

OpenCV Tutorial 9 - Chapter 10

Author: Noah Kuntz (2009)

Contact: nk752@drexel.edu

Keywords: OpenCV, computer vision, image processing, optic flow, tracking, motion

My Vision Tutorials Index

This tutorial assumes the reader:

- (1) Has a basic knowledge of Visual C++
- (2) Has some familiarity with computer vision concepts
- (3) Has read the previous tutorials in this series

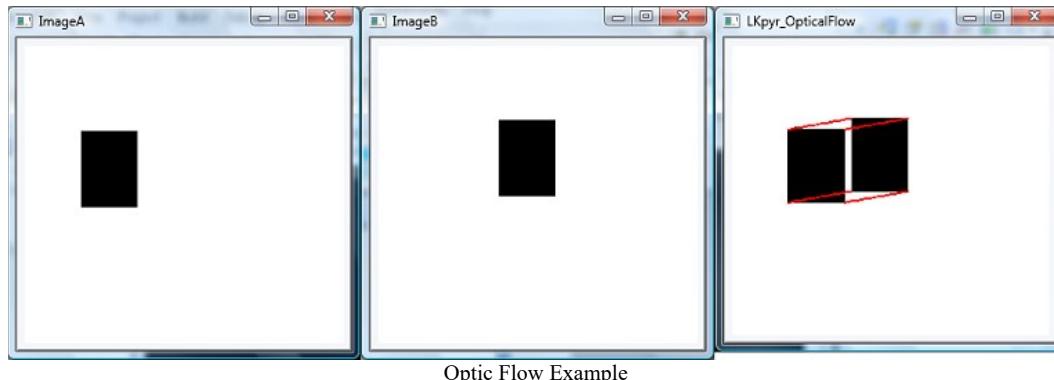
The rest of the tutorial is presented as follows:

- [Step 1: Optic Flow](#)
- [Step 2: Kalman Filter](#)
- [Final Words](#)

Important Note!

More information on the topics of these tutorials can be found in this book: [Learning OpenCV: Computer Vision with the OpenCV Library](#)

Step 1: Optic Flow



Optic Flow Example

Optic flow is a useful tool for tracking objects in motion. The Lucas-kanade algorithm is easily the most popular method of tracking movement. The LK tracker uses three assumptions, brightness constancy between the same pixels from one frame to the next, small movements between frames (requiring image pyramids to track larger movements), and spatial coherence, that points near each other are on the same surface. Then the basic concept of the tracker is to estimate the velocity of a moving pixel by the ratio of the derivative of the intensity over time divided by the derivative of the intensity over space. Check the book and the copious research on Lucas-Kanade for more technical detail of the method. The functions we use to perform this tracking are `cvGoodFeaturesToTrack` to find the features, `cvFindCornerSubPix` to refine their location, and `cvCalcOpticalFlowPyrLK` to do the actual optic flow calculations. And then lastly we can draw the flow vectors on a combination of the two images to show that it works. Here is the code:

```
const int MAX_CORNERS = 500;

int _tmain(int argc, _TCHAR* argv[])
{
    // Load two images and allocate other structures
    IplImage* imgA = cvLoadImage("image0.png", CV_LOAD_IMAGE_GRAYSCALE);
    IplImage* imgB = cvLoadImage("image1.png", CV_LOAD_IMAGE_GRAYSCALE);

    CvSize img_sz = cvGetSize( imgA );
    int win_size = 15;

    IplImage* imgC = cvLoadImage("OpticalFlow1.png", CV_LOAD_IMAGE_UNCHANGED);

    // Get the features for tracking
    IplImage* eig_image = cvCreateImage( img_sz, IPL_DEPTH_32F, 1 );
    IplImage* tmp_image = cvCreateImage( img_sz, IPL_DEPTH_32F, 1 );

    int corner_count = MAX_CORNERS;
```

```

CvPoint2D32f* cornersA = new CvPoint2D32f[ MAX_CORNERS ];

cvGoodFeaturesToTrack( imgA, eig_image, tmp_image, cornersA, &corner_count,
    0.05, 5.0, 0, 3, 0, 0.04 );

cvFindCornerSubPix( imgA, cornersA, corner_count, cvSize( win_size, win_size ),
    cvSize( -1, -1 ), cvTermCriteria( CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 20, 0.03 ) );

// Call Lucas Kanade algorithm
char features_found[ MAX_CORNERS ];
float feature_errors[ MAX_CORNERS ];

CvSize pyr_sz = cvSize( imgA->width+8, imgB->height/3 );

IplImage* pyrA = cvCreateImage( pyr_sz, IPL_DEPTH_32F, 1 );
IplImage* pyrB = cvCreateImage( pyr_sz, IPL_DEPTH_32F, 1 );

CvPoint2D32f* cornersB = new CvPoint2D32f[ MAX_CORNERS ];

cvCalcOpticalFlowPyrLK( imgA, imgB, pyrA, pyrB, cornersA, cornersB, corner_count,
    cvSize( win_size, win_size ), 5, features_found, feature_errors,
    cvTermCriteria( CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 20, 0.3 ), 0 );

// Make an image of the results

for( int i=0; i < 550 ){
    printf("Error is %f/n", feature_errors[i]);
    continue;
}
printf("Got it/n");
CvPoint p0 = cvPoint( cvRound( cornersA[i].x ), cvRound( cornersA[i].y ) );
CvPoint p1 = cvPoint( cvRound( cornersB[i].x ), cvRound( cornersB[i].y ) );
cvLine( imgC, p0, p1, CV_RGB(255,0,0), 2 );
}

cvNamedWindow( "ImageA", 0 );
cvNamedWindow( "ImageB", 0 );
cvNamedWindow( "LKpyr_OpticalFlow", 0 );

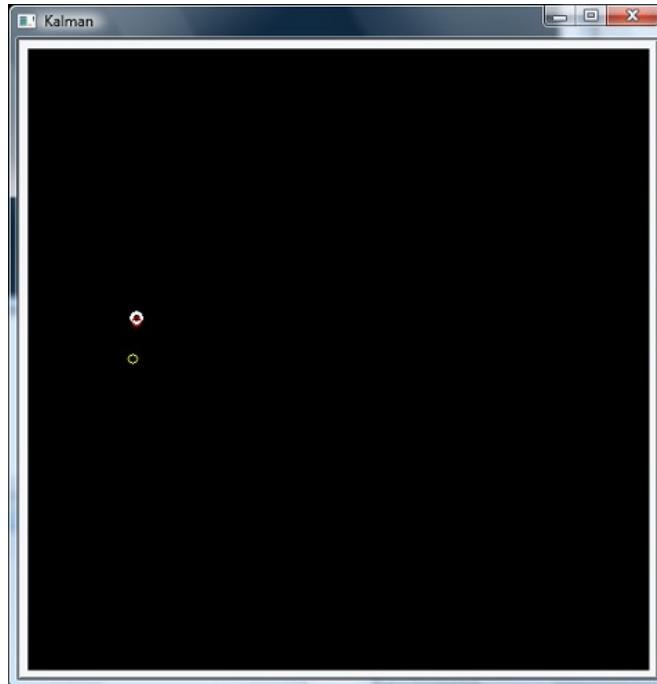
cvShowImage( "ImageA", imgA );
cvShowImage( "ImageB", imgB );
cvShowImage( "LKpyr_OpticalFlow", imgC );

cvWaitKey(0);

return 0;
}

```

Step 2: Kalman Filter



Kalman Filter Prediction Example

The Kalman filter is another powerful tool for analyzing motion. The filter can be used to predict the real position of something being tracked at a better accuracy than raw sensor data. The Kalman filter uses the history of measurements to build a model of the state of the system that maximizes the probability for the position of the target based on the past measurements. Use the book or other resources to examine the math behind this. There are good details in the comments here, so look at the code to see how to use the Kalman filter in OpenCV:

```

int _tmain(int argc, _TCHAR* argv[])
{
    // Initialize Kalman filter object, window, number generator, etc
    cvNamedWindow( "Kalman", 1 );
    CvRandState rng;
    cvRandInit( &rng, 0, 1, -1, CV_RAND_UNI );

    IplImage* img = cvCreateImage( cvSize(500,500), 8, 3 );
    CvKalman* kalman = cvCreateKalman( 2, 1, 0 );

    // State is phi, delta_phi - angle and angular velocity
    // Initialize with random guess
    CvMat* x_k = cvCreateMat( 2, 1, CV_32FC1 );
    cvRandSetRange( &rng, 0, 0.1, 0 );
    rng.disttype = CV_RAND_NORMAL;
    cvRand( &rng, x_k );

    // Process noise
    CvMat* w_k = cvCreateMat( 2, 1, CV_32FC1 );

    // Measurements, only one parameter for angle
    CvMat* z_k = cvCreateMat( 1, 1, CV_32FC1 );
    cvZero( z_k );

    // Transition matrix F describes model parameters at and k+1
    const float F[] = { 1, 1, 0, 1 };
    memcpy( kalman->transition_matrix->data.fl, F, sizeof(F) );

    // Initialize other Kalman parameters
    cvSetIdentity( kalman->measurement_matrix, cvRealScalar(1) );
    cvSetIdentity( kalman->process_noise_cov, cvRealScalar(1e-5) );
    cvSetIdentity( kalman->measurement_noise_cov, cvRealScalar(1e-1) );
    cvSetIdentity( kalman->error_cov_post, cvRealScalar(1) );

    // Choose random initial state
    cvRand( &rng, kalman->state_post );

    // Make colors
    CvScalar yellow = CV_RGB(255,255,0);
    CvScalar white = CV_RGB(255,255,255);
    CvScalar red = CV_RGB(255,0,0);

    while( 1 ){
        // Predict point position
        const CvMat* y_k = cvKalmanPredict( kalman, 0 );

        // Generate Measurement (z_k)
        cvRandSetRange( &rng, 0, sqrt( kalman->measurement_noise_cov->data.fl[0] ), 0 );
        cvRand( &rng, z_k );
        cvMatMulAdd( kalman->measurement_matrix, x_k, z_k, z_k );

        // Plot Points
        cvZero( img );
        // Yellow is observed state
        cvCircle( img,
                  cvPoint( cvRound(img->width/2 + img->width/3*cos(z_k->data.fl[0])), 
                           cvRound( img->height/2 - img->width/3*sin(z_k->data.fl[0])) ),
                  4, yellow );
        // White is the predicted state via the filter
        cvCircle( img,
                  cvPoint( cvRound(img->width/2 + img->width/3*cos(y_k->data.fl[0])), 
                           cvRound( img->height/2 - img->width/3*sin(y_k->data.fl[0])) ),
                  4, white, 2 );
        // Red is the real state
        cvCircle( img,
                  cvPoint( cvRound(img->width/2 + img->width/3*cos(x_k->data.fl[0])), 
                           cvRound( img->height/2 - img->width/3*sin(x_k->data.fl[0])) ),
                  4, red );
        cvShowImage( "Kalman", img );

        // Adjust Kalman filter state
        cvKalmanCorrect( kalman, z_k );

        // Apply the transition matrix F and apply "process noise" w_k
        cvRandSetRange( &rng, 0, sqrt( kalman->process_noise_cov->data.fl[0] ), 0 );
        cvRand( &rng, w_k );
        cvMatMulAdd( kalman->transition_matrix, x_k, w_k, x_k );
    }
}

```

```
// Exit on esc key
if( cvWaitKey( 100 ) == 27 )
    break;
}

return 0;
}
```

Final Words

This tutorial's objective was to show how to implement Lucas-Kanade optic flow and a Kalman filter for the purposes of tracking motion.

Click [here](#) to email me.

Click [here](#) to return to my Tutorials page.