Documentation of Kalman.py

By Abdullah Karagøz

I have a pink square that's moving between two points with constant velocity back and forth. It's called "Target" and in the code it's described in the "Target" class. And I have two squares, a yellow and blue one. These are called "Missile" and in the code it's described in the "Projectile" class. The blue Missile is the one we'll implement the Kalman-filter in.

The Missile is moving upwards in constant speed and the goal is to get the blue hit the pink square. The Missiles get data about the position (position in pixels) of the Target so it knows where the Target is. But the data Missile gets is very noisy. Also the position data is normally distributed where the standard-deviation (sigma) is 300 pixels. Thus the Missiles can't hit the Target just by using that data because there's so much noise. The goal is to implement Kalman-filter to the position data so the Missile gets more accurate data about Targets position so the Missile can hit the Target.

The Kalman-filter contains of two parts:
- Prediction: predict where the next position is
- Correction: since the data is noisy the prediction will likely fail. I make som corrections, improve our prediction, and then I predict again in next iteration.

The equations used in the Kalman-filter are following:
- State Extrapolation Equations
- Covariance Extrapolation Equations
- Kalman gain (alpha, beta) computations
- State Update Equations
- Covariance Update Equations

I used a blend of what's taught about Alpha-Beta filter and 1D Kalman-filter from class notes and KalmanFilter.net

The model I used to describe is from "equations of motion":

$$v = u + at$$

$$s = vt + \frac{1}{2}at^2$$

v is new velocity, *u* is old velocity. *S* is position *a* is acceleration t is time.

I am not going to use acceleartion (I assume acceleration = 0 in our model). Also I use a model of motion where the velocity is constant.

But this model doesn't describe our system perfectly. It's because while the velocity is constant, the velocity does change once in the system when Target changes direction.

How I solved this:
I just read the KalmanFilter.net, the lecture notes by Asbjørn Danielsen and their suggestions about using Alpha-Beta or Alpha-Beta-Gamma filter. Then I just implemented the equations in my code as described in the guides.

I first tried to use Alpha, Beta and Gamma but later decided to use only Alpha and Beta. Then I just fine-tuned the initialized values and process noise to get higher Kalman score.

After 600+ tries I got Kalman-score of 0.75. I get something between 0.69 - 0.75.

Why I didn't use Alpha-Beta-Gamma but just Alpha-Beta?
I have first tried using gamma, but that just makes things too complicated. Most of the time the Target has constant velocity, it only accelerates once in each try and that's when changing position. So I thought there's no need to include acceleration or gamma, and just use a model where there's constant velocity. But since the velocity isn't constant 100% of the time in reality, I added some "process noise" for velocity estimation error and position estimation error to make up for that, as described in KalmanFilter.net

In KalmanFilter.net it says one can add some "process noise" when that's the case. So this is what I did and added some process noise to compensate for that in the "Covariance Extrapolation Equations". I used the values by fine tuning, also by trying different values and figure out what works better.

Initialization:
Below is the code of initalization:

```
    def __init__(self):
        self._dt = 1  # time step since it's just one I didn't use
it in equations
        self._pos = 785 # postition estimated
        self._vel = 1 # velocity estimated
        self._pos_est_err = 300 # estimation error / uncertainty
for postiion
        self._vel_est_err = 100 # estimation error / uncertainty
for velocity
        self._pos_mea_err = 300 # measurement error for position
        self._vel_mea_err = 100 # measurement error for velocity
        self._alpha = 1 # alpha
        self._beta = 1 # beta

        self._q = 27 # 27 process noise for position
        self._q_v = 10 # 10 process noise for velocity
```

As I know when initializing values it's not that important to what value to start with. But still it matters, so I tried to initialize with best values. I initialize all variables with some hard-coded values. The position is initialized by 785 because I know the pink square starts at 785 so I think that's best value to use to initialize it. Velocity is set to 1, but it can be 1 or -1.

Measurement error for position is set to 300 because in Target class the noisy data is normally distributed where standard deviation is 300. Other values like measurement error for velocity, estimateion error for postion and velocity are chosed based on fine tuning or just a guess.

I used dynamic Alpha and Beta, it starts from 1. Alpha and Beta are Kalman gain values. Kalman gain being 1 means the Missile has no confidence in its estimations so it only relies on measured data. Kalman gain being 0 means the opposite.

The process noises are set to 27 and 10. These values are found after fine-tuning, also trying multiple values. As mentioned they are used to increase uncertainty since the model I use doesn't describe the system 100 %. The velocity does change once since the Target does change direction.

First I get the measured data "zi" in the "calc_next" function.

Then I use all the five equations. How they're used are described below.

State Extrapolation Equations:

$$x_{n+1} = x_n + \dot{x}_n dt$$

$$\dot{x}_{n+1} = \dot{x}_n$$

X is position and X with one dot is velocity. I just used "equations of motion" here.

These equations are taken from KalmanFilter.net, and it's based on "equations of motion" when acceleration is constant. Acceleration isn't constant in our case, but most of the time the acceleration is 0 because the speed of target is constant, and changes only from point to point. So I didn't use acceleration here.

I initialized the values with following values:
Position = 785, because that's where the pink square starts at.
Velocity = 1. The velocity is constant and it's either 1 or -1.

In the code it looks like this:

```
#State extrapolation
self._pos = self._pos + self._vel
```

I use the equation of motion with constant velocity. Since velocity is constant I don’t use any code to update it. And since dt = 1 I don’t use self._dt in this case.

In the code I don’t have to specify whether the position is new position or previous position, since it’s not mathematical equation. The variables on the right side are previous values, variables on the left side are the new values.

Covariance Extrapolation Equations:
Then I predict the estimation uncertainties with the equations below:

$$p_{x,n+1} = p_{x,n} + p_{v,n}dt^2 + q_x$$

$$p_{v,n+1} = p_{v,n} + q_v$$

These are taken from KalmanFilter.net, it’s shown here how they are derived.

The q values are some process noise which I added in the estimate uncertainty values. That’s becuse it’s as described in KalmanFilter.net that the used model (equatios of motion with constant acceleration) doesn’t describe the system correctly. Thus I add some process noise variables to increase the estimation uncertainty values.

In the code it looks like this:

```
#Covariance extrapolation
self._pos_est_err = self._pos_est_err + self._vel_est_err +
self._q
self._vel_est_err = self._vel_est_err + self._q_v
```

Again I don’t use dt in our code since dt = 1.

Kalman gain computations:
In here I used Kalman-gain equation to compute values of alpha and beta.

$$\alpha = \frac{p_{x,n-1}}{p_{x,n-1} + r_x}$$

$$\beta = \frac{p_{v,n-1}}{p_{v,n-1} + r_v}$$

R is the measurement error for position and velocity.

R for position is based on that the standard deviation for the normally distributed noisy position data is 300 (as it's described in the Target class). I used that standard deviation as measurement error for position. Measurement error for velocity is found based on fine-tuning.

It looks like this in our code:

```
#Alhpa-beta update
self._alpha = self._pos_est_err / (self._pos_est_err +
self._pos_mea_err)
self._beta = self._vel_est_err / (self._vel_est_err +
self._vel_mea_err)
```

State update equations:

$$\hat{x}_{n,n} = \hat{x}_{n,n-1} + \alpha\,(z_n - \hat{x}_{n,n-1})$$

$$\hat{\dot{x}}_{n,n} = \hat{\dot{x}}_{n,n-1} + \beta\left(\frac{z_n - \hat{x}_{n,n-1}}{\Delta t}\right)$$

These are equations that updates the state data I have about the pink square. It simply shows how much weight to give on estimated value vs measured value based on our Kalman gain values (also based on Alpha and Beta).

Kalman gain (alpha and beta) just shows how much the missile is confident about its estimations. High value of alpha and beta shows it's confident about the estimations of position and velocity, thus it puts more weight on our estimated values than measured values. Low alpha and beta means it's not confident about its estimations, so it use more weight on the measured data.

This is how it looks in the code:

```
#State update equations
pos_prev = self._pos
self._pos = pos_prev + self._alpha*(zi - pos_prev)
self._vel = self._vel + self._beta*((zi - pos_prev)/self._dt)
```

Covariance Update Equations:

$$p_{x,n} = (1-\alpha)\,p_{x,n-1}$$

$$p_{v,n} = (1-\beta)\,p_{v,n-1}$$

This is to reduce the estimation error. The more I make estimations, the better I become at estimating, so I reduce our estimation error.

And after each iteration I reduce our estimation error (or estimation uncertainty).

Kalman gain is calculated after each iteration based on relative difference between estimation error and measurement error.

This is how it looks in the code:

```
#Covariance update
self._pos_est_err = (1-self._alpha)*self._pos_est_err
self._vel_est_err = (1-self._beta)*self._vel_est_err
```

After all these 5 equations I just return self._pos.