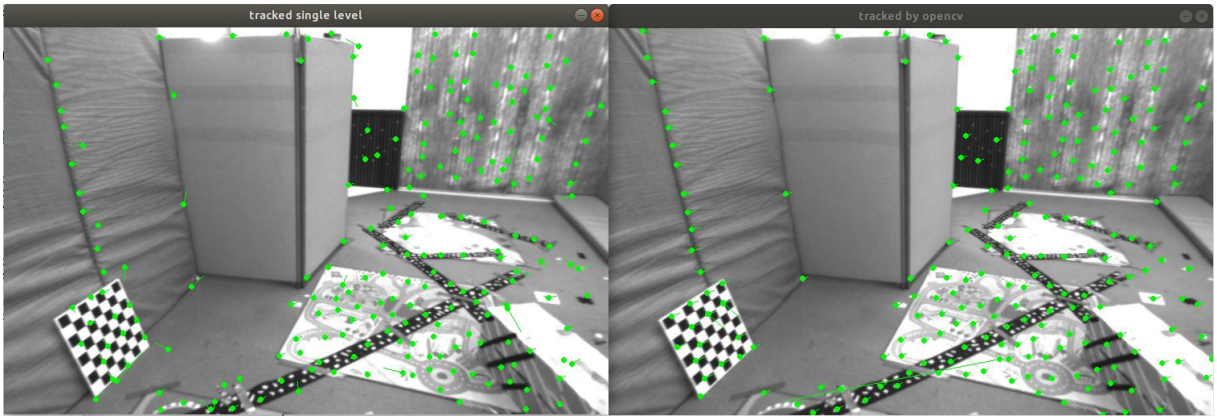
源文件 optical_flow.cpp 位于文件夹 Optical_Flow 内，部分代码如下:

```
// 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)
        double error = GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y) - GetPixelValue(img2, kp.pt.x + x + dx, kp.pt.y + y + dy);
        Eigen::Vector2d J;  // Jacobian
        if (inverse == false) {
            // Forward Jacobian
            // 注意: x,y为相对角点的距离坐标，为取得像素在图像内的绝对坐标需要加上角点的绝对坐标(kp.pt.x, kp.pt.y)
            // 注意: 雅可比前面的负号
            J(0) = (GetPixelValue(img2, kp.pt.x + x + dx + 1, kp.pt.y + y + dy) - GetPixelValue(img2, kp.pt.x + x + dx - 1, kp.pt.y + y + dy)) / (-2);
            J(1) = (GetPixelValue(img2, kp.pt.x + x + dx, kp.pt.y + y + dy + 1) - GetPixelValue(img2, kp.pt.x + x + dx, kp.pt.y + y + dy - 1)) / (-2);
        } else {
            // Inverse Jacobian
            // NOTE this J does not change when dx, dy is updated, so we can store it and only compute error
            // 用I1(x,y)处的梯度替换I2(x+dx,y+dy)处的梯度，使得计算得到的梯度总有意义
            J(0) = (GetPixelValue(img1, kp.pt.x + x + 1, kp.pt.y + y) - GetPixelValue(img1, kp.pt.x + x - 1, kp.pt.y + y)) / (-2);     // u方向梯度
            J(1) = (GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y + 1) - GetPixelValue(img1, kp.pt.x + x, kp.pt.y + y - 1)) / (-2);     // v方向梯度
        }
        // compute H, b and set cost;
        H += J * J.transpose();
        b += -J * error;
        cost += error * error / 2;
        // 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
```

前向法追踪角点结果如下:



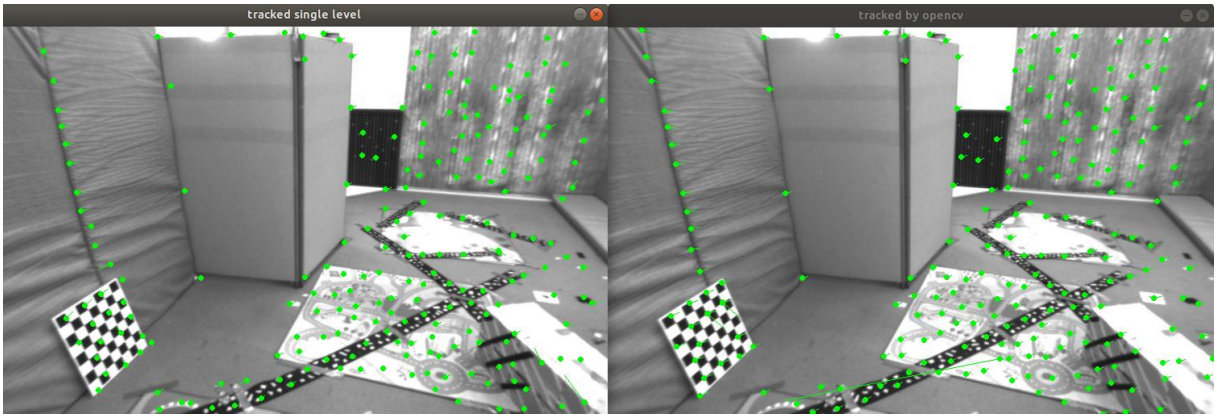
2.3 反向法

$I_2(\mathbf{W}(\mathbf{x};\Delta\mathbf{x}))$ 处的梯度可以用 $I_1(\mathbf{x})$ 处的梯度替换:

$$\frac{\partial I_2}{\partial \mathbf{W}} \approx \frac{\partial I_1}{\partial \mathbf{x}} = \left(\frac{I_1(x+1,\ y) - I_1(x-1,\ y)}{2},\ \frac{I_1(x,\ y+1) - I_1(x,\ y-1)}{2}\right)$$

由于$I_1(\mathbf{x})$ 为角点，故$I_1(\mathbf{x})$ 处的梯度总有意义。

反向法追踪角点结果如下:



2.4 推广至金字塔

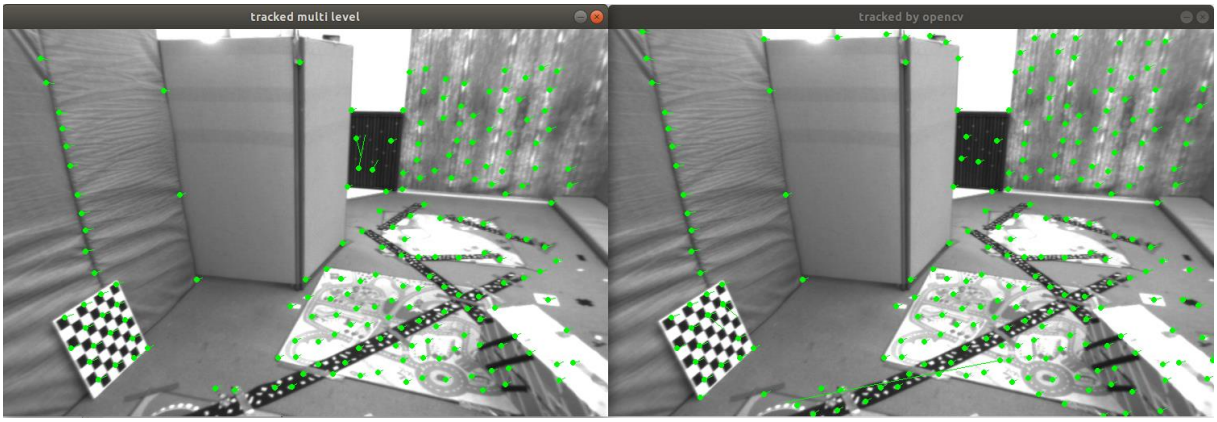
这里通过调用单层光流法来实现多层光流法，部分代码如下:

```
// create pyramids
typedef vector<KeyPoint> kpVecType;           // type of keypoint vector
vector<Mat> pyr1, pyr2;                       // image pyramids
vector<kpVecType> kp1_pyr_vect;               // kp1 pyr_vect用于从上到下存储所有金字塔内 imag1 的角点位置
// 步骤1. 从上到下存储金字塔图像 和 金字塔内 imag1 的角点位置
for (int i = 0; i < pyramids; i++) {
    Mat img1_pyr, img2_pyr;                   // 储存各层金字塔对应尺寸的图像
    // 使用函数 cv::resize(Mat, result, cv::Size(width,height));构建不同尺寸的图像
    resize(img1, img1_pyr, Size(img1.size().width * scales[pyramids - 1 - i], img1.size().height * scales[pyramids - 1 - i]));
    resize(img2, img2_pyr, Size(img2.size().width * scales[pyramids - 1 - i], img2.size().height * scales[pyramids - 1 - i]));
    pyr1.push_back(img1_pyr);
    pyr2.push_back(img2_pyr);

    kpVecType kp1_pyr;                        // kp1_pyr存储imag1内的角点对应在单层金字塔上的位置
    for(auto kp : kp1){                       // 注意: 这里对vector内成员的值进行操作，不会改变vector内的成员变量
        kp.pt *= scales[pyramids - 1 - i];
        kp1_pyr.push_back(kp);
    }
    kp1_pyr_vect.push_back(kp1_pyr);
}

//步骤2. 通过 存储的金字塔图像 和 每一层金字塔内imag1的角点位置 计算每一层的img2对应的角点位置  (coarse-to-fine LK tracking in pyramids)
kpVecType kp2_pyr;                            // kp2_pyr存储估计的imag2对应的角点 对应在各层金字塔上的位置，这个变量会在下面的for循环内不断迭代
for (int i = 0; i < pyramids; i++) {          // 注意: 对金字塔进行迭代计算的顺序为从上到下
    OpticalFlowSingleLevel(pyr1[i], pyr2[i], kp1_pyr_vect[i], kp2_pyr, success, inverse);  // 对每层金字塔使用单层光流计算出粗糙解kp2_pyr
    //将上一层求得的粗糙解 映射到下一层金字塔，作为下层金字塔迭代的初始值
    if(i < pyramids-1){
        for(auto &kp : kp2_pyr)               //注意: 这里必须对vector内成员的引用进行操作，否则无法改变vector内的成员变量
            kp.pt *= 2;
    }
}
// don't forget to set the results into kp2
for (auto &kp: kp2_pyr)
    kp2.push_back(kp);
```

利用金字塔跟踪角点结果如下:

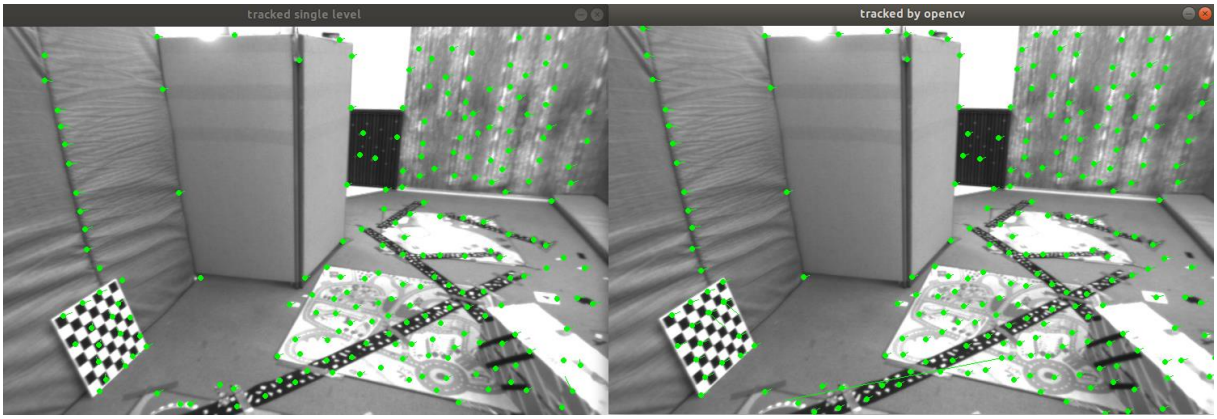


1) 所谓 coarse-to-fine 是指怎样的过程
我们使用的图像梯度往往只在局部有效，灰度值的下降在图像内较大区域看来往往是非凸的。所以当角点运动的像素距离过大时，迭代计算得到的 Δx 和 Δy往往只是局部最优解，不够精确，致使出现角点跟踪出现较大偏差，角点跟踪失败的情况。既然局部梯度无法预测长期图像灰度值走向，那么我们尝试缩小图像的尺寸。通过使用多层金字塔，上层较小尺寸图像下角点的运动较小，可顺利计算出一个较粗糙的运动，将其投影到下层金字塔时，可作为下层金字塔计算角点运动量的初始值。这样可使光流的计算在角点发生较大距离的运动时仍具有较好的鲁棒性。

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
由于特征点法中的 FAST 角点不具有尺度不变性，也就是说当图像尺度发生变化时，FAST 角点无法被重复提取。而通过构建图像金字塔，可在金字塔的每一层上检测角点，这样一来无论图像发生怎样的尺度变化，之前的 FAST 角点都可以被重复检测到。而光流法中的金字塔，是为了在计算光流时获得更好的迭代初始值，以使计算获得更好的鲁棒性，防止角点发生较大距离的运动时，角点跟踪不到的情况发生。

2.5 并行化
尝试对单层光流法使用多线程并行计算，但计算结果有误。怀疑是并行过程中存在数据依赖的问题，尝试用原子锁进行解决，但并未成功。

2.6 讨论
1) 当某个图像块在前后两帧图像中满足灰度不变假设，同一个图像块内的像素具有相同的运动，并且运动不太大时，选取的角点易于区分时，优化两个图像块之间的灰度差是合理的，否则会发生角点的误匹配。图像块的灰度变化过大的问题，可通过控制相机曝光时间解决。图像块运动过大导致误匹配，可通过图像金字塔解决。
2) 取更大的图像块会让计算更加准确一些，下面是单层光流法选取 16x16 大小图像块后的计算结果，比 8x8 的计算结果更接近 OpenCV 的计算结果。



3) 一般金字塔层数越高，计算结果越准确也越鲁棒，但代价是计算量的增加。如果金字塔层数过高，图像缩小倍数过大，会使得所有像素挤在一起，使得求得的解会过于粗糙，无法作为下一层金字塔一个好的初始值进行计算。

3. 直接法

3.1 单层直接法

1) 对于在参考图像 ref 中任意取得的一个点 $p_i = [x_i,\ y_i]^T$ 的误差$error_i$可定义为该像素点经变换矩阵$T_{cur,ref}$变换后，在当前图像 cur 内该像素所在 8x8 大小的窗口$\begin{pmatrix} x = x_i-4 \dots x_i+3 \\ y = y_i-4 \dots y_i+3 \end{pmatrix}$内所有像素和参考图像内该像素所在同样大小窗口内所有像素的光度误差的平方和:

$$error_i = \frac{1}{2}\sum_{x=x_i-4}^{x_i+3}\sum_{y=y_i-4}^{y_i+3}\left\| I_{ref}\left(\pi(p_i)+\begin{pmatrix}x\\y\end{pmatrix}\right) - I_{cur}\left(\pi\left(T_{cur,ref}p_i\right)+\begin{pmatrix}x\\y\end{pmatrix}\right)\right\|^2$$

其中误差项为

$$e(\xi) = I_{ref}\left(\pi(p_i)+\begin{pmatrix}x\\y\end{pmatrix}\right) - I_{cur}\left(\pi\left(T_{cur,ref}p_i\right)+\begin{pmatrix}x\\y\end{pmatrix}\right) = I_{ref}\left(\frac{1}{Z_{ref}}Kp_i+\begin{pmatrix}x\\y\end{pmatrix}\right) - I_{cur}\left(\frac{1}{Z_{cur}}K\exp(\xi^\wedge)\,p_i+\begin{pmatrix}x\\y\end{pmatrix}\right)$$

2) 该误差项对于自变量ξ的雅可比维度为 1×6。

对误差项左乘扰动:

$$\begin{aligned} e(\xi \oplus \delta\xi) &= I_{ref}\left(\frac{1}{Z_{ref}}Kp_i+\begin{pmatrix}x\\y\end{pmatrix}\right) - I_{cur}\left(\frac{1}{Z_{cur}}K\exp(\delta\xi^\wedge)\exp(\xi^\wedge)\,p_i+\begin{pmatrix}x\\y\end{pmatrix}\right) \\ &\approx I_{ref}\left(\frac{1}{Z_{ref}}Kp_i+\begin{pmatrix}x\\y\end{pmatrix}\right) - I_{cur}\left(\frac{1}{Z_{cur}}K(1+\delta\xi^\wedge)\exp(\xi^\wedge)\,p_i+\begin{pmatrix}x\\y\end{pmatrix}\right) \\ &= I_{ref}\left(\frac{1}{Z_{ref}}Kp_i+\begin{pmatrix}x\\y\end{pmatrix}\right) - I_{cur}\left(\underbrace{\frac{1}{Z_{cur}}K\exp(\xi^\wedge)\,p_i+\underbrace{\frac{1}{Z_{cur}}K\,\underbrace{\delta\xi^\wedge\exp(\xi^\wedge)\,p_i}_{q}}_{u}}+\begin{pmatrix}x\\y\end{pmatrix}\right) \end{aligned}$$

利用一阶泰勒展开:

$$e(\xi \oplus \delta\xi) = \underbrace{I_{ref}\left(\frac{1}{Z_{ref}}Kp_i+\begin{pmatrix}x\\y\end{pmatrix}\right) - I_{cur}\left(\frac{1}{Z_{cur}}K\exp(\xi^\wedge)\,p_i+\begin{pmatrix}x\\y\end{pmatrix}\right)}_{e(\xi)} \underbrace{-\frac{\partial I_{cur}}{\partial u}\frac{\partial u}{\partial q}\frac{\partial q}{\partial\delta\xi}}_{J}\delta\xi$$

其中$\frac{\partial I_{cur}}{\partial u}$为当前图像在$u$处的梯度:

$$\frac{\partial I_{cur}}{\partial u} = \left(\frac{I_{cur}(u_x+1,u_y)-I_{cur}(u_x-1,u_y)}{2},\ \frac{I_{cur}(u_x,u_y+1)-I_{cur}(u_x,u_y-1)}{2}\right)$$

其中$\frac{\partial u}{\partial q}$为投影方程关于相机坐标系下三维点$q = [X,Y,Z]^T$的导数为:

$$\frac{\partial u}{\partial q} = \begin{bmatrix}\frac{f_x}{Z} & 0 & -\frac{f_xX}{Z^2}\\ 0 & \frac{f_y}{Z} & -\frac{f_yY}{Z^2}\end{bmatrix}$$

$\frac{\partial q}{\partial\delta\xi}$为变换后的三维点对变换的导数:

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

合并$\frac{\partial u}{\partial q}$和$\frac{\partial q}{\partial\delta\xi}$得到$\frac{\partial u}{\partial\delta\xi}$为:

$$\frac{\partial u}{\partial\delta\xi} = \begin{bmatrix}\frac{f_x}{Z} & 0 & -\frac{f_xX}{Z^2} & -\frac{f_xXY}{Z^2} & f_x+\frac{f_xX^2}{Z^2} & -\frac{f_yY}{Z}\\ 0 & \frac{f_y}{Z} & -\frac{f_yY}{Z^2} & -f_y-\frac{f_yY^2}{Z^2} & \frac{f_yXY}{Z^2} & \frac{f_yX}{Z}\end{bmatrix}$$

最终得到的误差相对于自变量李代数ξ的雅可比为:

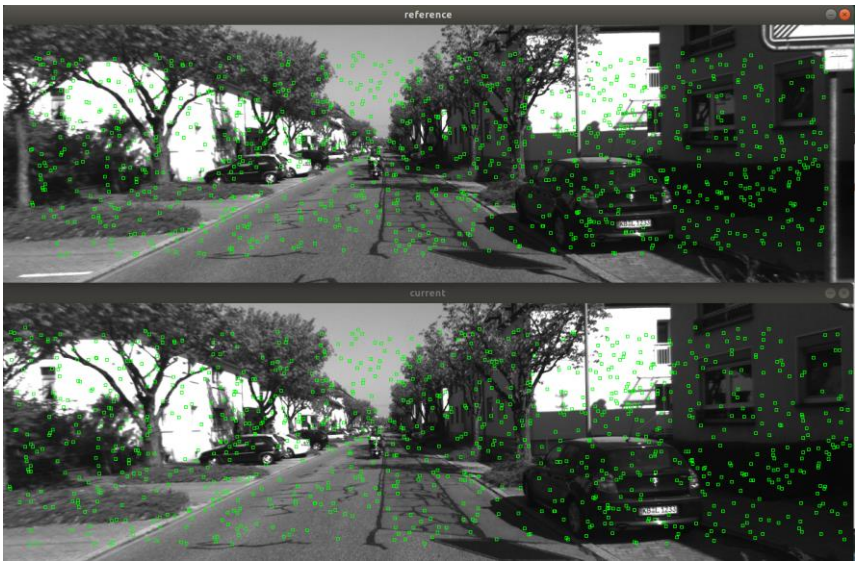
$$J = -\frac{\partial I_{cur}}{\partial u}\frac{\partial u}{\partial\delta\xi}$$

3) 窗口可以同光流法一样取 8x8 大小。不可以取单个点，因为单个点的像素没有区分度，和一个像素的灰度值相同的像素太多了，只有计算该像素周围图像块的光度差，才能很好地区分该像素点。

源文件 direct_method.cpp 位于文件夹 Direct_Method 内。估计的图像 000001.png 到 000005.png 的相机位姿计算如下:

```
jindong@jindong-virtual-machine:~/SLAM/Chap6/L6_code/Direct_Method/OUTPUT$ ./direct_method
T21 =
   0.999991  0.00242132  0.00337215 -0.00184408
-0.00242871    0.999995  0.00218895   0.0026733
-0.00336684 -0.00219713    0.999992   -0.725126
          0           0           0           1
T21 =
   0.999972  0.00137282   0.0072893  0.00740354
-0.00140116    0.999991  0.00388439 -0.00131612
 -0.0072839  -0.0038945    0.999966     -1.4707
          0           0           0           1
T21 =
   0.999937  0.00160028   0.0111304  0.00603945
-0.00165679    0.999986  0.0050703  0.00429011
 -0.0111221 -0.00508842    0.999925    -2.20745
          0           0           0           1
T21 =
    0.999827   4.5056e-05    0.0185863    -0.189007
-0.000149931    0.999984   0.00564126   0.00703802
  -0.0185858  -0.00564307    0.999811     -2.76817
           0           0           0           1
T21 =
    0.999741   0.00024302   0.0227672   -0.255295
-0.000360092    0.999987   0.00513819   0.0363201
  -0.0227656  -0.00514506    0.999728     -3.44626
           0           0           0           1
```

作为示例，这里只展示参考图像和 000001.png 的像素点匹配情况如下:



3.2 多层直接法

部分代码如下:

```
// 步骤1. 从上到下存储金字塔图像 和 金字塔内 img1 的角点位置
 double fxG = fx, fyG = fy, cxG = cx, cyG = cy;  // backup the old values

 for (int level = pyramids - 1; level >= 0; level--) {
    cv::Mat img1_pyr, img2_pyr;                          // 储存各层金字塔对应尺寸的图像
    // 使用函数 cv::resize(Mat, result, cv::Size(width,height));构建不同尺寸的图像
    resize(img1, img1_pyr, cv::Size(img1.size().width * scales[level], img1.size().height * scales[level]));
    resize(img2, img2_pyr, cv::Size(img2.size().width * scales[level], img2.size().height * scales[level]));
    pyr1.push_back(img1_pyr);
    pyr2.push_back(img2_pyr);
    VecVector2d px_ref_pyr;                      // set the keypoints in this pyramid level
    for (auto px: px_ref) {
        px_ref_pyr.push_back(scales[level] * px);
    }
    px_ref_pyr_vect.push_back(px_ref_pyr);
 }

//步骤2. 通过 存储的金字塔图像 和 每一层金字塔内img1的角点位置 计算每一层的img2对应的角点位置  (coarse-to-fine LK tracking in pyramids)
T21 = Sophus::SE3(Eigen::Matrix3d::Identity(), Eigen::Vector3d::Zero());          // 初始化估计的位姿,令R = I,  t = 0;
for (int level = pyramids - 1; level >= 0; level--) {          // 注意: 对金字塔进行迭代计算的顺序为从上到下
        // 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];
        // 对每层金字塔使用单层直接法计算出粗糙解T21
        DirectPoseEstimationSingleLayer(pyr1[pyramids - 1 - level], pyr2[pyramids - 1 - level], px_ref_pyr_vect[pyramids - 1 - level], depth_ref, T21);
        //将上一层求得的粗糙解作为下层金字塔迭代的初始值
}
// 最后恢复fx, fy, cx, cy
fx = fxG;
fy = fyG;
cx = cxG;
cy = cyG;
```

估计的图像 000001.png 到 000005.png 的相机位姿计算如下(每个位姿只取出多层直接法最后计算出的结果):

```
T21 =
   0.999991  0.00242145  0.00337204 -0.00183494
-0.00242884    0.999995  0.00218914  0.00266688
-0.00336672 -0.00219732    0.999992   -0.725144
          0           0           0           1
T21 =
   0.999972  0.00137295  0.00728909  0.00741255
-0.00140129    0.999991   0.0038846 -0.00132238
 -0.0072837 -0.00389471    0.999966     -1.4707
          0           0           0           1
T21 =
   0.999937  0.00161501   0.0111191  0.00706417
-0.00167162    0.999986  0.00508391  0.00378665
 -0.0111107 -0.00510218    0.999925    -2.20885
          0           0           0           1
T21 =
    0.999874  0.000357061   0.0158873   0.00854227
-0.000448245    0.999983  0.00573628  0.00291085
  -0.0158849 -0.00574267    0.999857     -2.9963
          0           0           0           1
T21 =
   0.999803  0.00119957   0.0198238    0.018916
-0.00132922    0.999978  0.00652782 -0.0102634
 -0.0198155 -0.00655288    0.999782      -3.793
          0           0           0           1
```

3.3 并行化

尝试对单层直接法使用多线程并行计算，但计算结果有误。怀疑是并行过程中存在数据依赖的问题。

3.4 * 延伸讨论

1) inverse additive: 可以将当前图像在u处的梯度$\frac{\partial I_{cur}}{\partial u}$近似为参考图像在角点$p_i$处的梯度$\frac{\partial I_{ref}}{\partial p_i}$。

composional forward: 可以将对变换矩阵李代数ξ的更新改为对仿射变换$\pi(T_{cur,ref}p_i)$的更新。

意义可能是可以减少一些计算量，但它们实质上应该是等价的。

2) 可以利用 inverse 算法: 将当前图像在u处的梯度$\frac{\partial I_{cur}}{\partial u}$近似为参考图像在角点$p_i$处的梯度$\frac{\partial I_{ref}}{\partial p_i}$，只要计算一帧图像的梯度，就可以重复给后面几帧图像使用。适用于角点运动不是特别大的情况。

3) 我们假设了图像块运动前后灰度不变，并且可以利用相同的深度进行投影。

4) 我们不取角点或线上的点也可以运行直接法，因为直接法的结果由图像内所有选取的点在变换矩阵前后的投影关系所决定，而不依赖于特征匹配，对特征点的可重复性提取不做要求。因此只要提取的点有梯度就行，而不一定要是角点。

5) 直接法优点/特点: 1. 不用计算特征点和描述子，只需要选取几个有梯度的像素点即可，所以直接法也适用于墙体等找不到很多特征点的场景。2.可通过选取的像素点构建半稠密至稠密的地图。

直接法缺点: 1. 当选取的像素点运动太大时，由于图像灰度值的非凸性，很难得到理想的优化值。2. 我们必须假设图像块运动前后灰度不变，但实际场景中无法保证。3. 为了区分像素点的不同，我们必须计算图像块的灰度值差异，而非单个像素点的灰度值差异，这加大了计算量。

4. * 使用光流计算视差

源文件位于文件夹 Disparity_From_Optical_Flow，是在第二题的基础上增加函数 GetDisparityFromOpticalFlow:

```
void GetDisparityFromOpticalFlow(
    const vector<KeyPoint> &kp1,
    const vector<KeyPoint> &kp2,
    const vector<bool> &success,
    cv::Mat & img_disparity){

    vector<float> disparity_groundtruth;
    vector<float> disparity_estimated;
    for(int i = 0; i < kp1.size(); i++){
        if(success[i] == true){
            disparity_groundtruth.push_back(img_disparity.at<uchar>(kp1[i].pt.y, kp1[i].pt.x));      //注意opencv存储图像行在前，列在后
            disparity_estimated.push_back(kp1[i].pt.x - kp2[i].pt.x);              // 给定的视差图显然是由左图特征点的位置减去右图得到的，所以我们也这样定义
        }
    }

    cout << endl << "Error between disparity_groundtruth and disparity_estimated is: " << endl;
    for(int i = 0; i < disparity_estimated.size(); i++)
        cout << "Keypoint " << i << " : "<< "disparity_groundtruth: "<< disparity_groundtruth[i]
             << ", disparity_estimated: "<< disparity_estimated[i]
             << ", error: " << disparity_groundtruth[i] - disparity_estimated[i] << endl;

}
```

我们比较通过多层光流法计算得到的视差和给定的视差图中的视差，误差如下:

```
Error between disparity_groundtruth and disparity_estimated is:
Keypoint 0 : disparity_groundtruth: 16, disparity_estimated: 2.71448, error: 13.2855
Keypoint 1 : disparity_groundtruth: 25, disparity_estimated: 24.9683, error: 0.0316772
Keypoint 2 : disparity_groundtruth: 24, disparity_estimated: 23.5099, error: 0.490051
Keypoint 3 : disparity_groundtruth: 12, disparity_estimated: 12.38, error: -0.380005
Keypoint 4 : disparity_groundtruth: 53, disparity_estimated: 1.77563, error: 51.2244
Keypoint 5 : disparity_groundtruth: 8, disparity_estimated: 9.39191, error: -1.39191
Keypoint 6 : disparity_groundtruth: 6, disparity_estimated: 5.565, error: 0.434998
Keypoint 7 : disparity_groundtruth: 9, disparity_estimated: 9.80109, error: -0.801086
Keypoint 8 : disparity_groundtruth: 13, disparity_estimated: 13.6899, error: -0.689941
Keypoint 9 : disparity_groundtruth: 10, disparity_estimated: 9.42249, error: 0.577515
Keypoint 10 : disparity_groundtruth: 12, disparity_estimated: 12.4862, error: -0.486237
Keypoint 11 : disparity_groundtruth: 26, disparity_estimated: -3.45258, error: 29.4526
Keypoint 12 : disparity_groundtruth: 9, disparity_estimated: 8.93884, error: 0.0611572
Keypoint 13 : disparity_groundtruth: 17, disparity_estimated: 16.59, error: 0.410004
Keypoint 14 : disparity_groundtruth: 16, disparity_estimated: 1.94135, error: 14.0587
Keypoint 15 : disparity_groundtruth: 28, disparity_estimated: 5.27515, error: 22.7249
Keypoint 16 : disparity_groundtruth: 9, disparity_estimated: 9.40375, error: -0.403748
Keypoint 17 : disparity_groundtruth: 20, disparity_estimated: 20.8113, error: -0.811279
Keypoint 18 : disparity_groundtruth: 27, disparity_estimated: 26.9051, error: 0.0949097
Keypoint 19 : disparity_groundtruth: 6, disparity_estimated: 6.57388, error: -0.573883
Keypoint 20 : disparity_groundtruth: 23, disparity_estimated: -9.11771, error: 32.1177
Keypoint 21 : disparity_groundtruth: 6, disparity_estimated: 6.25104, error: -0.251038
Keypoint 22 : disparity_groundtruth: 54, disparity_estimated: 9.63623, error: 44.3638
Keypoint 23 : disparity_groundtruth: 25, disparity_estimated: 4.99261, error: 20.0074
Keypoint 24 : disparity_groundtruth: 4, disparity_estimated: 3.81314, error: 0.186859
Keypoint 25 : disparity_groundtruth: 34, disparity_estimated: 25.5175, error: 8.48252
Keypoint 26 : disparity_groundtruth: 1, disparity_estimated: 0.65387, error: 0.34613
Keypoint 27 : disparity_groundtruth: 25, disparity_estimated: 25.118, error: -0.118042
Keypoint 28 : disparity_groundtruth: 25, disparity_estimated: 7.81958, error: 17.1804
Keypoint 29 : disparity_groundtruth: 13, disparity_estimated: 12.4816, error: 0.518372
Keypoint 30 : disparity_groundtruth: 23, disparity_estimated: 22.0069, error: 0.993103
Keypoint 31 : disparity_groundtruth: 6, disparity_estimated: 6.30872, error: -0.308716
Keypoint 32 : disparity_groundtruth: 8, disparity_estimated: 6.57019, error: 1.42981
Keypoint 33 : disparity_groundtruth: 23, disparity_estimated: 21.3291, error: 1.6709
Keypoint 34 : disparity_groundtruth: 16, disparity_estimated: 16.4274, error: -0.427429
Keypoint 35 : disparity_groundtruth: 16, disparity_estimated: 16.4614, error: -0.461426
Keypoint 36 : disparity_groundtruth: 13, disparity_estimated: 11.0087, error: 1.9913
Keypoint 37 : disparity_groundtruth: 25, disparity_estimated: 10.05, error: 14.95
Keypoint 38 : disparity_groundtruth: 11, disparity_estimated: 10.5886, error: 0.411438
Keypoint 39 : disparity_groundtruth: 22, disparity_estimated: 22.5464, error: -0.546356
```

我们可以看到大部分特征点的深度与给定的视差图中的深度相符合(error < 2)，但是有部分特征点的深度明显异于给定的深度图(error > 10)。

参考文献:

[1] Lucas-Kanade 20 Years On: A Unifying Framework