Xiangyi Cheng (xxc283)

Concepts and Backgroud

This notebook is for explaining the motion estimation with gradient method. As discussed in exercise 1, the application of motion estimation is broad and useful in reality such as object tracking, detecting, 3D reconstruction and etc.. One of the most popular method is called gradient-based motion estimation. Unless using a patch and doing correlation with the search region, the gadient method is focusing on deriving the changing in mathematical approach and set several constraints to solve the motion matrix.

The transformation in two different frames can be described by Talyor expanding around the first frame as:

$$\begin{aligned} \frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} &= 0 \\ I_x u + I_y v + I_t &= 0 \end{aligned}$$

And the derivatives above can be expressed with $h = 1$ into:

$$I_x = \frac{I(x+1, y) - I(x-1, y)}{2}$$

$$I_y = \frac{I(x, y+1) - I(x, y-1)}{2}$$

I_t is simplified with only two frames as:

$$I_t = I(x, t+1) - I(x, t)$$

This problem therefore becomes a least square problem, we have a equation to solve the motion:

$$A^T W^2 A\mathbf{v} = A^T W^2 \mathbf{b}$$

where

$$\begin{aligned} A &= [\nabla I(\mathbf{x}_1), \ldots, \nabla I(\mathbf{x}_n)]^T \\ W &= \text{diag}[W(\mathbf{x}_1), \ldots, W(\mathbf{x}_n)] \\ \mathbf{b} &= -[I_t(\mathbf{x}_1), \ldots, I_t(\mathbf{x}_n)]^T \end{aligned}$$

As a Ax=b problem, we can easily solve the motion matrix v using some build-in functions in python.

However, this is only the gradient calculation in one grid while computation in different grids are needed. Therefore, a function needs to be developed to fill all the motion information into the matrices. With these matrices, we are able to plot the optical flow.

Implement

The example below demonstrates the gradient method for displaying the optical flow field by solving the least square problem with the motion gradient constraint equation.

At the first beginning, several libraries which will be used is imported.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io
```

The image series are read in using io.imread. The first frame is called I1 and the next frame is called I2. The motion is about to obtain by comparing and calculating based on these two frames.

In [2]:
```
seq1 = {'I1': io.imread('data/image/seq1/frame1.png', as_grey=True),
        'I2': io.imread('data/image/seq1/frame3.png', as_grey=True),
        'U' : np.loadtxt('data/flow/seq1/flow3.u', dtype='double',
                         delimiter=','),
        'V' : np.loadtxt('data/flow/seq1/flow3.v', dtype='double',
                         pdelimiter=',')}

rubic = {'I1':io.imread('data/rubic/rubic.0.png', as_grey=True),
         'I2':io.imread('data/rubic/rubic.5.png', as_grey=True)}

sphere= {'I1': io.imread('data/sphere/sphere.1.png', as_grey=True),
         'I2': io.imread('data/sphere/sphere.3.png', as_grey=True)}
```

Discrete Derivative Components Computation

The image is seperated into grids and a self-defined function is developed to calculate the derivative components, I_x,I_y and I_t, on each grid location based on

$$I_x = \frac{I(x+1,y) - I(x-1,y)}{2}$$
$$I_y = \frac{I(x,y+1) - I(x,y-1)}{2}$$
$$I_t = I(x,t+1) - I(x,t)$$

Several boundary conditions should be satisfied to aviod the value going beyond the reality.

In [3]:
```
def derivative_components(I1, I2, x_grid, y_grid):
    h_img= I1.shape[0]
    w_imd= I1.shape[1]
    x_grid = int(x_grid)
    y_grid = int(y_grid)

    if x_grid>0 and x_grid< (w_imd-1) and y_grid>=0 and y_grid<h_img:
        Ix=(I1[y_grid, x_grid+1] - I1[y_grid, x_grid-1])/2;
    else:
        Ix=0

    if x_grid>=0 and x_grid<w_imd and y_grid>0 and y_grid< (h_img-1):
        Iy=(I1[y_grid+1, x_grid] - I1[y_grid-1, x_grid])/2;
    else:
        Iy=0

    if x_grid>=0 and x_grid<w_imd and y_grid>=0 and y_grid<h_img:
        It=I2[y_grid,x_grid] - I1[y_grid,x_grid]
    else:
        It=0

    return (Ix, Iy, It)
```

Matrix A and Vector b Calculation & Equation Solving for one grid

To calculate the motion matrix v, it should satisfy:

$$A^T W^2 A v = A^T W^2 b$$

After simplifying, the equation looks like:

$$Av = b$$

where

$$A = \begin{bmatrix} I_x(x_1) & I_y(x_1) \\ \dots & \\ I_x(x_n) & I_y(x_n) \end{bmatrix}$$

$$b = -\begin{bmatrix} I_t(x_1) \\ \dots \\ I_t(x_n) \end{bmatrix}$$

And the v matrix can be obtained by np.linalg.lstsq() which returns the least-squares solution to a linear matrix equation. The returned result in this case is called motion_estimation which is the v matrix.

In [4]:
```
def solve_motion_gradient_equation(I1,I2,x_grid,y_grid,width,disx,disy):


    A = np.zeros((width*width, 2))
    b = np.zeros(width*width)



    # obtain A and b from I matrix
    for i in range(0, width*width):

        x_grid_m= x_grid+disx[i]
        y_grid_m= y_grid+disy[i]

        Ix, Iy, It = derivative_components(I1, I2, x_grid_m, y_grid_m)
        A[i, 0] = Ix
        A[i, 1] = Iy
        b[i] = -It

    motion_matrix = np.linalg.lstsq(np.matmul(A.T, A),
                                    np.matmul(A.T, b))

    motion_estimation = motion_matrix[0]

    return motion_estimation
```

Motion Estimation for the whole image

The motion_estimation obtained above is only for one grid. However, the image consists of numerous of grids. One iteration is needed to fill all the motion vector into matrices that express the motion in the whole image.

In [5]:
```
def estimate_flow(img_series):
    # get the image size.
    h_img = img_series['I1'].shape[0]
    w_img= img_series['I1'].shape[1]

    # define the grid size
    grid_size = 9
    width  = 21

    # x, y are the locations of the grids.
    x = np.arange(0, w_img-grid_size, grid_size) + np.floor(grid_size/2)
    y = np.arange(0, h_img-grid_size, grid_size) + np.floor(grid_size/2)


    x_grid, y_grid = np.meshgrid(x,y);


    # get the height and width of the grid
    h_grid= x_grid.shape[0]
    w_grid= x_grid.shape[1]

    U = np.zeros((h_grid, w_grid))
    V = np.zeros((h_grid, w_grid))

    grid_center = np.arange(0, width) - np.floor(width/2)
    disx, disy = np.meshgrid(grid_center, grid_center)
    disx = disx.reshape(width*width, 1)
    disy = disy.reshape(width*width, 1)


    # get the U,V matrix
    for i in range(0, h_grid):
        for j in range(0, w_grid):
            # x_grid[i,j] is the x coordinate; y_grid[i,j] is the y coor
            motion_estimation =
            solve_motion_gradient_equation(img_series["I1"],
                                           img_series["I2"],x_grid[i,j],
                                           y_grid[i,j], width,disx,disy)
            U[i, j] = motion_estimat &ion[0]
            V[i, j] = motion_estimation[1]

    quiver_drawing(img_series["I1"], x, y, U, V, 5)
```

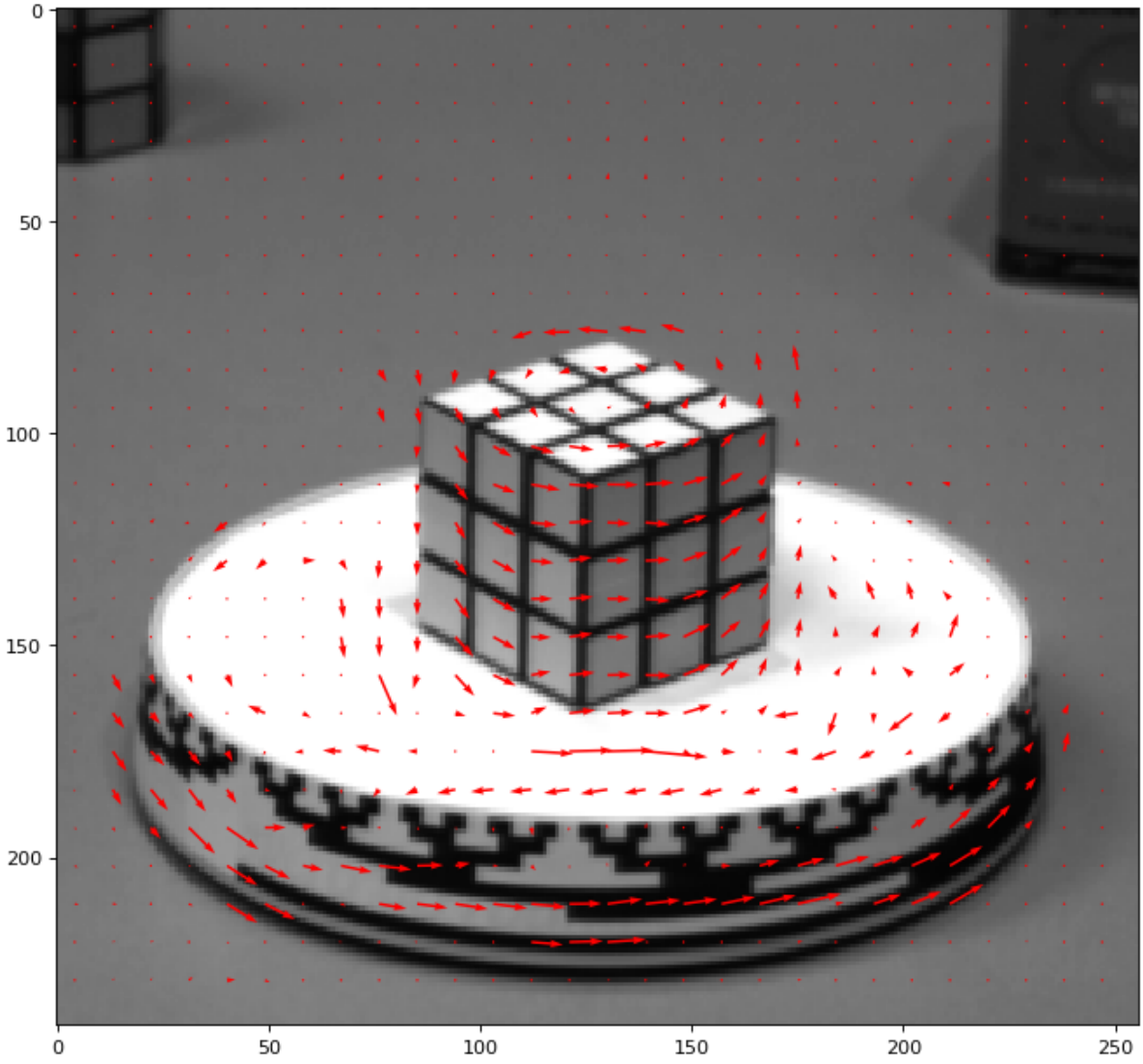
Optical Flow Field Display

A function called quiver_drawing() in the last line has not been explained yet. As its name, the function is built for drawing quivers to display the optical flow field. The basic idea behind this is simply that drawing quivers based on the grid locations, motion matrices returned back from estimate_flow() as well as a scale scalar.

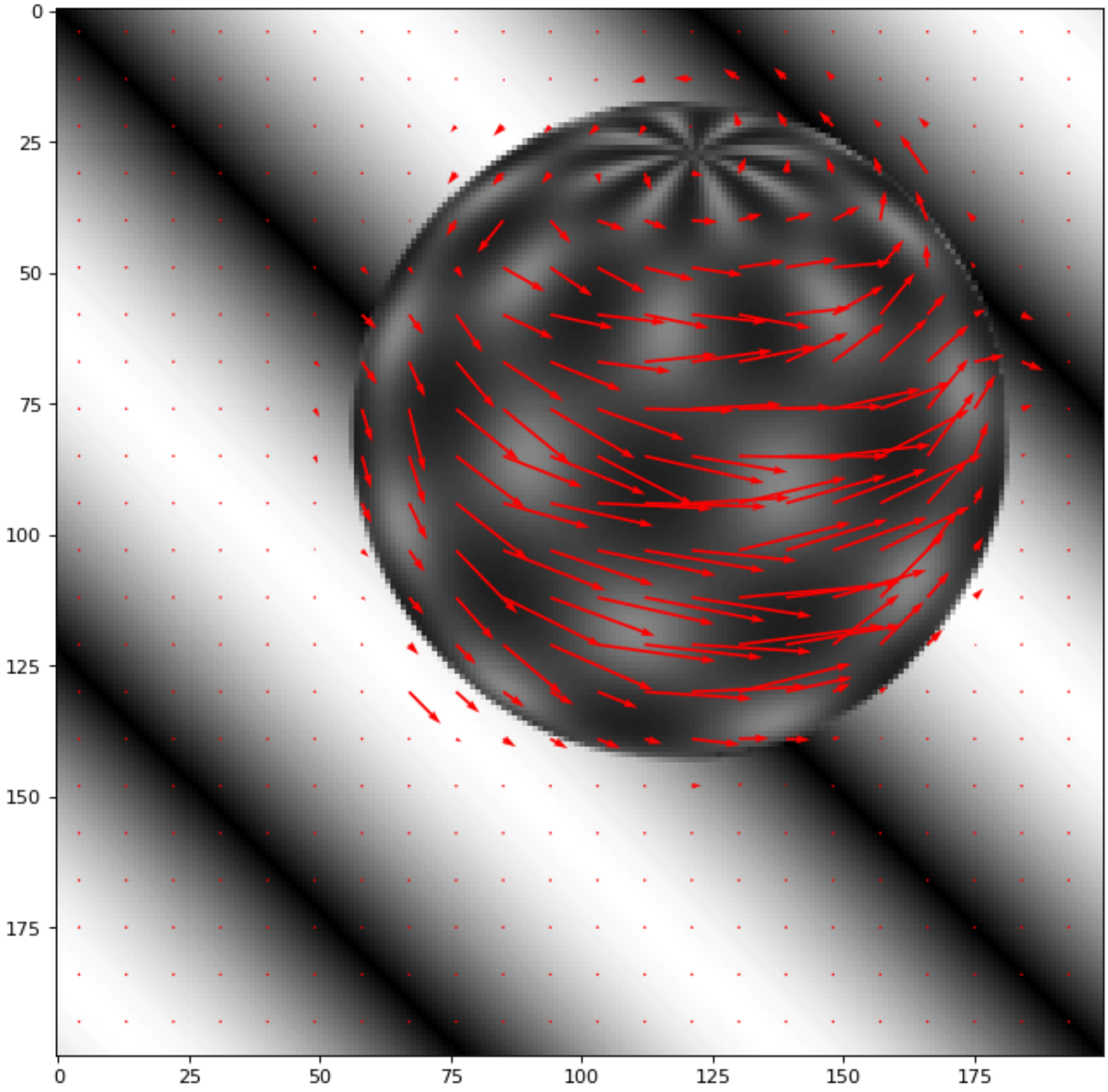
In [6]:
```
def quiver_drawing(I, x_grid, y_grid, U, V, scale):
    fig, ax = plt.subplots(figsize=(10, 10), dpi=80)
    ax.imshow(I, cmap='gray')
    ax.quiver(x_grid, y_grid, U*scale, V*scale, color='red',
              angles='xy', scale_units='xy', scale=1)
    ax.set_aspect('equal')
    plt.show()
```

Apply image series into the function to check the results.

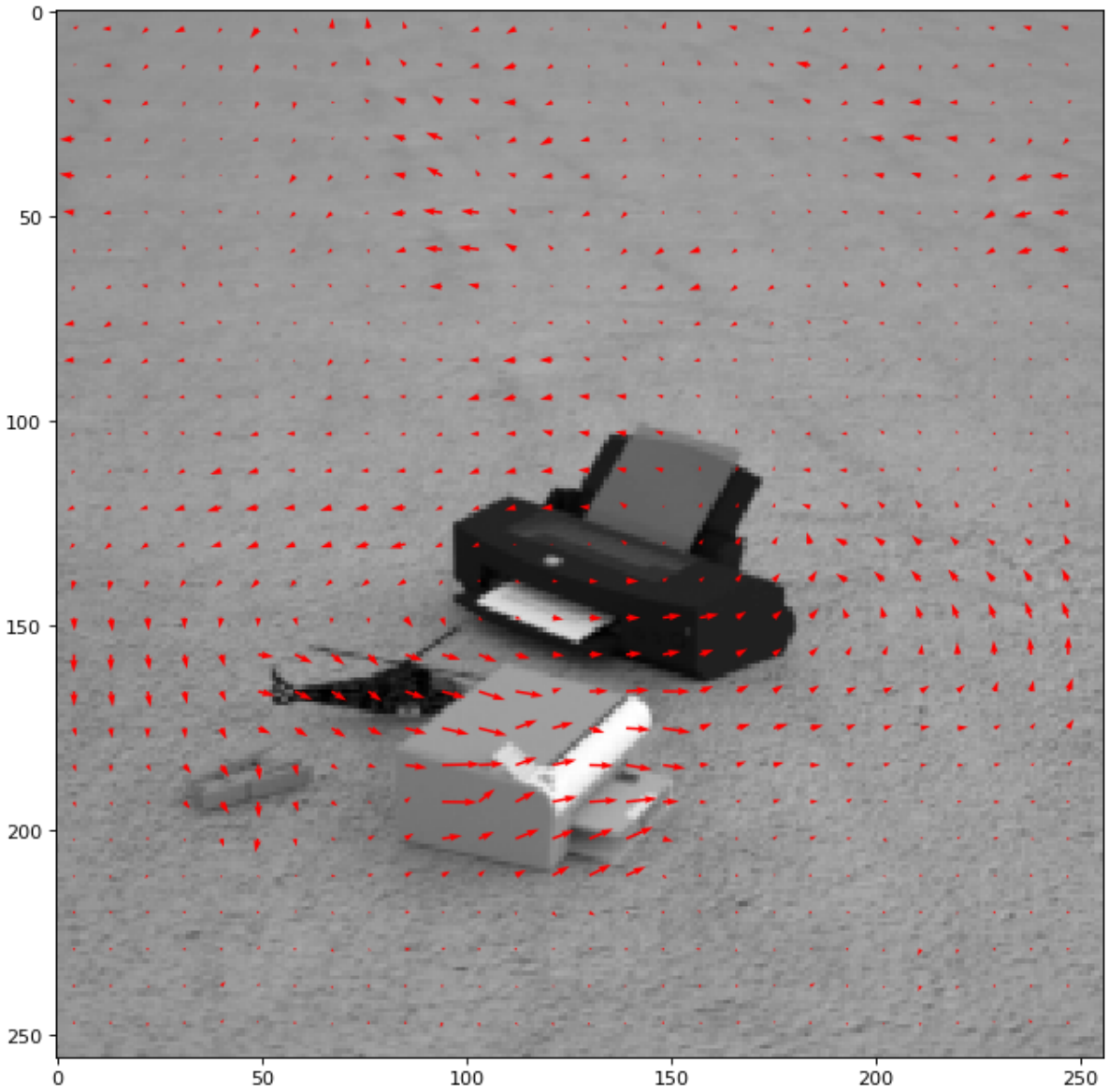
In [7]:
```
estimate_flow(rubic)
```



In [8]:
```
estimate_flow(sphere)
```

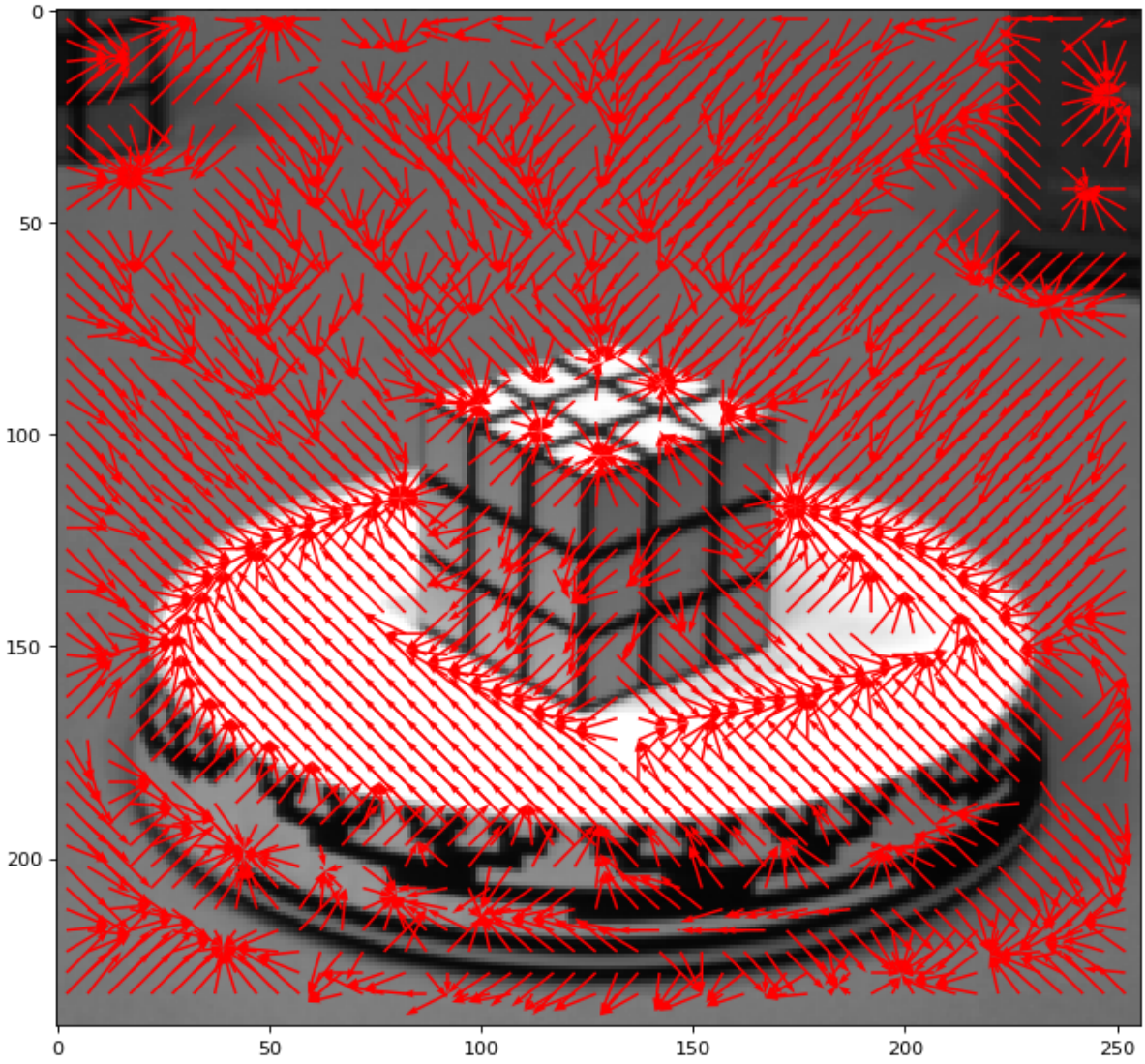


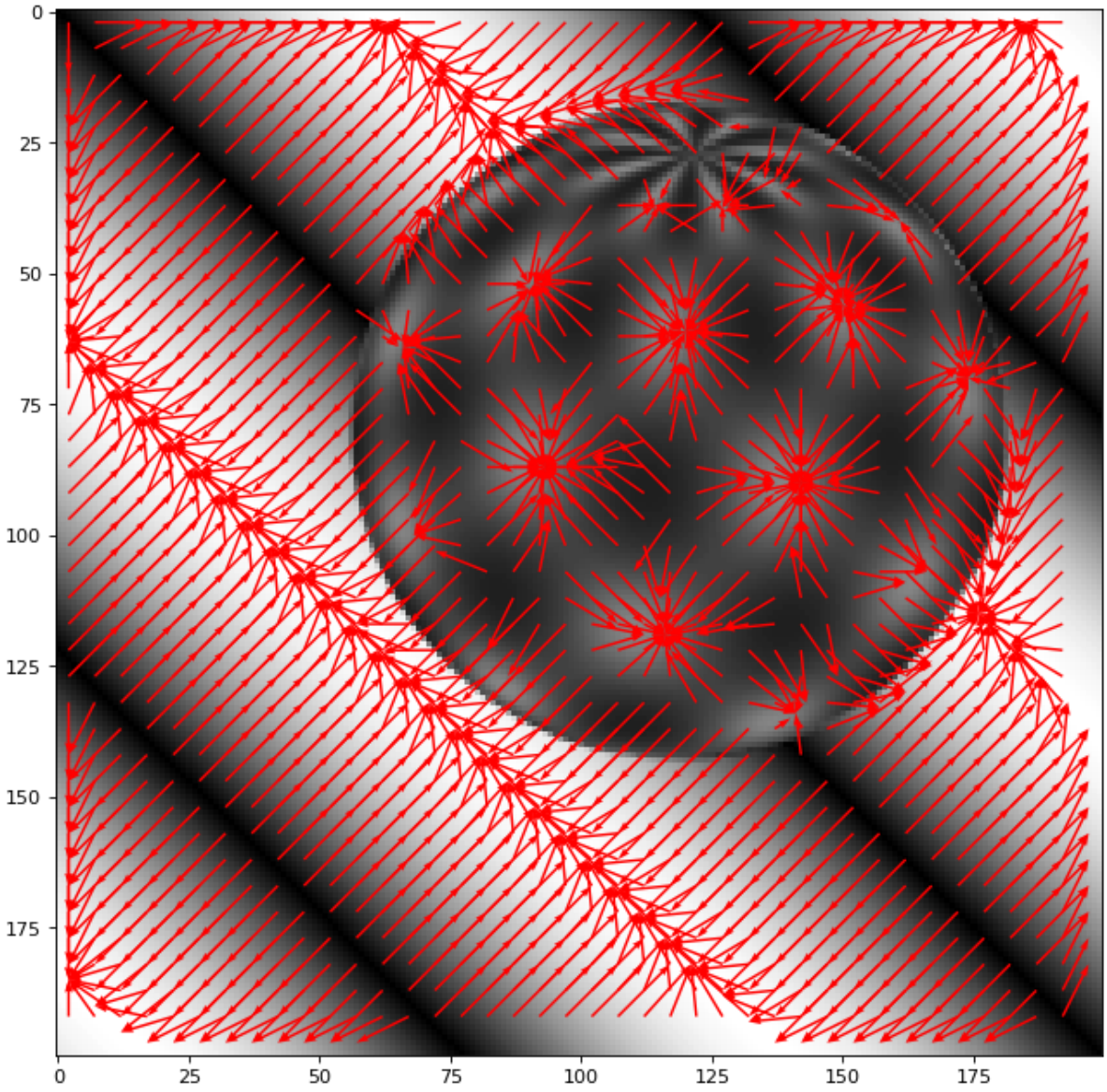
In [9]:
```
estimate_flow(seq1)
```

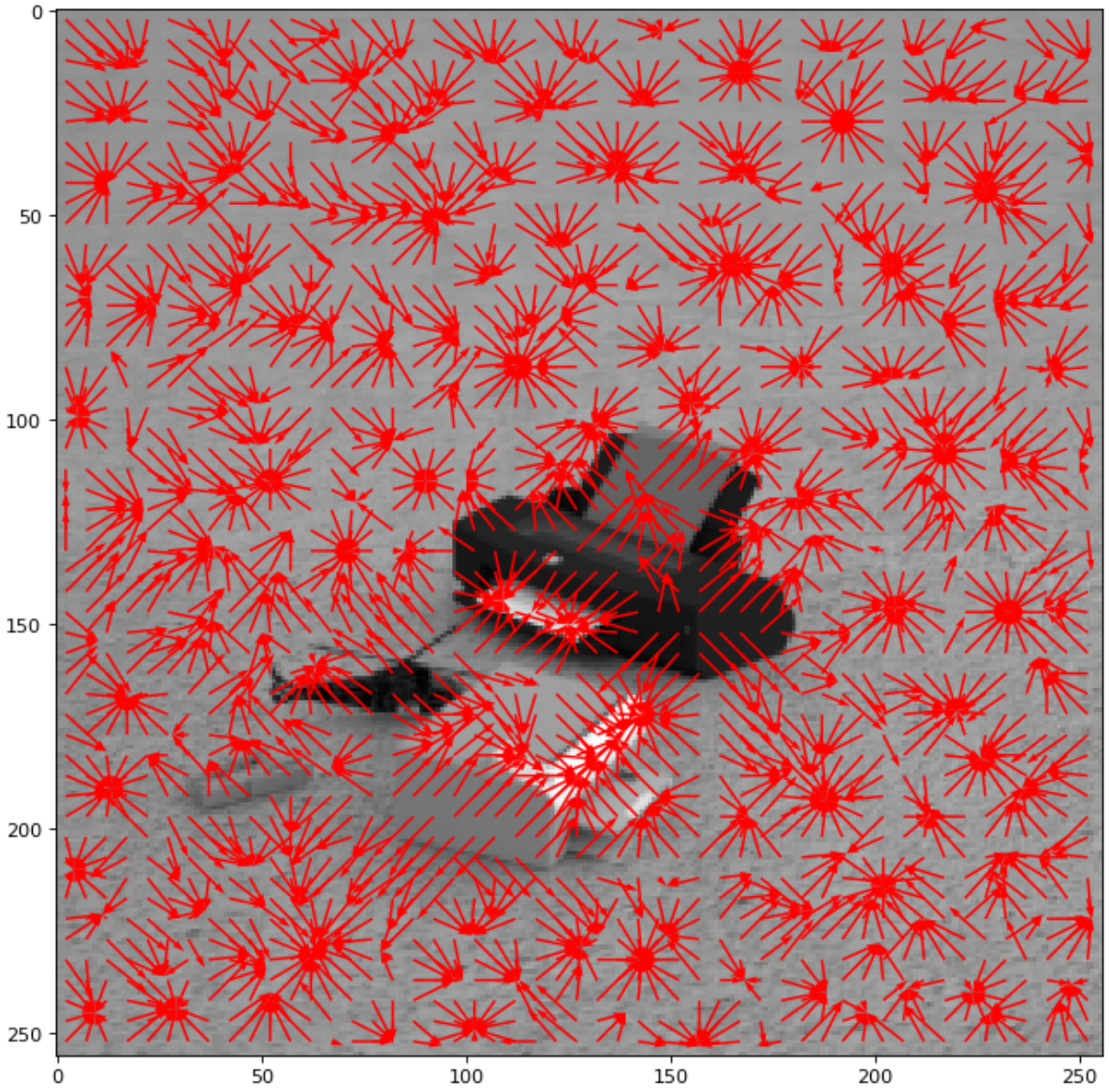


Comparison between Gradient Method and Correlation Method

The results from the correlation method are:


0
50
100
150
200
0
50
100
150
200
250


0
25
50
75
100
125
150
175
0
25
50
75
100
125
150
175



Compare these three optical flow field images with the results using gradient and we can say that the gradient method definitely works better. It is not saying that correlation method is worse than gradient method in general. We have to admit that correlation method is outstanding in some cases but not in these specific ones.

Exploration

From the results above, the estimated motion is meaning and does reflect the correct direction of the movement. However, the background sometimes is meaningless to be detected and wastes the computation storage while people mostly focus on the objects. How about using the edge detector to highlight the edge and then apply the gradient method to estimate the motion? The code shown below is a demonstration for this idea.

Take the rubic image series for an example. Use canny edge detector with different threshold to do the image pre-processing.

In [16]:
```
import cv2

rubic_start=cv2.imread('data/rubic/rubic.0.png')
rubic_end=cv2.imread('data/rubic/rubic.5.png')

# canny edge detector, set different
# lower and upper thresholds for the detector.
rubic_s300=cv2.Canny(rubic_start,300,300)
rubic_e300=cv2.Canny(rubic_end,300,300)
rubic_s400=cv2.Canny(rubic_start,400,400)
rubic_e400=cv2.Canny(rubic_end,400,400)
cv2.imwrite('rubic_s300.png',rubic_s300)
cv2.imwrite('rubic_e300.png',rubic_e300)
cv2.imwrite('rubic_s400.png',rubic_s400)
cv2.imwrite('rubic_e400.png',rubic_e400)
```

Out[16]:
```
True
```

Read the images in with io.image as the same as the previous step.

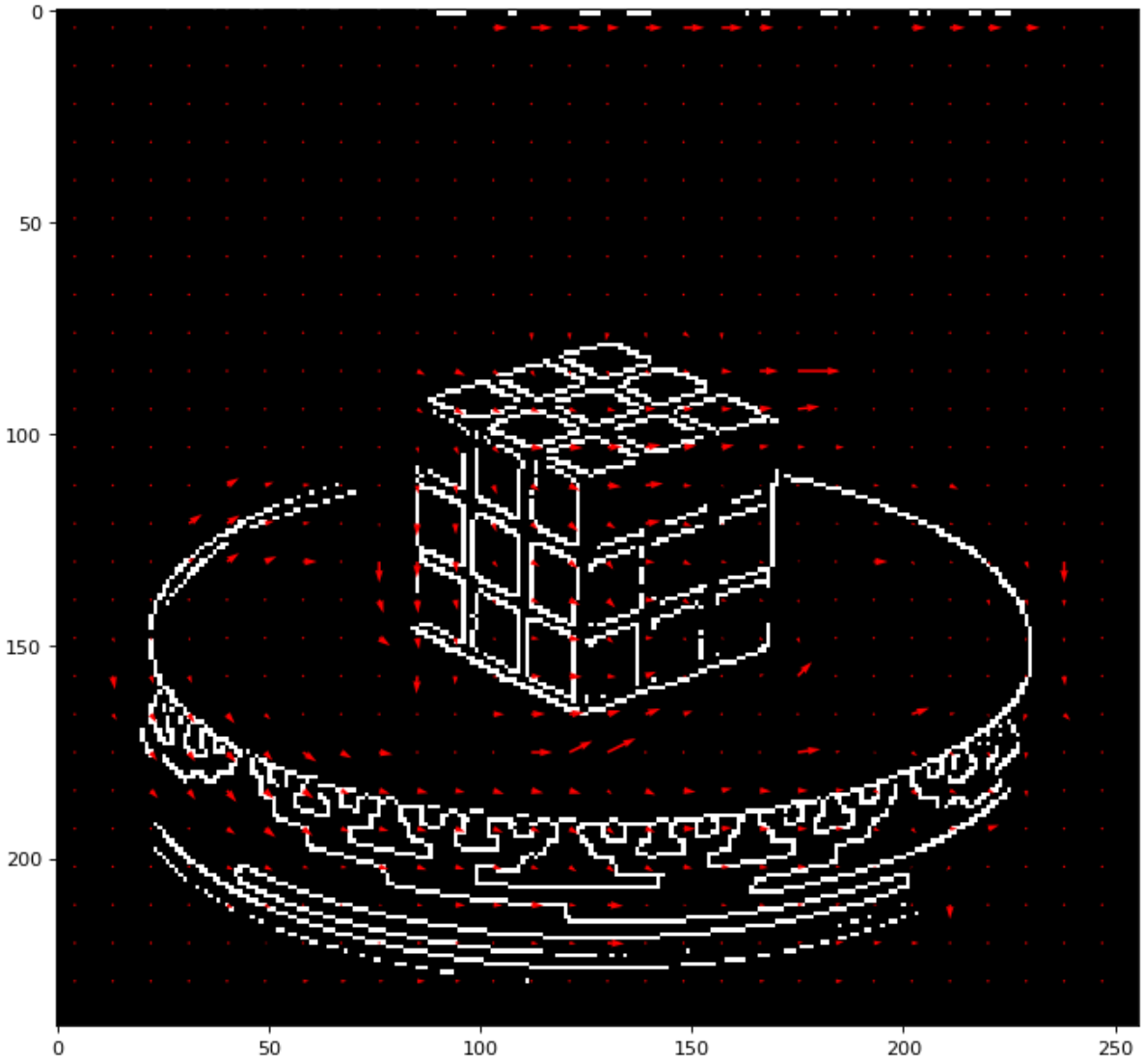
In [17]:
```
rubic300 = {'I1':io.imread('rubic_s300.png', as_grey=True),
         'I2':io.imread('rubic_e300.png', as_grey=True)}
rubic400 = {'I1':io.imread('rubic_s400.png', as_grey=True),
         'I2':io.imread('rubic_e400.png', as_grey=True)}
```

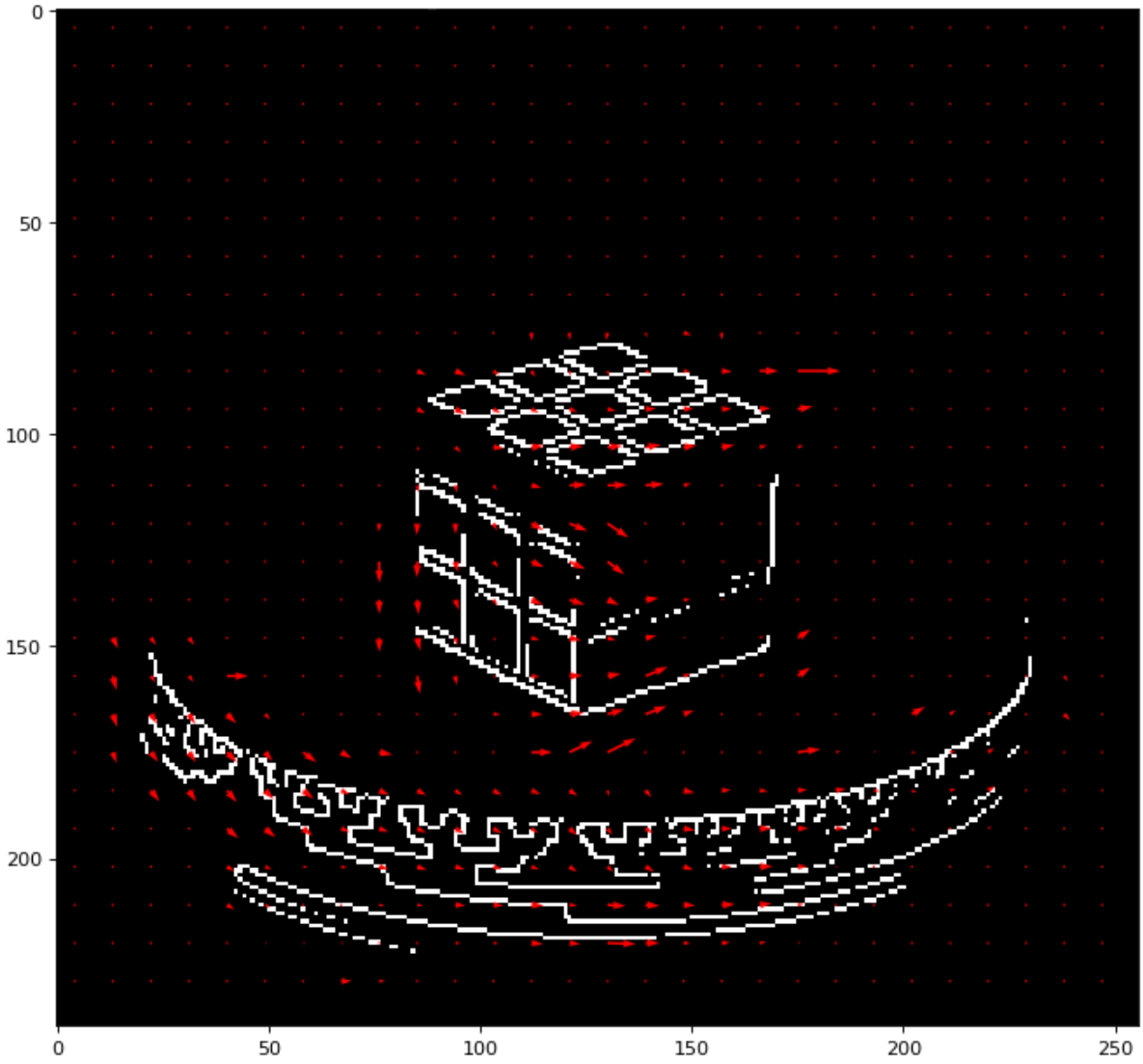
Test the image on the functions defined above.

In [18]:
```
estimate_flow(rubic300)
estimate_flow(rubic400)
```

```
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:8: Runtime
Warning: overflow encountered in ubyte_scalars

/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:13: Runtim
eWarning: overflow encountered in ubyte_scalars
  del sys.path[0]
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:18: Runtim
eWarning: overflow encountered in ubyte_scalars
```





Conclusion and Discussion

Motion is successfully estimated with gradient method which is basically to solve the motion matrix using the motion gradient constraint equation and least-square equation.The quivers correctly reflect the direction of the object movement in three different image series. Compared with the results obtained from correlation method, this optical flow field make more sense and is more realistic.

As for the exploration, although the results are not as good as the ones without applying edge detector, it illuminates how robust the gradient method is. The values of the pixels in original images are from 0 to 255 while they are binary after pre-processing. In other words, most of the information are lost during the processing. Surprisely, most of the flow are still pointing to the correct direction. Moreover, without annoying by the background, the movement of the object is shown explicitly. Also, the results are slightly different with the changing of the threshold of the canny edge detector. The interesting field is that some quivers in the black area near the object are still pointing to the correct direction. An initial thought of it is that point actually is a key point

in the second frame which didn't show directly on the first frame. I don't have time to explore more on this, but a way to solve this is to circle this key point on the second frame out on the first frame. Therefore, we would have a straightforward view of this point.