天津大学

Optical flow estimation：
lucas kanade

学　　院 智能与计算学部

专　　业 软件工程

学　　号 3020001267

姓　　名 王旭

目录

实验目标

利用课上学习的 Lucas Kanade 方法计算稀疏光流，

（1）加载一段视频。

（2）检测其中的特征点，使用 Lucas-Kanade 光流迭代跟踪这些点，并绘制线段。

分别使用 opencv 官方提供的库函数以及自行实现 LK 光流计算。

1 程序

1.1 稀疏光流估计 opencv 调库实现

A．opencv 函数接口简介：

```
void cv::goodFeaturesToTrack(
        InputArray image,
        OutputArray corners,
        int maxCorners,
        double qualityLevel,
        double minDistance,
        InputArray mask = noArray(),
        int blockSize = 3,
        bool useHarrisDetector = false,
        double k = 0.04
);
```

这个函数输出一个适合进行目标跟踪的角点。Image 是输入要进行角点检测的图像、检车到的角点保存在 corners，是一个 N*2 的数组，maxCorners 只是角点数量的上限，qualityLevel 指示角点的质量，minDistance 指示角点之间的最小距离，mask 是一个掩码指示进行角点检测的区域，blockSize 是计算角点的窗口大小、useHarrisDetector 指示是否使用 harris 角点检测，k 是角点检测的 k 值。

```
void cv::calcOpticalFlowPyrLK(
        InputArray prevImg,
        InputArray nextImg,
        InputArray prevPts,
        InputOutputArray nextPts,
        OutputArray status,
        OutputArray err,
        Size winSize = Size(21, 21),
        int maxLevel = 3,
        TermCriteria criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01),
        int flags = 0,
        double minEigThreshold = 1e-4
);
```

这个函数用于稀疏光流估计。prevImg 是前一幅图像、nextImg 是后一幅图像、prevPts 是前一幅图像中的跟踪点（以 vector<Point2f>的形式提供），nextPts 是运算得到的后一幅图像的跟踪点、status 指示跟踪点的质量、err 是跟踪误差。这个函数其实用的不是最初始的 lucas kanade 算法，而是引入了图像金字塔的改进结果，maxLevel 指示的就是金字塔的层数，criteria 是迭代求解的终止条件。

B．程序片段（带注释）：

```
int main()
{
    // 加载视频
    VideoCapture cap("D:\\study\\CV\\assignment3\\testVideo.mp4");

    if (!cap.isOpened())
    {
        cerr << "Failed to open video file." << endl;
        return -1;
    }
```

```
    Mat old_frame, old_gray;
    vector<Point2f> p0, p1;

    // 获取第一帧并转为灰度图像
    cap >> old_frame;
    cvtColor(old_frame, old_gray, COLOR_BGR2GRAY);

    goodFeaturesToTrack(old_gray, p0, 100, 0.3, 7, Mat(), 7, false, 0.04);
    // 创建一个用来画线的模板，线段将在这个模板上画出来
    Mat mask = Mat::zeros(old_frame.size(), old_frame.type());


    // 使用Lucas-Kanade光流算法跟踪特征点并绘制线段
    while (true)
    {
        Mat frame, frame_gray;
        cap >> frame;
        if (frame.empty())
        {
            break;
        }

        // 转为灰度图像
        cvtColor(frame, frame_gray, COLOR_BGR2GRAY);

        vector<uchar> status;
        vector<float> err;
        TermCriteria criteria = TermCriteria((TermCriteria::COUNT)+(TermCriteria::EPS),
10, 0.03);
        calcOpticalFlowPyrLK(old_gray, frame_gray, p0, p1, status, err, Size(15, 15),
2, criteria);

        vector<Point2f> good_new;
        for (uint i = 0; i < p0.size(); i++)
        {
            // 只有状态良好的点会被接着跟踪
            if (status[i] == 1) {
                good_new.push_back(p1[i]);
                // 在模板上画出线段,在原图上将点画出来
                line(mask, p1[i], p0[i], Scalar(0, 0, 255), 2);
                circle(frame, p1[i], 5, Scalar(0, 0, 255), -1);
            }
```

```
        }
        Mat img;
        //将模板与图像相加得到有跟踪轨迹的图像
        add(frame, mask, img);
        imshow("Frame", img);



        // 按ESC键退出
        if (waitKey(10) == 27)
        {
            break;
        }
        old_gray = frame_gray.clone();
        p0 = good_new;

    }

    // 释放资源
    cap.release();
    destroyAllWindows();
    return 0;


}
```

C. 程序说明:

程序首先读入一张图像，进行角点检测后，记录这些要被跟踪的角点，随后进入循环，每读入一帧新的图片后，就利用两帧图片，以及被跟踪点进行稀疏光流估计，随后提取质量高的被跟踪点，在模板上将这些点的上一帧图片的位置与光流估计的移动后的点的位置作为线段的起始来画线。并且对光流估计的点的位置画圈，随后将这些估计的点作为新一轮循环的被跟踪点。重复执行上述循环直到图像所有帧都被处理。

1.2 自行实现 lucas kanade

A．程序片段

```
void LK(Mat& old_gray, Mat& next_gray, vector<Point2f>& p0, vector<Point2f>& p1,
cv::Size window_size) {
    int width = next_gray.cols;
    int height = next_gray.rows;
    Mat Ix(height, width, CV_32FC1);
    Mat Iy(height, width, CV_32FC1);
    Mat It(height, width, CV_32FC1);
    //cout << 1 << endl;

    Sobel(next_gray, Ix, CV_32FC1, 1, 0);
    Sobel(next_gray, Iy, CV_32FC1, 0, 1);
    //absdiff(old_gray, next_gray, It);
    It = -next_gray + old_gray;



    int window_width = window_size.width;
    int window_height = window_size.height;
    int area = window_width * window_height;
    p1.clear();

    //对每个上一帧图片的每个跟踪点，计算其偏移值
    for (auto p : p0) {
        Mat A(area, 2, CV_32FC1);
        Mat b(area, 1, CV_32FC1);
        int px = p.x, py = p.y;
        int cnt = 0;
        for (int x = px - window_width / 2; x <= px + window_width / 2; x++) {
            for (int y = py - window_height / 2; y <= py + window_height / 2; y++) {
                //cout << cnt << endl;
                if (x >= 0 && x < width&&y>=0&&y<height) {
                    //cout << Ix.at<float>(y, x);
                    A.at<float>(cnt, 0) = Ix.at<float>(y, x);
                    A.at<float>(cnt, 1) = Iy.at<float>(y, x);
                    b.at<float>(cnt, 0) = (float)It.at<uchar>(y, x);
                }else {
                    A.at<float>(cnt, 0) = 0;
                    A.at<float>(cnt, 1) = 0;
                    b.at<float>(cnt, 0) = 0;
                }
                cnt++;

            }
```

```
        }
        Mat result = (A.t() * A).inv() * A.t() * b;
        float u = result.at<float>(0, 0),v= result.at<float>(1, 0);
        p1.push_back(Point2f(u*20 + px, v*20 + py));
    }


    }
```

B．程序说明

实现的这个函数是对 calcOpticalFlowPyrLK 函数的替代，只是没有 err、status 指示跟踪结果的误差，质量。程序对于输入的上一帧图片的跟踪点，每个运用 lucas kanade 的两个不变性假设去做运算，计算跟踪点的偏移值。

由 lucas kanade 的两个不变性推导可得如下方程：对于每个点的速度矢量 u、v 可以由这个方程解出。

$$\begin{bmatrix} I_x(\mathbf{p_1}) & I_y(\mathbf{p_1}) \\ I_x(\mathbf{p_2}) & I_y(\mathbf{p_2}) \\ \vdots & \vdots \\ I_x(\mathbf{p_{25}}) & I_y(\mathbf{p_{25}}) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(\mathbf{p_1}) \\ I_t(\mathbf{p_2}) \\ \vdots \\ I_t(\mathbf{p_{25}}) \end{bmatrix}$$

两个未知数，有不只两个方程，也就是去计算形如$Ax = b$的最小二乘解。上式的最小二乘解为$x = (A^T * A)^{-1} * A^T * b$,

A 即为每个跟踪点周围一个窗口的所有像素的 x、y 方向梯度的矩阵，行为此窗口像素的数量，列 0 为 x 方向的梯度、列 1 为 y 方向梯度。

b 即为每个跟踪点周围一个窗口的所有像素的 t 方向的梯度，也就是下一帧图像减去上一帧图像。程序这一部分计算了方程所需的 x、y、t 方向的梯度：

```
Mat Ix(height, width, CV_32FC1);
Mat Iy(height, width, CV_32FC1);
Mat It(height, width, CV_32FC1);
//cout << 1 << endl;

Sobel(next_gray, Ix, CV_32FC1, 1, 0);
Sobel(next_gray, Iy, CV_32FC1, 0, 1);
//absdiff(old_gray, next_gray, It);
It = -next_gray + old_gray;
```

随后对每一个跟踪点，根据窗口大小、填充 A 矩阵、b 矩阵。

```
Mat A(area, 2, CV_32FC1);
Mat b(area, 1, CV_32FC1);
int px = p.x, py = p.y;
int cnt = 0;
for (int x = px - window_width / 2; x <= px + window_width / 2; x++) {
    for (int y = py - window_height / 2; y <= py + window_height / 2; y++) {
        //cout << cnt << endl;
        if (x >= 0 && x < width&&y>=0&&y<height) {
            //cout << Ix.at<float>(y, x);
            A.at<float>(cnt, 0) = Ix.at<float>(y, x);
            A.at<float>(cnt, 1) = Iy.at<float>(y, x);
            b.at<float>(cnt, 0) = (float)It.at<uchar>(y, x);
        }else {
            A.at<float>(cnt, 0) = 0;
            A.at<float>(cnt, 1) = 0;
            b.at<float>(cnt, 0) = 0;
        }
        cnt++;

    }
}
```

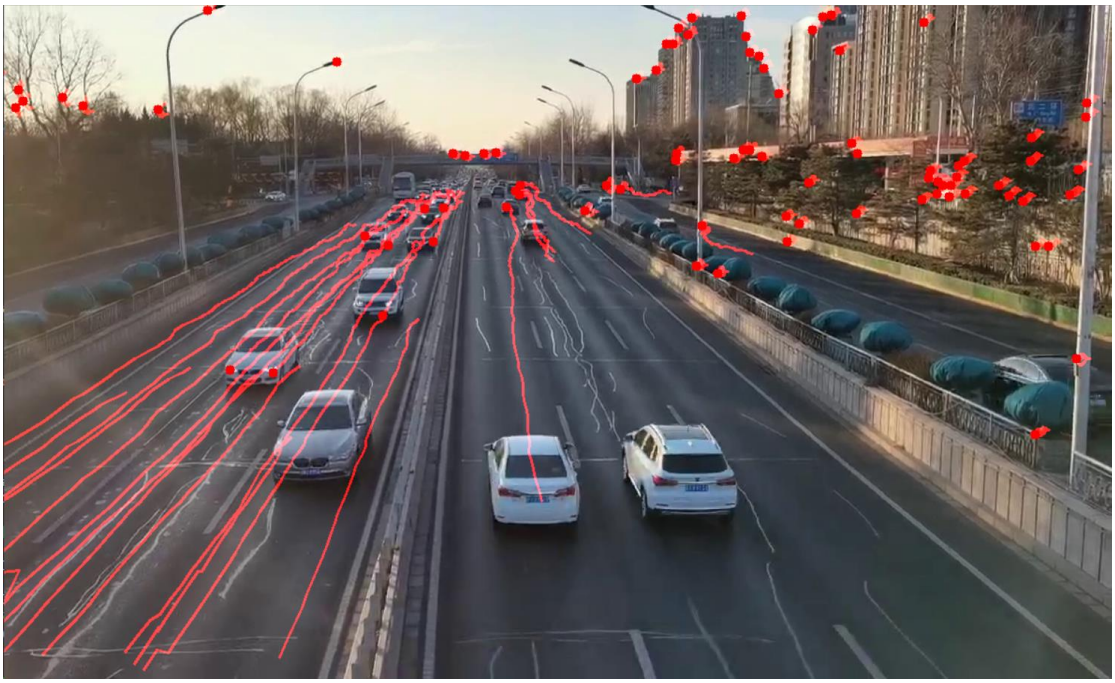
最后使用 opencv 提供的矩阵运算，计算得到跟踪点的偏移值。

```
Mat result = (A.t() * A).inv() * A.t() * b;
float u = result.at<float>(0, 0),v= result.at<float>(1, 0);
```

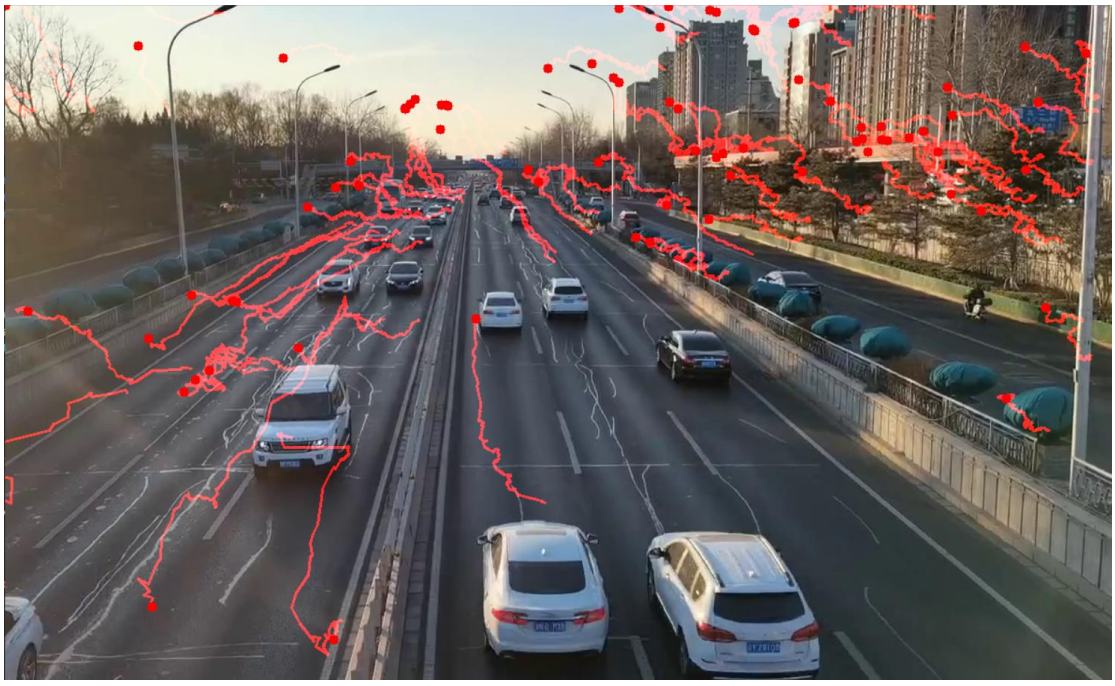
2 结果与讨论

2.1 实验结果

A． 稀疏光流估计调库实现结果



B．手写光流估计实现效果



C．讨论

可以看到调库实现的效果很好，远远好于自己手写实现的光流估计，是因为调库实现的稀疏光流估计是改进过的，引入了图像金字塔。而且还对于质量不高的跟踪点有退出机制，自己手写的光流估计效果当然就没有这么好了，不过也可以

大致反应物体的移动情况。更多效果详见截屏。