Kalman Filter

Saurabh Burewar (B18CSE050)

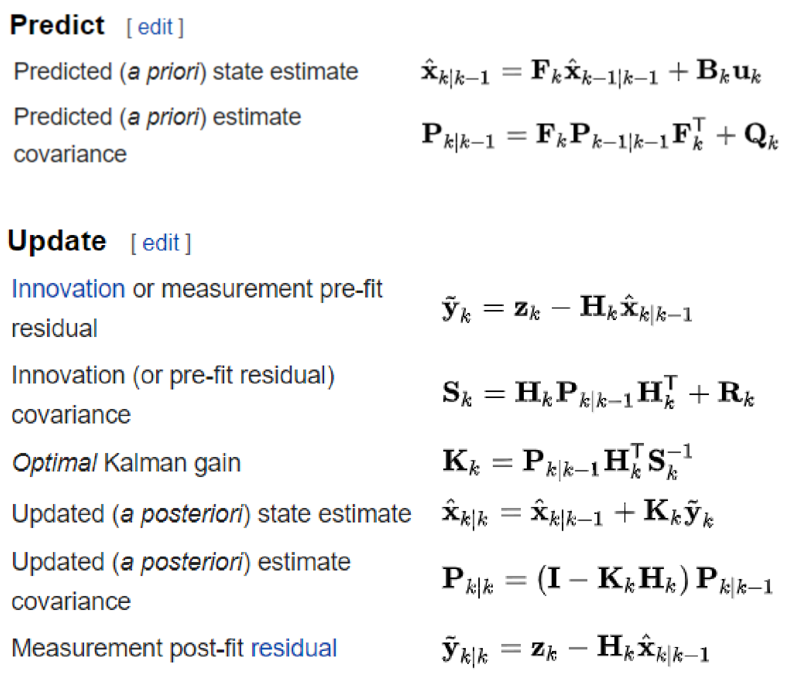
Algorithm

The Kalman filter is implemented in a class called Kalman. When initialized, it prepares all the variables. I have taken all the values of variables from their derivations in the example application of a moving object like a vehicle. One such example is given on Wikipedia as well [1]. Here, A is the state transition matrix, B is the control input matrix and H is the transformation matrix.

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \frac{1}{2}(\Delta t)^2 \\ \Delta t \end{bmatrix} \qquad \mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

For, Q and R, we are using $Q = cI$ and $R = dI$, as given in question first. For the scalers, c and d, I am using the acceleration standard deviation and measurement standard deviation.

The predict and update methods are implemented using numpy as follows -



Predict [edit]

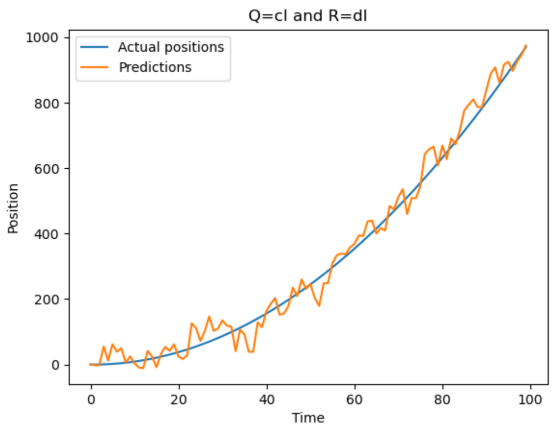
Predicted (*a priori*) state estimate	$\hat{\mathbf{x}}_{k\mid k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1\mid k-1} + \mathbf{B}_k \mathbf{u}_k$
Predicted (*a priori*) estimate covariance	$\mathbf{P}_{k\mid k-1} = \mathbf{F}_k \mathbf{P}_{k-1\mid k-1} \mathbf{F}_k^\mathsf{T} + \mathbf{Q}_k$

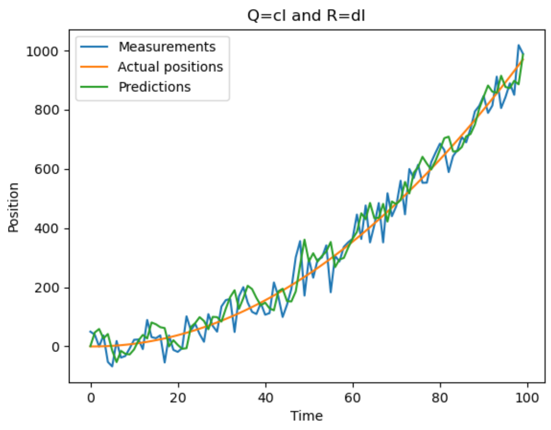
Update [edit]

Innovation or measurement pre-fit residual	$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k\mid k-1}$
Innovation (or pre-fit residual) covariance	$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k\mid k-1} \mathbf{H}_k^\mathsf{T} + \mathbf{R}_k$
Optimal Kalman gain	$\mathbf{K}_k = \mathbf{P}_{k\mid k-1} \mathbf{H}_k^\mathsf{T} \mathbf{S}_k^{-1}$
Updated (*a posteriori*) state estimate	$\hat{\mathbf{x}}_{k\mid k} = \hat{\mathbf{x}}_{k\mid k-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k$
Updated (*a posteriori*) estimate covariance	$\mathbf{P}_{k\mid k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k\mid k-1}$
Measurement post-fit residual	$\tilde{\mathbf{y}}_{k\mid k} = \mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k\mid k-1}$

Results

We create an example to run our Kalman filter on. For this, we have a list of positions that we treat as real positions and run the Kalman filter to get estimations for 100 time steps. The results can be seen in the two plots below. The left one shows actual position vs predictions made by Kalman filter and the right one additionally shows the measurements.

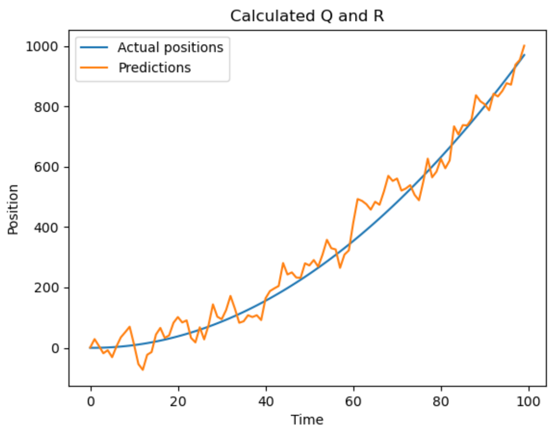


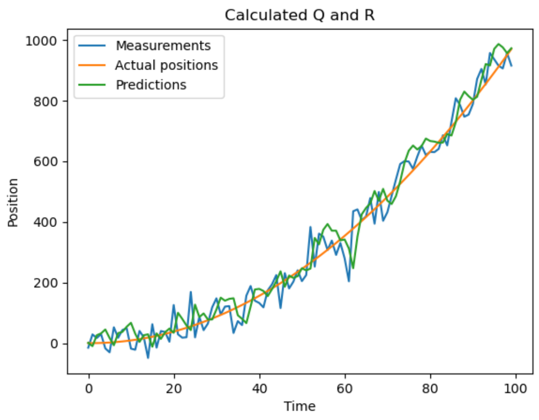


Now, we change the values of Q and R to values that were derived [1].

$$\mathbf{Q} = \mathbf{G}\mathbf{G}^{\mathsf{T}}\sigma_a^2 = \begin{bmatrix} \frac{1}{4}\Delta t^4 & \frac{1}{2}\Delta t^3 \\ \frac{1}{2}\Delta t^3 & \Delta t^2 \end{bmatrix}\sigma_a^2. \qquad \mathbf{R} = \mathrm{E}\left[\mathbf{v}_k\mathbf{v}_k^{\mathsf{T}}\right] = \left[\sigma_z^2\right]$$

Running it again for 100 timensteps, the following plots are obtained -





The predictions are more fluctuating at the start in the second case as compared to the first case, since we have simple cI noise there. But, over time, the second case shows more promising results while the first case is still fluctuating. This shows that update and learning is better in the second case as compared to first.

Kalman for 2D and 3D

To use Kalman filters in 2D and 3D, the update and predict operations will be the same. The only difference will be in the initialization of variables to be used and the input will be of higher dimension.

For 2D [2],

- The state transition matrix will be 4x4 instead of 2x2.

```
[[1, 0, self.t, 0],
[0, 1, 0, self.t],
[0, 0, 1, 0],
[0, 0, 0, 1]]
```

- The control matrix will be 4x2 instead of 2x1.

```
[[(self.t**2)/2, 0],
[0, (self.t**2)/2],
[self.t, 0],
[0, self.t]]
```

- The transformation matrix will be -

```
[[1, 0, 0, 0],
[0, 1, 0, 0]]
```

- We will take $Q = cI_4$ and $R = dI_2$

For 3D [3],

- The state transition matrix will be 6x6.

```
[[1, self.t, 0, 0, 0, 0],
[0, 1, 0, 0, 0, 0],
[0, 0, 1, self.t, 0, 0],
[0, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 1, self.t],
[0, 0, 0, 0, 0, 1]]
```

- The control matrix will be 6x3.

```
[[(self.t**2)/2, 0, 0],
[0, (self.t**2)/2, 0],
[0, 0, (self.t**2)/2],
[self.t, 0, 0],
[0, self.t, 0],
[0, 0, self.t]]
```

- The transformation matrix will be -

```
[[1, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0],
[0, 0, 0, 0, 1, 0]]
```

- We will take $Q = cI_6$ and $R = dI_3$

I just tried to test the 2D and 3D filters by creating a simple set of positions (Just repeating the same coordinates in all dimensions). Then, running the algorithm for 100 timesteps and we get the following list of predictions.

Predictions for 2D -
[(1.0, 1.0), (29.0, 29.0), (32.0, 32.0), (-20.0, -20.0), (-19.0, -19.0), (15.0, 15.0), (12.0, 12.0), (-26.0, -26.0), (-24.0, -24.0), (9.0, 9.0), (-28.0, -28.0), (-32.0, -32.0), (-2.0, -2.0), (5.0, 5.0), (-32.0, -32.0), ….]

Predictions for 3D -
[(1.0, 1.0, 1.0), (1.0, 1.0, 2.0), (32.0, 11.0, 33.0), (43.0, 12.0, 46.0), (49.0, 11.0, 52.0), (85.0, 18.0, 88.0), (128.0, 25.0, 131.0), (26.0, -6.0, 30.0), (-20.0, -15.0, -16.0), (-23.0, -11.0, -19.0), (6.0, 0.0, 9.0), (30.0, 7.0, 34.0), ….]

References

1. https://en.wikipedia.org/wiki/Kalman_filter
2. https://machinelearningspace.com/2d-object-tracking-using-kalman-filter/
3. https://dsp.stackexchange.com/questions/26115/kalman-filter-to-estimate-3d-position-of-a-node