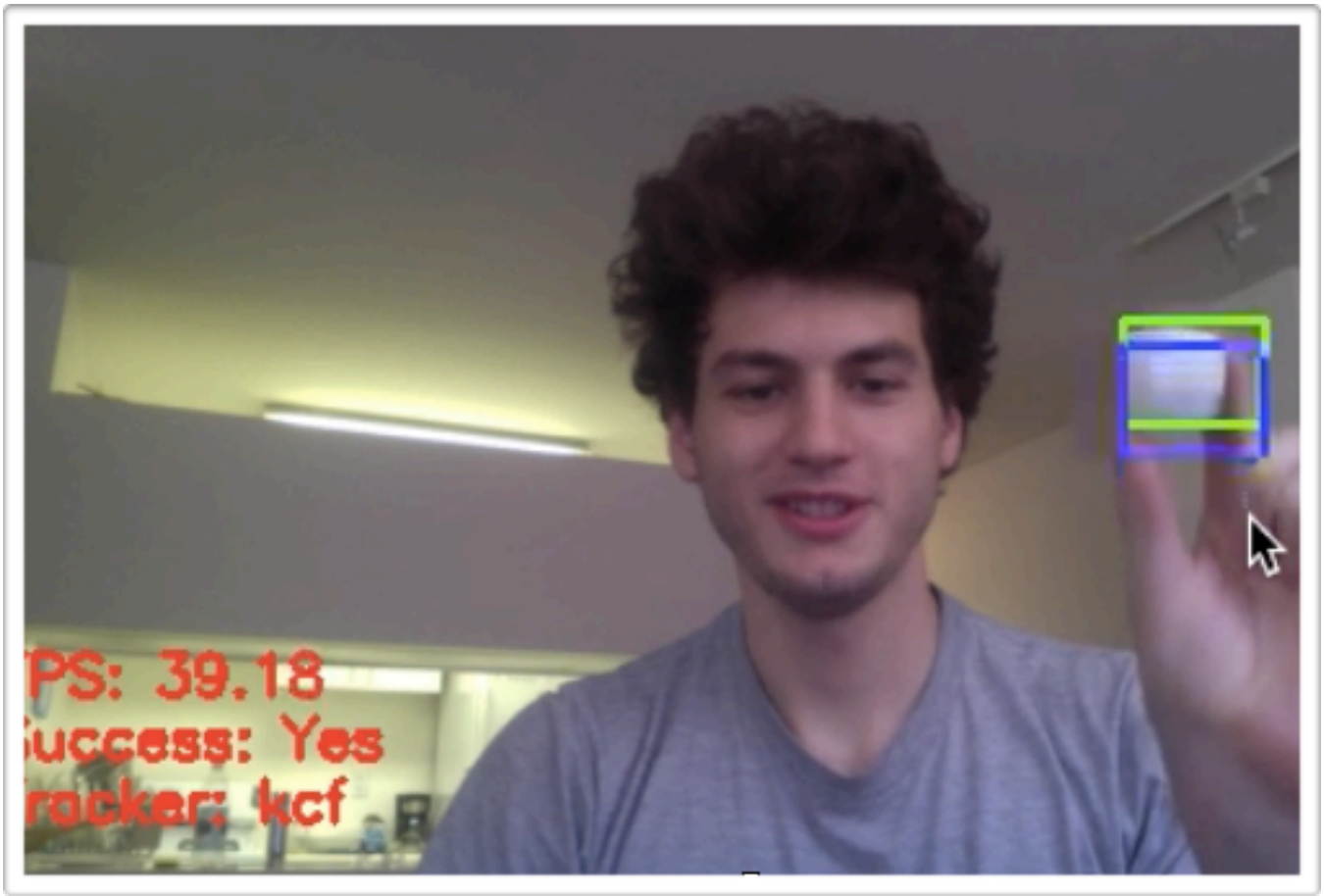
MISHA LUCZKIW



Kalman Filter Project Report

PH235: Professor Anita Raja
Prepared by: Misha Luczkiw
May 6, 2019

***Abstract* – Implementation of a Kalman filter for tracking an object with a camera using opencv software in Python. The Kalman filter should account for loss of tracking to smoothen the resulting tracking.**

MISHA LUCZKIW

SUMMARY

Objective

Implement a Kalman filter from the ground up without implementing built-in modules of opencv such as the cv.kalman module. The idea was to clearly understand the inner workings of this iterative algorithm. This project was done along side another project that utilizes tracking to focus a sound beam onto the person using it, so part of the motivation stems from that.

Goals

The end goal was to have a video feed that tracks a persons face using opencv techniques for tracking and apply a smoothing Kalman filter onto it.

Solution

A step-by-step approach was used: first a 1D Kalman filter was used to understand and outline the basic concept of a Kalman filter. Next a 2D model was implemented whereby the movement of the mouse was displayed on screen with random white Gaussian noise added onto it and the estimated true position calculated by the Kalman filter. Then the filter was applied with some modifications to the live video feed.

Outline

- Theoretical background of the Kalman Filter
- 1D examples explained
- 2D Kalman filter block diagram
- 2D Kalman filter model explained
- Live video explained

THEORETICAL BACKGROUND

History

The Kalman filter was developed in 1960 in the famous paper by Rudolf E. Kálmán. in which he describes a recursive solution to the discrete-data linear filtering problem. The Kalman filter is also known as a linear quadratic estimation that uses a series of observation over time containing statistical noise and other inaccuracies. The first use of the Kalman filter was in the nonlinear problem of trajectory estimation for the Apollo program. Since the navigation computers at the time had low computational power, an efficient algorithm was necessary and the Kalman filter is well suited for these types of applications.

The application of the Kalman filter ranges from trajectory optimization, signal processing, to econometrics. Anywhere where there is noisy data to be filtered, Kalman filters are a viable solution.

Mathematical Background

The Kalman filter is a set of mathematical equations that provides an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean of the squared error. The filter is very powerful in several aspects: it supports estimations of past, present, and even future states, and it can do so even when the precise nature of the modeled system is unknown.

The Kalman filter estimates the state $x_k = A\,x_{k-1} + B u_{k-1} + w_{k-1}$, with a measurement $z_k = H x_k + v_k$.

w_k and v_k represent the process and measurement noise (respectively). They are assumed to be independent, white, and with normal distributions with mean 0 and covariance Q and R respectively. The $n\,x\,n$ matrix A in the difference equation relates the state at the previous time step to the state at the current time step. It is basically just a matrix that makes sure that the variables in the state matrix are related correctly; depending on how the state matrix is set up the A matrix will change. The same goes for the H matrix, it just relates the measurement with the state matrix in the correct way.

The real strength of the Kalman filter lies in the Kalman gain $K = \dfrac{P_k^- H^T}{H P_k^- H^T + R}$. As the error covariance R approaches zero, the gain K weights the residual more heavily. On the other hand, as the error covariance P_k^- approaches zero, the gain K weights the residual less heavily. Basically the Kalman gain balances how much to trust the measurement data compared to the estimate data in an optimal way based on how the measurement data error is modeled.

Below a general block diagram of the Kalman filtering is shown:

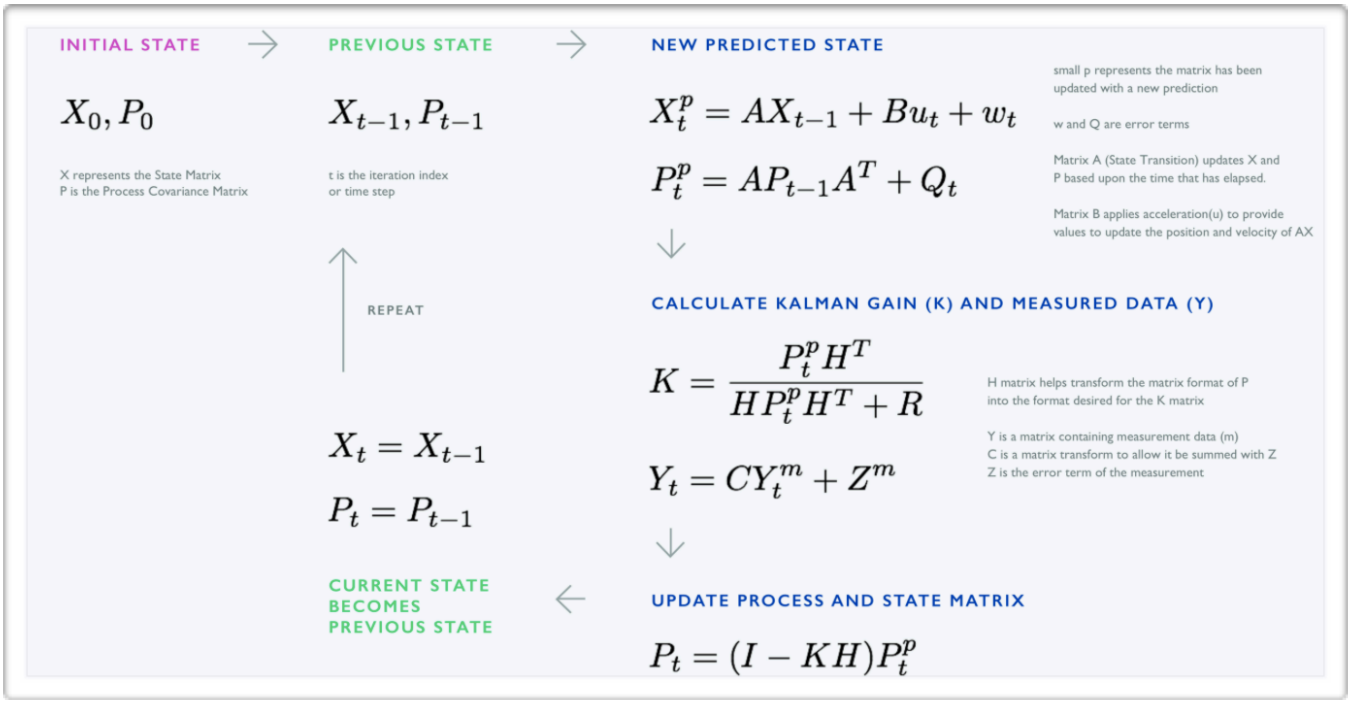


Fig. 1: General Block Diagram of the Kalman Filter

The Kalman filter estimates a process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of (noisy) measurements. As such, the equations for the Kalman filter fall into two groups: time update equations and measurement update equations. The time update equations are responsible for projecting forward (in time) the current state and error covariance estimates to obtain the a priori estimates for the next time step. The measurement update equations are responsible for the feedback —i.e. for incorporating a new measurement into the a priori estimate to obtain an improved a posteriori estimate.

A good way to understand the Kalman gain is to see a simple example put into practice.

1D EXAMPLE OF THE KALMAN FILTER

1D Kalman Filter vs Moving Average

A good way to understand the Kalman filter is to compare it to the simplest way of retrieving a filtered version of noisy measurement. One way to do that is to apply a running average. In a running average one simple takes the average of samples.

Suppose one had the following noisy data to analyze:

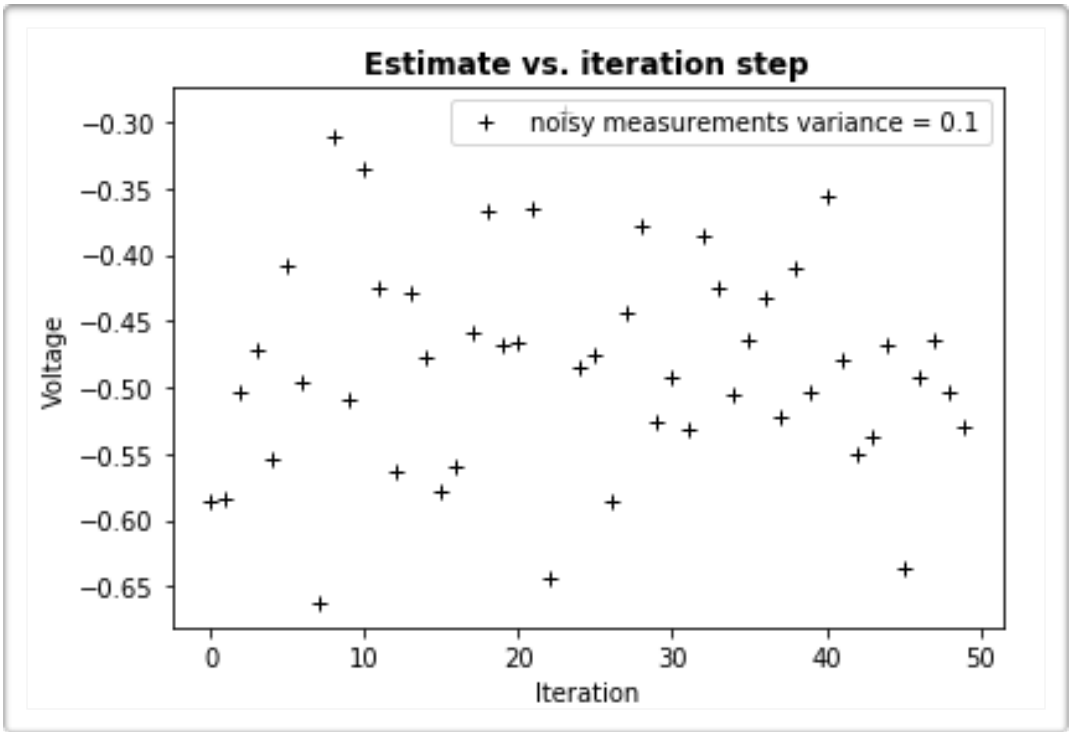


Fig. 2: Noisy measurements of data

Below we can compare the Kalman filtered version with a running average taking 2, 5, and 10 values at a time. It has to be noted that at any point the Kalman filter is only storing the previous estimate and inputting the new measurement, in addition to the error, which is calculated from the measurement and the state matrix. Therefore a running average of length 2 is storing the same amount as a Kalman filter.

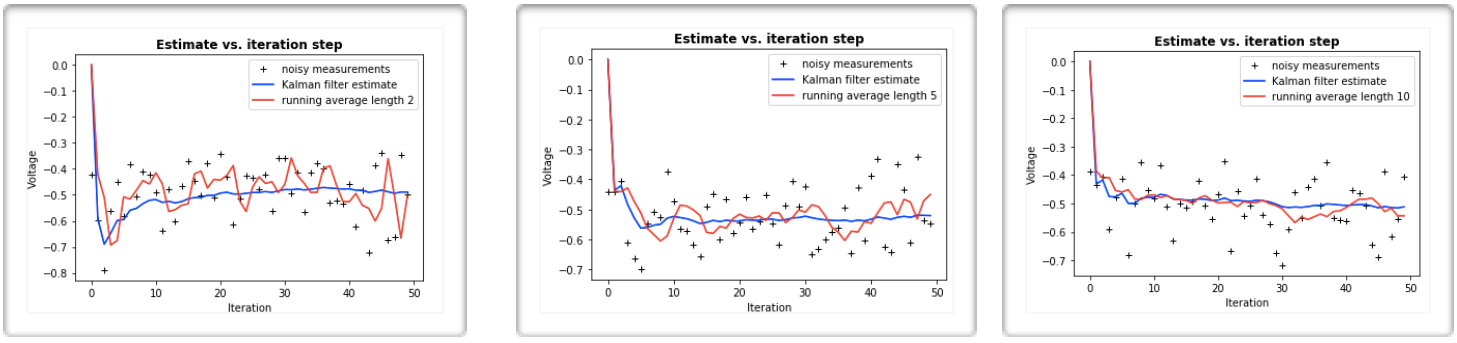


Fig. 3: Kalman Filter vs Running Average

We can see that the running average jitters a lot more in time compared to the Kalman filter. The true value in this case is -0.5 and the Kalman filter inches to that value a lot faster and more steadily than the running average length. In the following figures we'll see why; the Kalman filter is taking into account the variance in the model, in

this case 0.1. The strength of the Kalman filter is also that it performs well when the precise model of the measurement. Let's see what happens when the variance of the Kalman filter differs significantly from that of that measurement.

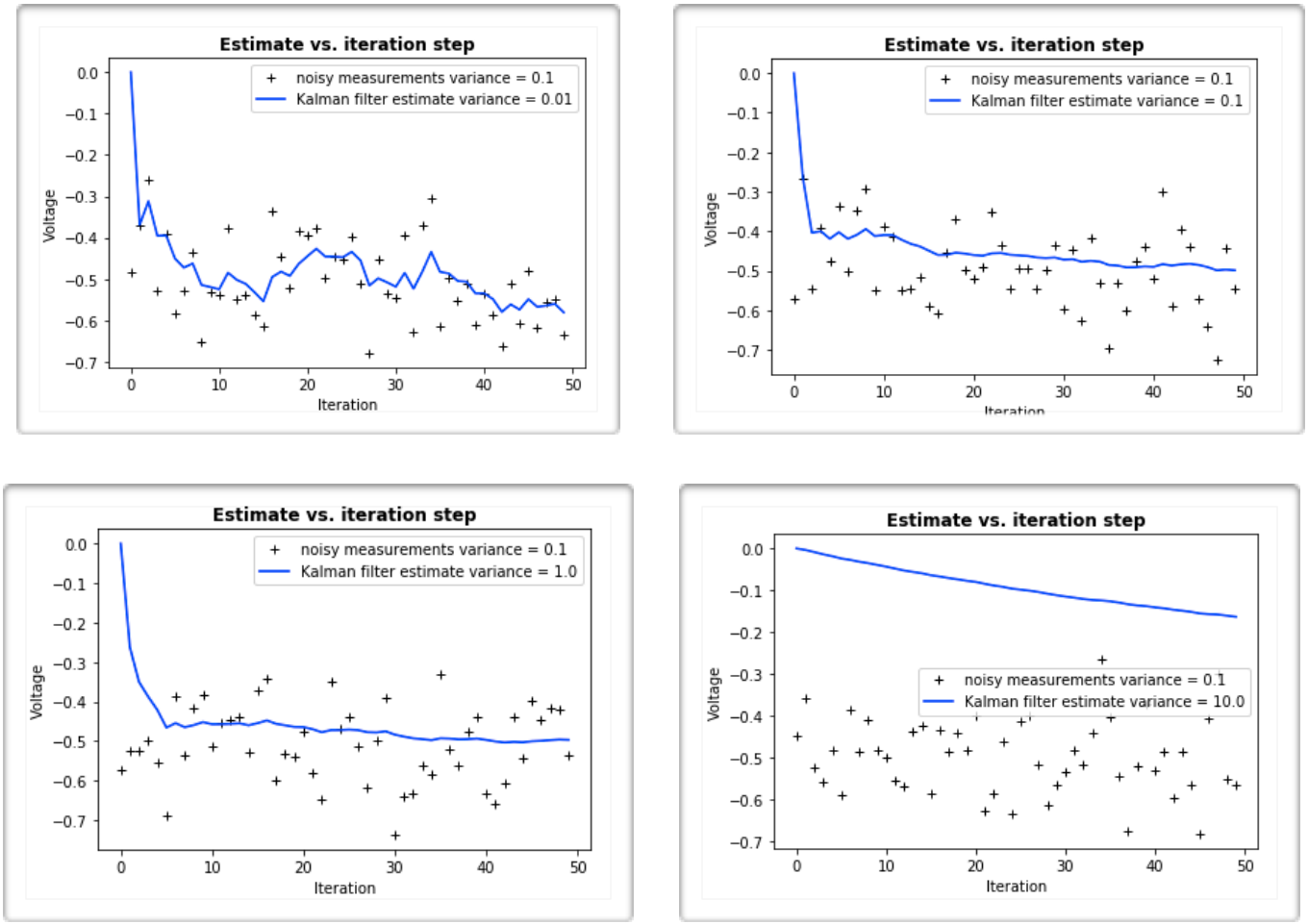


Fig. 4: Kalman Filter with Different Variances

If the variance is lower than that of the model that is being analyzed, the estimate jitters more since it views noisy data as more correct. Whereas with higher variances than that of the model, the estimated value takes longer to settle to the correct value. But as we can see, even though the variance of the assumed underlying model is off, it still tracks to the correct value, albeit either more slowly or with more jitter. Next we'll see the 2D Kalman filter implementation which will follow the block diagram in Fig. 1.

2D KALMAN FILTER MODEL

2D Mouse Model

To model the 2D Kalman filter I decided to use to track the mouse on a frame, add Gaussian noise to its position, use this noisy data as the input to the Kalman filter to see how it reacts to various changes in the different control matrices. I used the cs1lib.py library to track the mouse position. The whole file of the cs1lib.py is readily available online and works well on the spyder IDE. The following code displays the noisy measurement, whereby random noise is added to the true position x and y of the mouse.

```
def graphics():
    global P, last_x
    #print(x)
    clear()
    if draw:
        # true position
        #set_fill_color(r, g, b)
        #draw_circle(x, y, 4)

        # noisy measurement

        cur_xPos = x+100*(random()-0.5)
        cur_yPos = y+100*(random()-0.5)

        set_fill_color(1, 0, 0) # red
        draw_circle(cur_xPos,cur_yPos,1)
```

Another function performs the Kalman filtering:

```
def kf_predict(cur_xPos, cur_yPos, last_x):
    global P
    dt = 0.2
    A = array([
                [1, 0, dt, 0],
                [0, 1, 0, dt],
                [0, 0, 1, 0],
                [0, 0, 0, 1]
        ])

    B = array([
                [1, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 1, 0],
                [0, 0, 0, 1]
        ])

    H = array([
                [1, 0, 1, 0],
                [0, 1, 0, 1],
                [0, 0, 0, 0],
                [0, 0, 0, 0]
        ])
```

```
    Q = array([
                 [0, 0, 0, 0],
                 [0, 0, 0, 0],
                 [0, 0, 0.1, 0],
                 [0, 0, 0, 0.1]
         ])

    R = array([
                 [0.1, 0, 0, 0],
                 [0, 0.1, 0, 0],
                 [0, 0, 0.1, 0],
                 [0, 0, 0, 0.1]
         ])


    velX = cur_xPos - last_x[0]
    velY = cur_yPos - last_x[1]
    measurement = array([cur_xPos, cur_yPos, velX, velY])
    control = array([0,0,0,0])

    #prediction
    x = dot(A,last_x) + dot(B,control)
    P = dot(A, dot(P, A.T)) + Q


    # correction
    S = R + dot(H, dot(P, H.T))
    K = dot(P, dot(H.T, inv(S)))
    y = measurement-dot(H,x)

    cur_x = x+dot(K,y)
    cur_P = dot(eye(4)-dot(K,H),P)

    last_x = cur_x
    last_P = cur_P
    return (last_x,last_P)
```

The various matrices are defined, and the formulas in the block diagram are implemented. It is divided in a prediction step and a correction step. The kf_predict function is called to calculate the new estimated value. The variables P and last_x have to be made into global variables, otherwise they are local in the scope of the function and during the next iteration the previous value will be not be remembered. The new estimated values based on the previous values are displayed, giving the Kalman filtered estimated of the true position.

```
# Kalman filtered point

        new_x,new_P = kf_predict(cur_xPos, cur_yPos, last_x)
        last_x[0] = new_x[0]
        last_x[1] = new_x[1]
        P = new_P

        set_fill_color(0, 1, 0) # green
        draw_circle(new_x[0],new_x[1],4)
```

The last value becomes the new value for the next iteration and the same goes for the P covariance error. The output looks like the following (it's hard to show it without the animation):

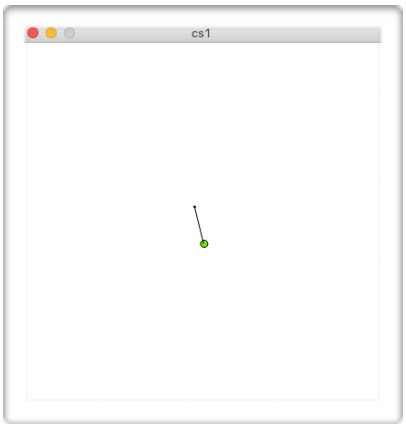


Fig. 5: Mouse Modeling of the Kalman Filter

The green dot is the estimated position of the Kalman filter, and the red is the noisy measurement. The line connects the previous noisy measurement to the estimated value. This gives an idea of what the Kalman filtering is doing under the hood, it is looking at the velocity, ie. the direction of the where the next measurement is happening compared to the previous estimate, and using that information to determine the next estimate. Changing the dt in the A matrix, determines how quickly the estimate reacts to the moving mouse; a smaller dt makes the estimate react faster, but it also makes it jitter more with the noise, whereas a larger dt value makes the estimate more immune to noise, but it also takes more time for it to zone into the true position of the mouse.

Changing other variables in the matrices gives wrong, albeit insightful observations that give an understanding of how the Kalman filter matrices need to be calibrated. For example changing H to the identity matrix makes the Kalman filter estimate something always at a offset from the true position. So you know the filter is working, it's just off by an offset.

Once the Mouse Model of the Kalman filter was figured out it was time to apply it to real life images.

2D Real Life Tracking

The other side of the project always has to do with what to filter. The ultimate goal was to filter a live video stream in order to achieve smoother tracking. There are different methods of achieving tracking. A very easy to use tool that is nonetheless very effective is opencv. Downloading opencv for Mac is not so straightforward and it doesn't work with spyder, because it has to be run in a virtual environment which to my knowledge is not straightforward on the spyder IDE.

I used a KCF (Kernelized Correlation Filters) tracker. The video feed can come from a video or the webcam. Once the live video feed shows up, one presses the 's' button to select the part of the image to track.

Once that part is selected the tracker tracks that image. When there is occlusion the tracker fails, and the Kalman filtered position is used to keep track of the image after tracking has stopped. The kf_predict function from before is used in the pretty much the same way, with minor changes to the dt in the A matrix.

The input to the Kalman filter are the x and y values of the The output values have to be cast to integers because of the way the opencv displays the rectangle is based on pixel values so floating numbers don't work. The blue box is the Kalman filtered estimate of the ball, the green the

Below is an image of the result similar to the one in the front page, but without the tracking working.

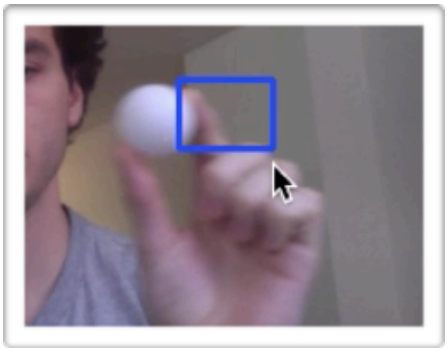


Fig. 6: Filter when the tracking stops working

CONCLUSIONS

End Result

Depending on which tracking method is used the tracking sometime stops working. The filter does an ok job of following with where the object would have been moving based on its velocity. It could be bettered by tweaking the various matrices further. Using that filtered box to tell the tracker where to look for the untracked object would make the whole tracking much more effective, however this requires more in-depth coding in the opencv aspects.

Challenges

The Kalman filter is not very complicated once all the blocks have been set up properly. The set up is the hard part, because the matrices have to be all the correct dimension and have the correct values. Some of the values have to be calibrated. It was also challenging to figure out the proper set up to demonstrate the filtering.

PH235 Related Topics

All of computational physics is done with discrete data so the Kalman filtering is perfectly suited for these applications. The equations governing the matrices the Kalman filter are based on the physical constraints of the object under observations.