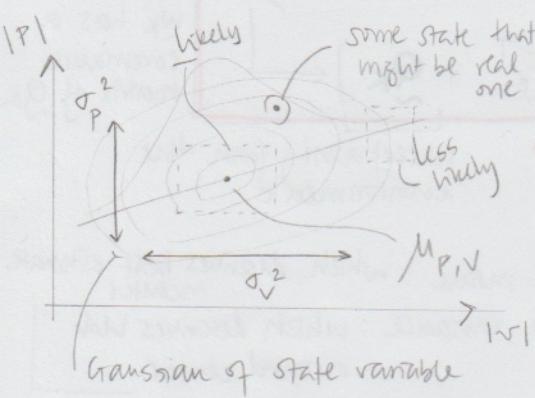


KALMAN FILTER

Bzargh tutorial & IEEE Lecture Notes (Ramsey Faragher)

- Kalman Filter: filter out uncertain information (noise) from a dynamic system and make an educated guess of what's system going to do next.

① State & covariance matrix



• $\tilde{x}_k = [\vec{P}, \vec{V}]^T$: state vector of a system at moment k

\vec{P} : position vector of random Gaussian variable

\vec{V} : velocity vector variable

covariance $\sigma_{PP}^2 = \sigma_P^2$

THE NOTION OF [A]

• $\mu_{P,V}$: mean state value of Gaussian state

variance of state

P: position

V: velocity

$$\tilde{x}_k = \begin{bmatrix} \sigma_{PP}^2 & \sigma_{PV}^2 \\ \sigma_{VP}^2 & \sigma_{VV}^2 \end{bmatrix} = \text{Cov}(\tilde{x}_k)$$

||

COVARIANCE MATRIX

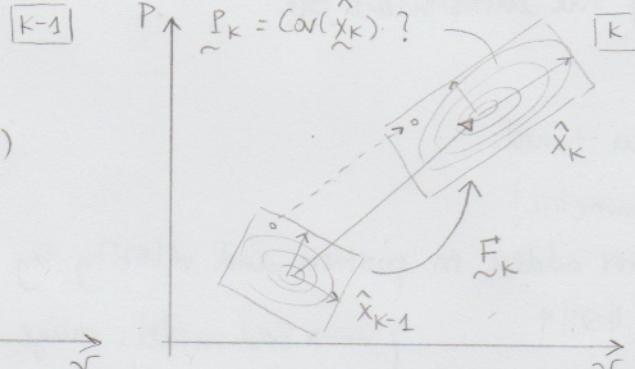
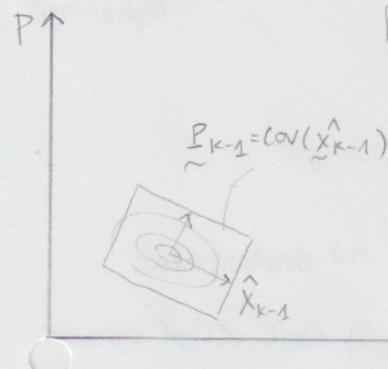
symmetric

- σ_{ij} : related to the degree of correlation between elements i & j

- State \tilde{x} can be built of different signals that can be noisy / uncertain \rightarrow VARIANCE

- Different components of state can be correlated: $|V| |P| \rightarrow |P| |V|$

② Prediction Step: Based on the system's dynamic equations



- Since: $\text{Cov}(x) = \sum \Leftrightarrow \text{Cov}(Ax) = A\sum A^T$

- PREDICTION STEP:

$$\begin{aligned} \hat{x}_k &= F_k \hat{x}_{k-1} \\ P_k &= F_k P_{k-1} F_k^T \end{aligned}$$

Note: we need only F_k to go from k-1 to k

state variable in next step is predicted

covariance matrix of state variable is predicted

$$P_k = P_{k-1} + \Delta t v_{k-1}$$

$$v_k = v_{k-1}$$

$$\begin{bmatrix} P_k \\ v_k \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} P_{k-1} \\ v_{k-1} \end{bmatrix}$$

F_k : PREDICTION MATRIX

↓

we capture 1st order influences related to the state itself \Rightarrow acceleration not considered yet

aka G_j

③ External influence: control matrix

$$P_k = P_{k-1} + \Delta t v_{k-1} + \frac{1}{2} a \Delta t^2$$

$$v_k = v_{k-1} + a \Delta t$$

$$\hat{x}_k = F_k \hat{x}_{k-1} + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} a$$

$$\hat{x}_k = F_k \hat{x}_{k-1} + B_k u_k$$

CONTROL MATRIX

CONTROL VECTOR
 $a = f/m$

Note:

B/G : matrix/vector

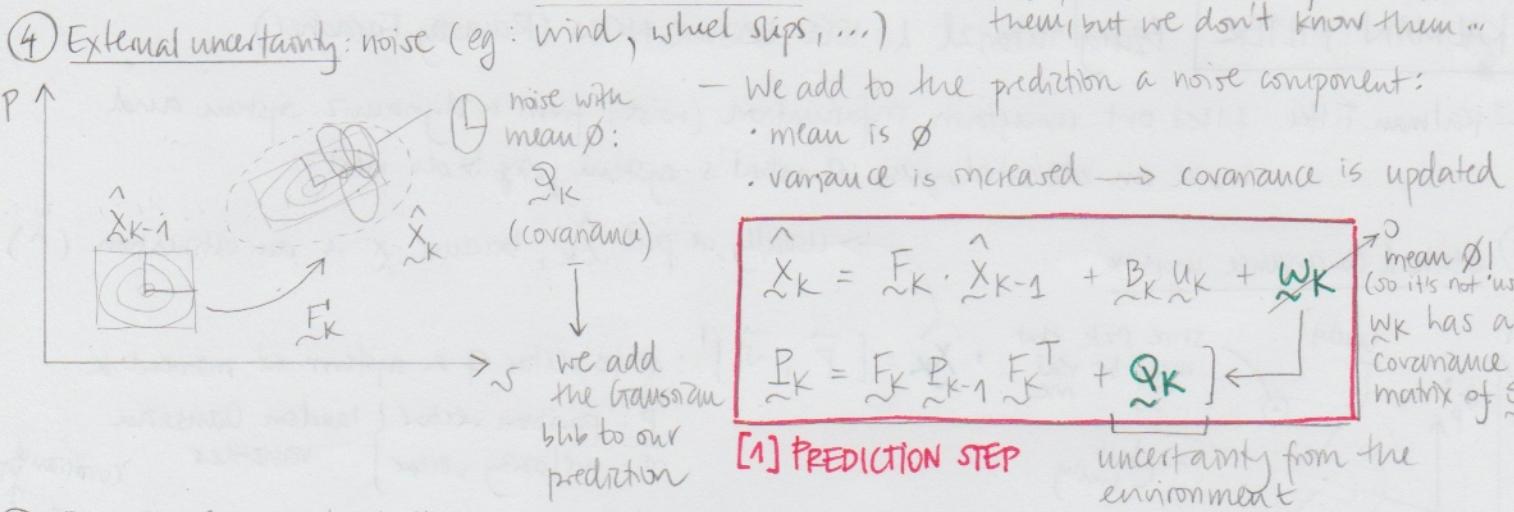
u : vector/scalar

depending on problem

WATCH OUT:

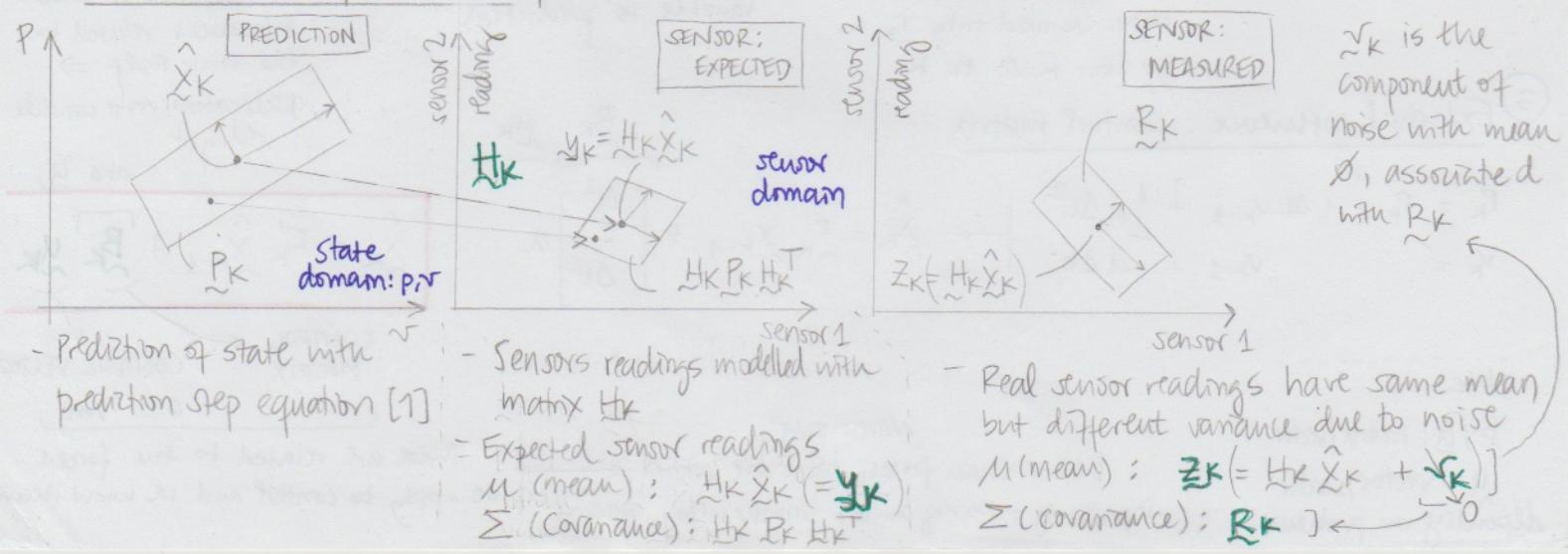
Better than forces, these are control inputs, eg: steering angle, throttle setting, braking force ...

These are related to the forces we apply to control and we know them



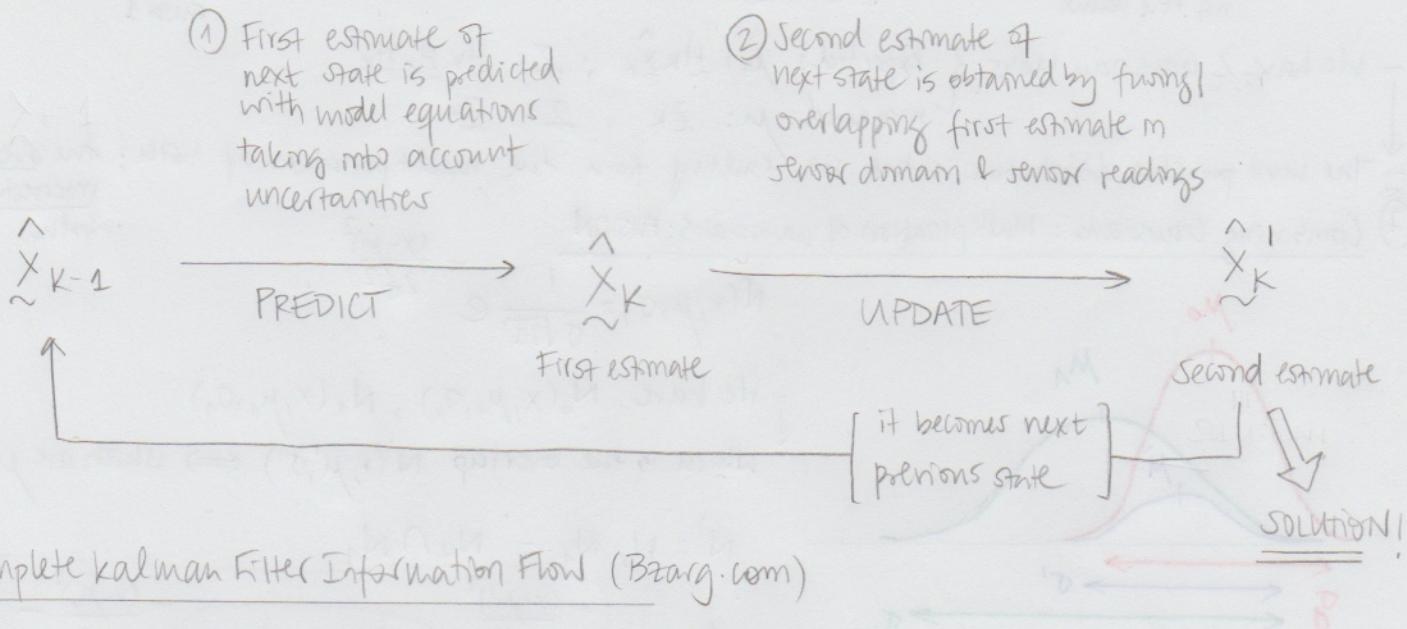
- ⑤ Insights from points 1-4
- Gaussian variables have 2 equations each
 - one for the mean: which describes best estimate in center
 - one for the variance: which describes blob around center
 - NEW STATE ESTIMATE (\hat{x}_k) is a PREDICTION made from
 - previous state \hat{x}_{k-1} estimate and system's (linear) model (\tilde{F}_k), and
 - correction for external influences/control ($\tilde{B}_k \tilde{u}_k = \tilde{g}_k$)
 - NEW UNCERTAINTY (VARIANCE), (\tilde{P}_k) is a PREDICTION made from:
 - previous uncertainty \tilde{P}_{k-1} and system's (linear) model (\tilde{F}_k), and
 - additional uncertainty of the environment \tilde{Q}_k
 - How to choose \tilde{Q}_k ?
 - Usually hand-chosen, then tuned
 - $\tilde{Q}_k = \begin{bmatrix} PP & PV \\ VP & VV \end{bmatrix}$
 - Symmetric!
 - error added in position and velocity by the environment
 - if $\|\tilde{Q}\| \uparrow$ } → Bad model: range of state vector extends
 - if $\|\tilde{Q}\| \gg \|P\|$
 - if \tilde{Q} diagonal: parameters independent from each other
BUT usually not the case; eg: under wind, pos. and vel. will vary in a correlated way...

⑥ Measurements of sensors: refine estimate

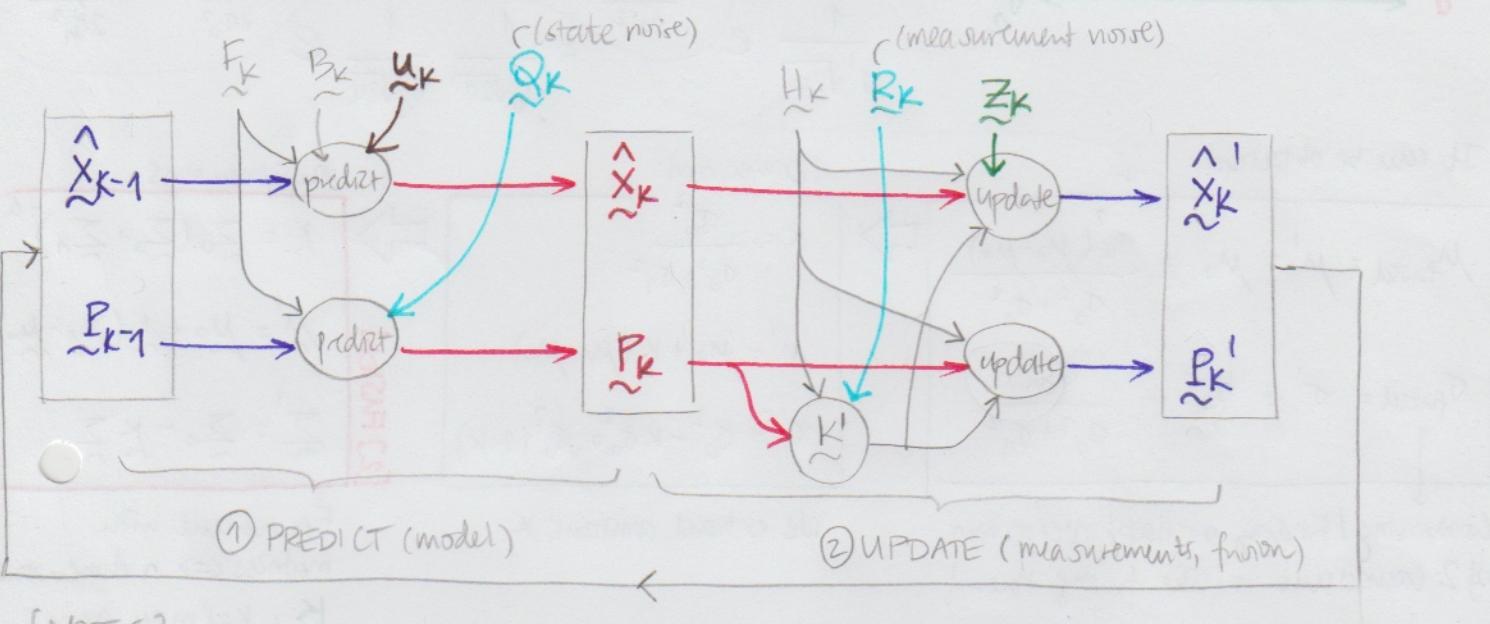


① Kalman Filter: summary of all steps

- The best estimate we can make is provided by combining our knowledge from PREDICTION and MEASUREMENT.
- \hat{x}_k is computed with 2 steps at each cycle:

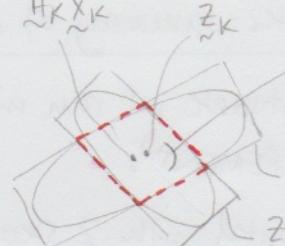


⑩ Complete Kalman Filter Information Flow (Bzarg.com)



[NOTES]

- How can we obtain \hat{P} , \hat{Q} , \hat{R} , \hat{H} ? Examples? → see Kalman Filter Tutorial: Rudy Salguero, ...
- Extended Kalman Filter: for non-linear systems; it works by linearizing PREDICTIONS and MEASUREMENTS around its mean.
- Note that we could fuse different sensor signals:
 - The product of 2 Gaussians is a Gaussian
 - An endless number of Gaussian PDFs can be multiplied over time
- Note that we fuse prediction and measurement Gaussians in sensor domain and then we transform them back to state domain (using H)

	PREDICTION	SENSOR: EXPECTED	SENSOR: MEASURED	
M	\hat{x}_k	$(y_k = H_k \hat{x}_k)$	$\tilde{z}_k = H_k \hat{x}_k + \tilde{v}_k^0$	
Σ	\tilde{P}_k	$H_k P_k H_k^T$	R_k	$\Sigma' = \tilde{P}_k - K^T H_k \tilde{P}_k H_k^T$

expected and measured are usually not the same

expected model measured value

Covariance due to sensor noise

overlap of both distributions is most probable!

$$H_k \hat{x}_k, H_k \tilde{P}_k H_k^T$$

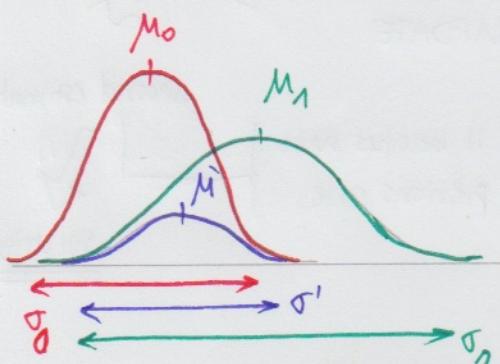
$$\tilde{z}_k, R_k$$

We have 2 Gaussian blobs:

- expected: $\mu: H_k \hat{x}_k$, $\Sigma: H_k P_k H_k^T$
- measured: $\mu: \tilde{z}_k$, $\Sigma: R_k$

The most probable distribution is the one resulting from the multiplication of both: the overlap or intersection of both

⑦ Combining Gaussians: Multiplication of Gaussians: FUSION



$$N(x, \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

We have $N_0(x, \mu_0, \sigma_0)$, $N_1(x, \mu_1, \sigma_1)$

Which is the overlap $N'(x, \mu', \sigma') \Leftrightarrow \text{which are } \mu', \sigma'?$

$$N' = N_0 \cdot N_1 = N_0 \cap N_1$$

$$\frac{1}{\sigma'_1 \sqrt{2\pi}} e^{-\frac{(x-\mu')^2}{2\sigma'^2}} = \frac{1}{\sigma_0 \sqrt{2\pi}} \frac{1}{\sigma_1 \sqrt{2\pi}} e^{-\frac{(x-\mu_0)^2}{2\sigma_0^2} - \frac{(x-\mu_1)^2}{2\sigma_1^2}}$$

It can be obtained:

$$\begin{aligned} \mu_{\text{fused}} &= \mu' = \mu_0 + \frac{\sigma_0^2(\mu_1 - \mu_0)}{\sigma_0^2 + \sigma_1^2} \\ \sigma_{\text{fused}}^2 &= \sigma'^2 = \sigma_0^2 - \frac{\sigma_0^4}{\sigma_0^2 + \sigma_1^2} \end{aligned}$$

Combining / Finding overlap / intersection of 2 Gaussians is like fusing them!

1 Dimension

$$K = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2}$$

$$\mu' = \mu_0 + K(\mu_1 - \mu_0)$$

$$\sigma'^2 = \sigma_0^2 - K\sigma_0^2 = \sigma_0^2(1-K)$$

We extract constant K

n Dimensions

$$\tilde{K} = \tilde{\Sigma}_0 (\tilde{\Sigma}_0 + \tilde{\Sigma}_1)^{-1}$$

$$\tilde{\mu}' = \tilde{\mu}_0 + \tilde{K}(\tilde{\mu}_1 - \tilde{\mu}_0)$$

$$\tilde{\Sigma}' = \tilde{\Sigma}_0 - \tilde{K}\tilde{\Sigma}_1$$

[2] FUSION

Equivalent with matrices \Leftrightarrow n dimensions

K : Kalman gain

⑧ Update step: Fusion of measured and expected sensor values

We have 2 distributions and want to find their overlap $\{(\mu_0, \Sigma_0) = (H_k \hat{x}_k, H_k \tilde{P}_k H_k^T)$: expected

$(\mu_1, \Sigma_1) = (\tilde{z}_k, R_k)$: measured

Applying Fusion eqs [2] (in sensor domain):

$$\begin{aligned} \frac{\mu'}{H_k \hat{x}_k} &= \frac{\mu_0}{H_k \hat{x}_k} + \frac{\mu_1}{H_k \hat{x}_k} (\tilde{z}_k - H_k \hat{x}_k) \\ \frac{\Sigma'}{\Sigma} &= \frac{\Sigma_0}{\Sigma} - \frac{\Sigma_1}{\Sigma} H_k \tilde{P}_k H_k^T \\ K &= \frac{H_k \tilde{P}_k H_k^T}{\Sigma_0} \cdot \frac{(H_k \tilde{P}_k H_k^T + R_k)^{-1}}{\Sigma_1} \end{aligned}$$

[3] UPDATE \rightarrow updated fused state!

$$\hat{x}'_k = \hat{x}_k + K'(\tilde{z}_k - H_k \hat{x}_k)$$

$$\tilde{P}'_k = \tilde{P}_k - K' H_k \tilde{P}_k$$

$$K' = \tilde{P}_k H_k^T (H_k \tilde{P}_k H_k^T + R_k)^{-1}$$

H_k is removed: we go from sensor to state domain

KALMAN FILTER

Tutorial for Undergraduates - Rhudy, Salgner, Holappa

Intl. J. of Computer Science & Engineering Survey
2017

- This is a practical tutorial on how to use the equations of the kalman filter - read derivation first (Bzarg.org)
- Kalman filter removes noise.
- Notes on the equations and variables

	(1) PREDICT	(2) UPDATE	
M	$\hat{x}_k = \tilde{F}_k \hat{x}_{k-1} + \tilde{B}_k u_k + \tilde{w}_k$	$\hat{y}_k = \tilde{H}_k \hat{x}_k + \tilde{v}_k$	\hat{z}_k
Z	$\tilde{P}_k = \tilde{F}_k \tilde{P}_{k-1} \tilde{F}_k^T + \tilde{Q}_k$	$\tilde{H}_k \tilde{P}_k \tilde{H}_k^T$	R_k
		EXPECTED	MEASURED

Example: object in free fall without air

$h \uparrow h_0$

$v(t) = -g$

$$\begin{aligned}\hat{x}'_k &= \hat{x}_k + K^T (z_k - \tilde{H}_k \hat{x}_k) \\ \tilde{P}'_k &= (I - K^T \tilde{H}_k) \tilde{P}_k \\ K^T &= \tilde{P}_k \tilde{H}_k^T (\tilde{H}_k \tilde{P}_k \tilde{H}_k^T + R_k)^{-1}\end{aligned}$$

Variable	Description	Dimension	Example	Comments/Notes
\hat{x}_k	state vector	$n_x \times 1$	$\hat{x}_k = [h \dot{h}]^T$	<ul style="list-style-type: none"> - Desired result! - It might include more variables than the really wanted if necessary. - STATE DOMAIN!
\tilde{F}_k	system matrix of state	$n_x \times n_x$	$\tilde{F}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$ (in this case it's constant)	<ul style="list-style-type: none"> - Always square: $\dim(\hat{x}) \times \dim(\hat{x})$ - Linear transformation applied to previous state to obtain current new state (without considering input/control variables) $\begin{aligned}\hat{x}_k &= \begin{bmatrix} h_k \\ \dot{h}_k \end{bmatrix} = \begin{bmatrix} h_{k-1} + h_{k-1} \Delta t - \frac{1}{2} g \Delta t^2 \\ \dot{h}_{k-1} - g \Delta t \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}}_{\tilde{F}_k} \underbrace{\begin{bmatrix} h_{k-1} \\ \dot{h}_{k-1} \end{bmatrix}}_{\hat{x}_{k-1}} + \underbrace{\begin{bmatrix} -\frac{1}{2} \Delta t^2 \\ -g \Delta t \end{bmatrix}}_{B_k u_k}\end{aligned}$
u_k	input or control vector	$n_u \times 1$	$u_k = g$ (in this case it's constant)	<ul style="list-style-type: none"> - It can be a vector or a scalar <ul style="list-style-type: none"> if vector, B_k will be a matrix if scalar, B_k will be a vector

(Cont.)

Variable	Description	Dimension	Example	Comments / Notes
\tilde{u}_k				<ul style="list-style-type: none"> - information related to the system <u>dynamics</u> - any other terms which appear in the filter that don't need to be estimated as states can be inputs - typical case: acceleration
\tilde{B}_k	system matrix of input/control	$n_x \times n_u$	$\tilde{B}_k = \begin{bmatrix} -\frac{\Delta t^2}{2} \\ -\Delta t \end{bmatrix}$ <p>(in this case it's constant)</p>	<ul style="list-style-type: none"> - It can be a vector or a matrix, depending on n_u (see \tilde{u}_k) - It contains the coefficients to apply to the input/control vector
\tilde{y}_k	output vector	$n_y \times 1$	$\tilde{y}_k = \tilde{H}_k \tilde{x}_k + \tilde{v}_k$ $= h_k$	<ul style="list-style-type: none"> - \tilde{y}_k is the transformation of our state prediction to the sensor domain: it's the <u>sensor measurement we'd expect!!</u>
\tilde{H}_k	observation matrix	$n_y \times n_x$	$\tilde{H}_k = [1 \ 0]$ <p>(in this case \tilde{H}_k is constant)</p>	<ul style="list-style-type: none"> - \tilde{H}_k transforms the predicted state to the sensor domain - \tilde{H}_k is a linear transformation that can <ul style="list-style-type: none"> SELECT $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ SCALE $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_3 \end{bmatrix}$ COMBINE $2 \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 + x_2$
\tilde{w}_k	process noise vector	$n_x \times 1$	mean \emptyset	<ul style="list-style-type: none"> - these noise vectors have mean \emptyset, thus they don't typically appear in the equations of interest
\tilde{v}_k	measurement noise vector	$n_y \times 1$	mean \emptyset	<ul style="list-style-type: none"> - However, they have a random component \Rightarrow the blob of the corresponding distribution is considered to be modified
\tilde{Q}_k	covariance of the process noise	$n_x \times n_x$	$\tilde{Q}_k = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$	<ul style="list-style-type: none"> - Process noise (w_k, v_k) is related to the error that might be in the model equations, e.g., we could consider the effects of wind or air resistance
\tilde{R}_k	covariance of the measurement	$n_y \times n_y$	$\tilde{R}_k = 4$ <p>(they can be constant)</p>	<ul style="list-style-type: none"> - BUT: if equations/model are considered accurate enough, we can set $\tilde{Q} = Q$.

(cont.)

Variable	Description	Dimension	Example	Comments / Notes
\tilde{Q}_K				- Measurement covariance can be derived from sensor data sheet: if standard deviation of error is assumed 2m $\rightarrow \tilde{R}_K = 4m^2$ Measurement
\tilde{R}_K				- IMPORTANT: \tilde{Q} & R affect considerably the performance of the filter; they are like weighting factors between the prediction (state) equations and the measurement (output) equations:
K_K	Kalman gain	$n_x \times n_y$	Computed each step, see formula	$ K \sim \frac{ Q }{ Q + R }$
				$ Q \uparrow$ larger uncertainty in state equations, correct more with measurement $ R \uparrow$ larger uncertainty in measurement, correct less with measurement
P_0	assumed initial state error cov. matrix	$n_x \times n_x$	$\tilde{P}_0 = \begin{bmatrix} 10 & 0 \\ 0 & 0.01 \end{bmatrix}$	<ul style="list-style-type: none"> - The Kalman gain controls the impact of the measurement correction: $\hat{\tilde{x}}_K = \hat{\tilde{x}}_K + K^T (\tilde{z}_K - H_K \hat{\tilde{x}}_K)$ <p style="text-align: center;">updated state predicted first state diff between measured and expected states in sensor domain</p> <ul style="list-style-type: none"> - \tilde{P}_0 is related to the assumed initial state \tilde{x}_0 and it affects the initial convergence of the filter. - It is updated to P_k in the following steps - Its value is not that important, sometimes it's initialized to identity matrix - In our case the ball starts at $\tilde{x}_0 = [100 \ 0]^T$, but we assume $\hat{\tilde{x}}_0 = [105 \ 0]^T$ with a non-zero velocity of 0.1 m/s $\rightarrow 0.01 m^2/s^2$
\tilde{x}_0	true initial State vector	$n_x \times 1$	$[100 \ 0]^T$	
$\hat{\tilde{x}}_0$	assumed initial State vector	$n_x \times 1$	$[105 \ 0]^T$	Note: in the absence of info, we can guess initial values of P by observing expected & measured values: if big diff, big values in P .
Δt	Step time or time increment	1×1	0.001s	

Notes on the covariance matrix

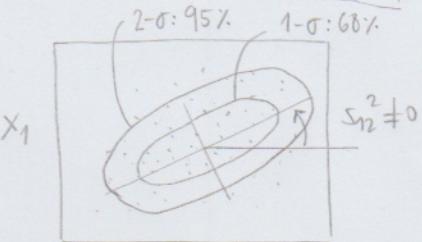
$$\tilde{x} = [x_1 \dots x_m]^T \quad \xrightarrow{\text{m: size/dim of } \tilde{x}}$$

↔
of vars

n : # samples (i)

$$\tilde{x}_1 \dots \tilde{x}_n$$

$$\begin{bmatrix} x_m \\ x_{12} \\ | \\ x_{1m} \end{bmatrix} \dots \begin{bmatrix} x_{n1} \\ x_{n2} \\ | \\ x_{nm} \end{bmatrix}$$



$$\text{Cov}(\tilde{x}) = \begin{bmatrix} s_1^2 & & s_{jk}^2 \\ & s_2^2 & \\ \vdots & & \ddots & s_m^2 \end{bmatrix} \quad \left\{ \begin{array}{l} s_j^2 = \frac{1}{n} \sum_{i=1}^n (\bar{x}_j - x_{ij})^2 \text{ of var. } j \text{ along all samples} \\ s_{jk}^2 = \frac{1}{n} \sum_{i=1}^n (\bar{x}_j - x_{ij})(\bar{x}_k - x_{ik}) \text{ between vars } j \& k \text{ along all samples} \\ \bar{x}_j = \frac{1}{n} \sum_{i=1}^n x_{ij} \text{ mean of var. } j \end{array} \right.$$

if non-zero, there's a correlation between vars j & k

$$\tilde{x} = [p \ v]^T$$

$$\text{Cov}(\tilde{x}) = \begin{bmatrix} s_p^2 & s_{pv} \\ s_{vp} & s_v^2 \end{bmatrix} \quad \left\{ \begin{array}{l} s_p^2 = \frac{1}{n} \sum_{i=1}^n (\bar{p} - p_i)^2 \\ s_v^2 = \frac{1}{n} \sum_{i=1}^n (\bar{v} - v_i)^2 \\ s_{pv} = s_{vp} = \frac{1}{n} \sum_{i=1}^n (\bar{p} - p_i)(\bar{v} - v_i) \end{array} \right.$$

Correlation matrix

$$\tilde{R} = \begin{bmatrix} 1 & r_{12} & r_{13} & \dots & r_{1m} \\ & 1 & r_{23} & \dots & r_{2m} \\ \vdots & & 1 & & \vdots \\ & & & 1 & \end{bmatrix} \quad \text{sim}$$

When covariance matrix isn't diagonal, there's correlation between the j & k variables that have non-zero s_{jk} or r_{jk}

$$r_{jk} = \frac{s_{jk}^2}{s_j s_k} = \frac{\sum_{i=1}^n (\bar{x}_j - x_{ij})(\bar{x}_k - x_{ik})}{\sqrt{\sum_{i=1}^n (\bar{x}_j - x_{ij})^2 \sum_{i=1}^n (\bar{x}_k - x_{ik})^2}}$$

correlation coeff. between variables j and k along all i samples

(ψ) You can also compute your covariance matrices by performing measurements...

```

% Simulate the true signal and measurement
% Define the system
N = 1000; % number of time steps
dt = 0.001; % Sampling time (s)
t = dt*(1:N); % time vector (s)
F = [1, dt; 0, 1]; % system matrix - state
G = [-1/2*dt^2; -dt]; % system matrix - input
H = [1 0]; % observation matrix
Q = [0, 0; 0, 0]; % process noise covariance
u = 9.80665; % input = acceleration due to gravity (m/s^2)
I = eye(2); % identity matrix
% Define the initial position and velocity
y0 = 100; % m
v0 = 0; % m/s
% Initialize the state vector (true state)
xt = zeros(2, N); % True state vector
xt(:, 1) = [y0; v0]; % True initial state
% Loop through and calculate the state
for k = 2:N
    % Propagate the states through the prediction equations
    xt(:, k) = F*xt(:, k-1) + G*u;
end
% Generate the noisy measurement from the true state
R = 4; % m^2/s^2
v = sqrt(R)*randn(1, N); % measurement noise
z = H*xt + v; % noisy measurement

```

$$\tilde{G} = \tilde{B}$$

PREDICTION
(along all time
steps with dt)

```

% Perform the Kalman filter estimation
% Initialize the state vector (estimated state)
x = zeros(2, N); % Estimated state vector
x(:, 1) = [105; 0]; % Guess for initial state
% Initialize the covariance matrix
P = [10, 0; 0, 0.01]; % Covariance for initial state error
% Loop through and perform the Kalman filter equations recursively
for k = 2:N
    % Predict the state vector
    x(:, k) = F*x(:, k-1) + G*u; % 1.PREDICT
    % Predict the covariance
    P = F*P*F' + Q;
    % Calculate the Kalman gain matrix
    K = P*H'/(H*P*H' + R); % 2.UPDATE
    % Update the state vector
    x(:, k) = x(:, k) + K*(z(k) - H*x(:, k));
    % Update the covariance
    P = (I - K*H)*P;
end

```

```

% Plot the results
% Plot the states
figure(1);
subplot(211);
plot(t, z, 'g-', t, x(1,:), 'b--', 'LineWidth', 2);
hold on; plot(t, xt(1,:), 'r:', 'LineWidth', 1.5)
xlabel('t (s)'), ylabel('x_1 = h (m)'), grid on;
legend('Measured', 'Estimated', 'True');
subplot(212);
plot(t, x(2,:), 'b--', 'LineWidth', 2);
hold on; plot(t, xt(2,:), 'r:', 'LineWidth', 1.5)
xlabel('t (s)'), ylabel('x_2 = v (m/s)'), grid on;
legend('Estimated', 'True');

% Plot the estimation errors
figure(2);
subplot(211);
plot(t, x(1,:)-xt(1,:), 'm', 'LineWidth', 2)
xlabel('t (s)'), ylabel('\Delta x_1 (m)'), grid on;
subplot(212);
plot(t, x(2,:)-xt(2,:), 'm', 'LineWidth', 2)
xlabel('t (s)'), ylabel('\Delta x_2 (m/s)'), grid on;

```

now we must add it,
because we're simulating !!

1.PREDICT

2.UPDATE

plots on
next page

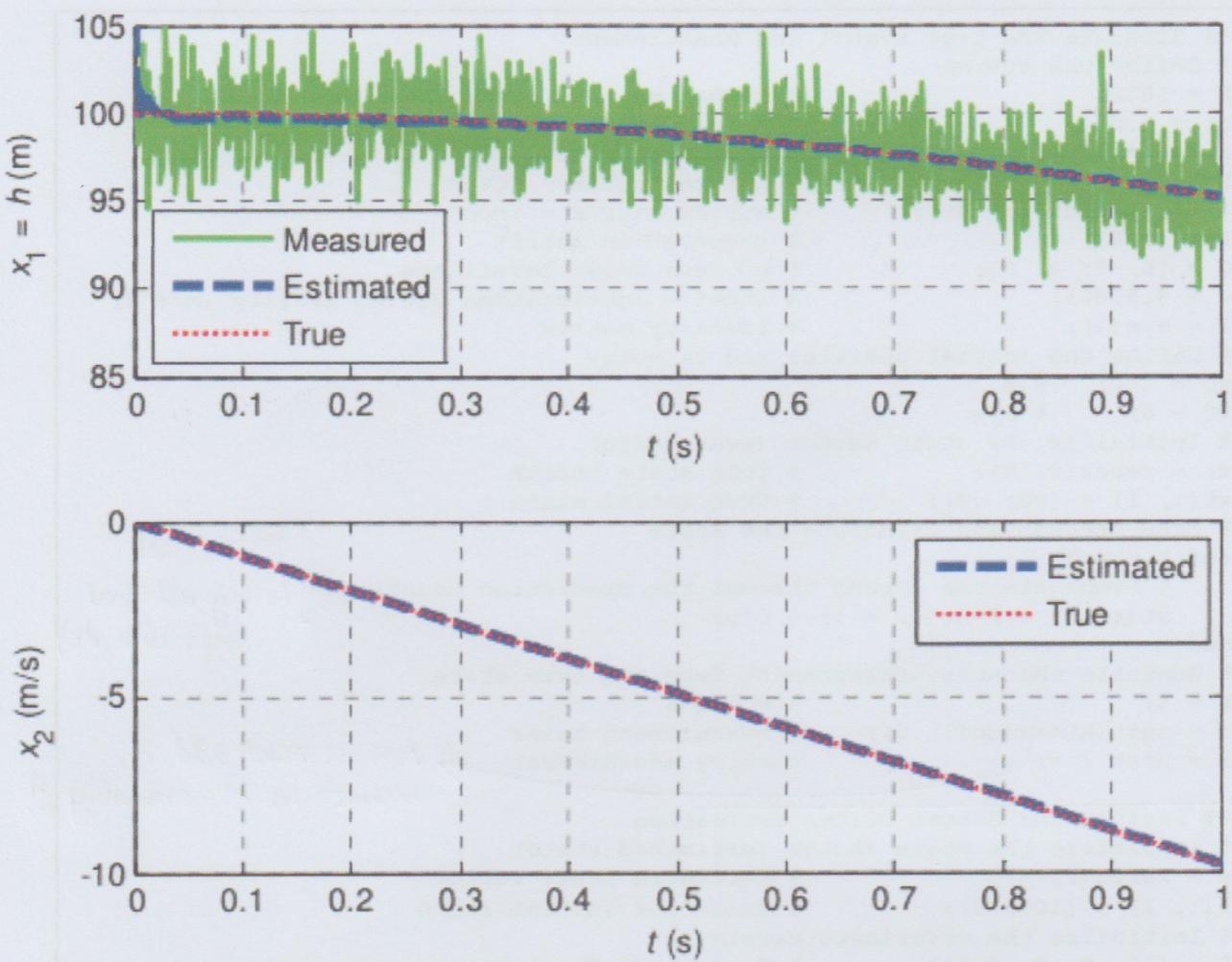
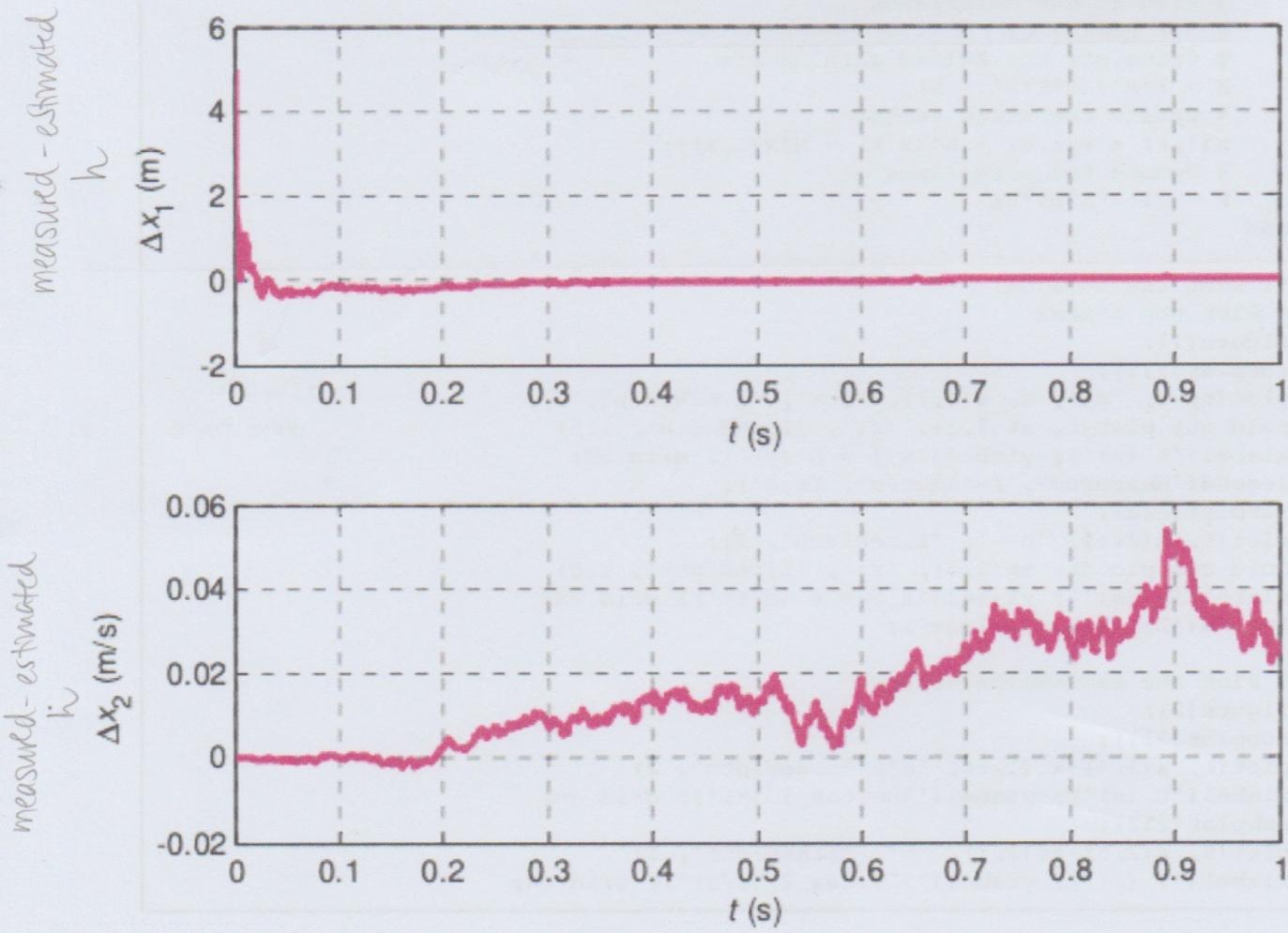


Figure 3. Kalman Filtering Example Estimated and True States



KALMAN FILTER: Non-linear systems

- Two possible options in case we have a non-linear system (state and/or output eq.s):

- Extended Kalman Filter (EKF):

- a function f is used to predict current state \hat{x}_k given previous
- the Jacobian of f gives \tilde{F}
- similarly a function h is used; its Jacobian yields \tilde{H}
- the method is computationally efficient (thus, for realtime applications)

- Unscented Kalman Filter (UKF):

- aka Sigma-Point Kalman filter (SPKF)
- more accurate, but computationally not that efficient
- no Jacobians used, a statistical linearization technique used (unscented transformation)
- more difficult to understand

EXTENDED KALMAN FILTER

LINEAR CASE

$$\begin{aligned}\hat{x}_k &= \tilde{F}_k \hat{x}_{k-1} + \tilde{B}_k u_k + w_k \\ \hat{y}_k &= \tilde{H}_k \hat{x}_k + v_k\end{aligned}$$

EXTENDED / NON-LINEAR CASE

$$\begin{aligned}\hat{x}_k &= f(\hat{x}_{k-1}, u_k) + w_k \\ \hat{y}_k &= h(\hat{x}_k) + v_k\end{aligned}$$

1. Predict

$$F_k = \left. \frac{\partial f}{\partial x} \right|_{\hat{x}_{k-1}}, \quad H_k = \left. \frac{\partial h}{\partial x} \right|_{\hat{x}_k} \quad \text{(Jacobi matrices)}$$

$$\begin{aligned}P_k &= \tilde{F}_k P_{k-1} \tilde{F}_k^T + Q_k \\ K &= P_k H_k^T (H_k P_k H_k^T + R_k)^{-1}\end{aligned} \quad \left. \begin{array}{l} \text{same} \\ \text{as} \\ \text{before} \end{array} \right\}$$

2. Update

$$\begin{aligned}\hat{x}'_k &= \hat{x}_k + K (\hat{z}_k - h(\hat{x}_k)) \\ P'_k &= (I - K H_k) P_k\end{aligned}$$

Tracking in 2D of object:

$$\text{state: } x = [x, y]^T$$

$$\text{measurement: } z = [r, \theta]^T$$

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \tan(y/x)$$

$$\tilde{H} = \frac{\partial h}{\partial x} = \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} \\ \frac{\partial \theta}{\partial x} & \frac{\partial \theta}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} \\ \frac{-y}{x^2+y^2} & \frac{x}{x^2+y^2} \end{bmatrix} \rightarrow \text{this must be evaluated at each time step!!!}$$

KALMAN FILTER

Udemy Course: Exercises with simulators

Kalman Filter in a nutshell:

Variables: 5+3 matrices

- \tilde{x} : state vector : $n_{\text{states}} \times 1$
- $\begin{matrix} \tilde{u} \\ \tilde{B} \end{matrix}$ } input/control: $B\tilde{u} : n_{\text{states}} \times 1$
- \tilde{P} : uncertainty of model : $n_{\text{states}} \times n_{\text{states}}$
- \tilde{F} : state transition matrix : $n_{\text{states}} \times n_{\text{states}}$
- \tilde{H} : measurement matrix: $n_{\text{measurements}} \times n_{\text{states}}$
- \tilde{R} : uncertainty of measurement: $n_{\text{meas}} \times n_{\text{meas}}$
- \tilde{Q} : noise of model: $n_{\text{states}} \times n_{\text{states}}$

Algorithm in 2+1 steps:

- 0) Define initial values of 5+3 matrices

- 1) Predict next state:

$$\hat{\tilde{x}}_k = \tilde{F} \cdot \tilde{x}_{k-1} + \tilde{B} \tilde{u}_k$$

$$\hat{\tilde{P}}_k = \tilde{F} \tilde{P}_{k-1} \tilde{F}^T + \tilde{Q}$$

- 2) Measure and update state with measurement information

\tilde{z} = measurements: they don't need to be the same as the state!

$$\tilde{y} = \tilde{z} - \tilde{H} \hat{\tilde{x}}_k$$

$$\tilde{S} = \tilde{H} \tilde{P}_k \tilde{H}^T + \tilde{R}$$

$$\tilde{K} = \tilde{P}_k \tilde{H}^T \tilde{S}^{-1}$$

$$\begin{cases} \hat{\tilde{x}}_k' = \hat{\tilde{x}}_k + \tilde{K} \tilde{y} \\ \hat{\tilde{P}}_k' = (\tilde{I} - \tilde{K} \tilde{H}) \tilde{P}_k \end{cases}$$

that's
the output!
they become next

$$\tilde{x}_{k-1} \text{ & } \tilde{P}_{k-1}$$

Assignments: a KalmanFilter class is built and passed to a simulation environment which visualizes a simulation; each version of the class has one member method for each of the 2+1 steps of the algorithm:

- 0) `--init--(self)`
- 1) `predict(self, t)`
- 2) `measure_and_update(self, measurements, t)`

NOTE: of all assignments, probably the 3rd is the most interesting for a fast implementation of the Kalman filter...

Assignment ①

```
import numpy as np  
from sim.sim1d import sim_run  
  
# Simulator options.  
options = {}  
options['FIG_SIZE'] = [8,8]  
options['CONSTANT_SPEED'] = False  
  
class KalmanFilterToy:  
    def __init__(self):  
        self.v = 0  
        self.prev_x = 0  
        self.prev_t = 0  
    def predict(self,t):  
        prediction = self.prev_x + self.v * (t - self.prev_t)  
        return prediction  
    def measure_and_update(self,x,t):  
        measured_v = (x - self.prev_x) / (t - self.prev_t)  
        self.v += 0.5 * (measured_v - self.v)  
        self.prev_x = x  
        self.prev_t = t  
        return  
  
sim_run(options,KalmanFilterToy)
```

Dummy example, without Kalman equations yet

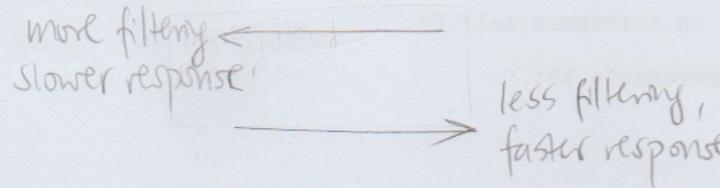
$$x_k = x_{k-1} + \dot{x}_{k-1} \cdot \Delta t$$

\rightarrow We update \dot{x} with what we have measured

previous

weight that determines how much we rely on measurement

usual range: $[0.5 \longleftrightarrow 1]$



Assignment ② 1D Kalman filter: $[x \dot{x}]^T$

```
import numpy as np
from sim.sim1d import sim_run
```

```
# Simulator options.
options = {}
options['FIG_SIZE'] = [8,8]
options['CONSTANT_SPEED'] = False

class KalmanFilter:
    def __init__(self):
        self.v = 0
        self.prev_time = 0
        # Initial State: [x x_dot]
        self.x = np.matrix([[0.0],
                           [0.0]])
    # Uncertainty Matrix: state model uncertainty
    self.P = np.matrix([[1000.0, 0.0],
                       [0.0, 1000.0]])
    # Next State Function: transition from state in k-1 to k
    self.F = np.matrix([[1.0, 1.0],
                       [0.0, 1.0]])
    # Measurement Function: transform state to measurement domain (only x)
    self.H = np.matrix([[1.0, 0.0]])
    # Measurement Uncertainty
    self.R = np.matrix([[0.01]])
    # Identity Matrix
    self.I = np.matrix([[1.0, 0.0],
                       [0.0, 1.0]])
```

```
def predict(self,t):
    # Calculate dt.
    dt = t - self.prev_time
    # Put dt into the state transition matrix.
    self.F[0,1] = dt
    self.x = self.F * self.x
    self.P = self.F * self.P * np.transpose(self.F)
    return self.x[0,0]
```

```
def measure_and_update(self, measurements, t):
    dt = t - self.prev_time
    self.F[0,1] = dt
    # We measure only x.
    Z = np.matrix(measurements)
    y = np.transpose(Z) - (self.H * self.x)
    S = (self.H * self.P * np.transpose(self.H)) + self.R
    K = self.P * np.transpose(self.H) * np.linalg.inv(S)
    # Update values.
    self.x = self.x + (K*y)
    self.P = (self.I - (K*self.H)) * self.P
    self.P[0,0] += 0.1
    self.P[1,1] += 0.1
    print (self.P)
    # Return.
    self.v = self.x[1,0]
    self.prev_time = t
    return
```

```
sim_run(options,KalmanFilter)
```

$$\underline{x} = [x \dot{x}]^T$$

it's going to be updated...
set any reasonable value

$$\begin{cases} x_k = x_{k-1} + \Delta t \dot{x}_{k-1} \\ \dot{x}_k = \dot{x}_{k-1} \end{cases} \rightarrow \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

set later

we measure only x

+ B_u not used
+ \tilde{Q} not used

manual tuning (why?): P decreases a lot,
if we add uncertainty each step, it works
better...

Assignment ③ 2D Kalman filter of a car : $[x \ y \ \dot{x} \ \dot{y}]^T$

```

import numpy as np
from sim.sim2d import sim_run

# Simulator options.
options = {}
options['FIG_SIZE'] = [8,8]

options['DRIVE_IN_CIRCLE'] = False
# If False, measurements will be x,y.
# If True, measurements will be x,y, and current angle of the car.
# Required if you want to pass the driving in circle.
options['MEASURE_ANGLE'] = False
options['RECIEVE_INPUTS'] = False

```

```

class KalmanFilter:
    def __init__(self):
        # Initial State
        self.x = np.matrix([[0.],
                           [0.],
                           [0.],
                           [0.]])
        # Uncertainty Matrix
        self.P = np.matrix([[1000., 0., 0., 0.],
                           [0., 1000., 0., 0.],
                           [0., 0., 1000., 0.],
                           [0., 0., 0., 1000.]])
        # Next State Function
        self.F = np.matrix([[1., 0., 1., 0.],
                           [0., 1., 0., 1.],
                           [0., 0., 1., 0.],
                           [0., 0., 0., 1.]])
        # Measurement Function
        self.H = np.matrix([[1., 0., 0., 0.],
                           [0., 1., 0., 0.]])
        # Measurement Uncertainty
        self.R = np.matrix([[1.0, 0.0],
                           [0.0, 1.0]])
        # Identity Matrix
        self.I = np.matrix([[1., 0., 0., 0.],
                           [0., 1., 0., 0.],
                           [0., 0., 1., 0.],
                           [0., 0., 0., 1.]])

```

```

def predict(self, dt):
    # Calculate dt.
    #dt = t - self.prev_time
    # Put dt into the state transition matrix.
    self.F[0,2] = dt
    self.F[1,3] = dt
    self.x = self.F * self.x
    self.P = self.F * self.P * np.transpose(self.F)
    return

```

```

def measure_and_update(self, measurements, dt):
    #dt = t - self.prev_time
    self.F[0,2] = dt
    self.F[1,3] = dt
    # We measure only x.
    Z = np.matrix(measurements)
    y = np.transpose(Z) - (self.H * self.x)
    S = (self.H * self.P * np.transpose(self.H)) + self.R
    K = self.P * np.transpose(self.H) * np.linalg.inv(S)
    # Update values.
    self.x = self.x + (K*y)
    self.P = (self.I - (K*self.H)) * self.P
    self.P[0,0] += 0.1
    self.P[1,1] += 0.1
    self.P[2,2] += 0.1
    self.P[3,3] += 0.1
    #print (self.P)
    # Return.
    self.v = self.x[1,0]
    #self.prev_time = t
    return [self.x[0], self.x[1]]

```

```

def recieve_inputs(self, u_steer, u_pedal):
    return

```

```

sim_run(options, KalmanFilter)

```

analogous to 1D case

$$\begin{aligned} X_K &= X_{k-1} + \Delta t \dot{X}_{k-1} \\ Y_K &= Y_{k-1} + \Delta t \dot{Y}_{k-1} \\ \dot{X}_K &= \dot{X}_{k-1} \\ \dot{Y}_K &= \dot{Y}_{k-1} \end{aligned} \quad \left[\begin{array}{cccc} x & y & \dot{x} & \dot{y} \\ 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right]$$

set later

uncertainty of model is manually increased in each step because the algorithm decreases it, and with higher values it works better (why?)

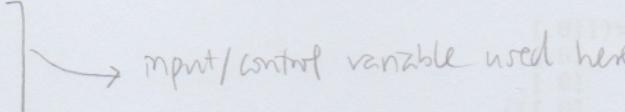
This function is not used here — it consider the input with another state model : $[x \ y \ \omega \ \theta]^T$

Assignment ④: Prediction — when a car arrives to a traffic lamp which gets yellow, we need to predict whether we can cross or not with and without velocity change (acceleration).

Basically Δt is artificially changed in E to check where we would be.

Prediction implemented in

- `predict_red_light(self, light_location)`, and
- `predict_red_light_speed(self, light_location)`



```

import numpy as np
from sim.sim2d_prediction import sim_run

# Simulator options.
options = {}
options['FIG_SIZE'] = [8,8]
options['ALLOW_SPEEDING'] = True

class KalmanFilter:
    def __init__(self):
        # Initial State
        self.x = np.matrix([[55.],
                           [3.],
                           [5.],
                           [0.]])
        # Input vector
        self.u = np.matrix([[0],
                           [0.],
                           [0.],
                           [0.]])
        # Uncertainty Matrix
        self.P = np.matrix([[1000., 0., 0., 0.],
                           [0., 1000., 0., 0.],
                           [0., 0., 1000., 0.],
                           [0., 0., 0., 1000.]])
        # Next State Function
        self.F = np.matrix([[1., 0., 1., 0.],
                           [0., 1., 0., 1.],
                           [0., 0., 1., 0.],
                           [0., 0., 0., 1.]])
        # Measurement Function
        self.H = np.matrix([[1., 0., 0., 0.],
                           [0., 1., 0., 0.]])
        # Measurement Uncertainty
        self.R = np.matrix([[1.0, 0.0],
                           [0.0, 1.0]])
        # Identity Matrix
        self.I = np.matrix([[1., 0., 0., 0.],
                           [0., 1., 0., 0.],
                           [0., 0., 1., 0.],
                           [0., 0., 0., 1.]])
    def predict(self, dt):
        self.F[0,2] = dt
        self.F[1,3] = dt
        self.x = self.F * self.x
        self.P = self.F * self.P * np.transpose(self.F)
        return
    def measure_and_update(self, measurements, dt):
        self.F[0,2] = dt
        self.F[1,3] = dt
        # We measure only x.
        Z = np.matrix(measurements)
        y = np.transpose(Z) - (self.H * self.x)
        S = (self.H * self.P * np.transpose(self.H)) + self.R
        K = self.P * np.transpose(self.H) * np.linalg.inv(S)
        # Update values.
        self.x = self.x + (K*y)
        self.P = (self.I - (K * self.H)) * self.P
        self.P[0,0] += 0.1
        self.P[1,1] += 0.1
        self.P[2,2] += 0.1
        self.P[3,3] += 0.1
        #print (self.P)
        # Return.
        self.v = self.x[1,0]
        #self.prev_time = t
        return [self.x[0], self.x[1]]

```

```

def predict_red_light(self, light_location):
    # Light turns yellow and from yellow to red in 3 seconds
    # Is it possible to cross the street in yellow?
    # Approach:
    #   Predict x with dt = 3 seconds
    #   Check if x < light_location_x
    light_duration = 3
    F_new = np.copy(self.F)
    F_new[0,2] = light_duration
    F_new[1,3] = light_duration
    x_new = F_new * self.x + self.u
    if x_new[0] < light_location:
        return [False, x_new[0]]
    else:
        return [True, x_new[0]]

```

explanation

```

def predict_red_light_speed(self, light_location):
    # Light turns yellow and from yellow to red in 3 seconds
    # We can increase speed 1.5 m/s in 1 s
    # Is it possible to cross the street in yellow?
    # Approach:
    #   First call previous function
    #   If it doesn't work
    #       predict x in 1 s with u = 1.5
    #       predict x with previous x in 2 s
    #       check if x < light_location_x
    check = self.predict_red_light(light_location)
    if check[0]:
        return check
    light_duration = 3
    F_new = np.copy(self.F)
    u_new = np.copy(self.u)
    u_new[2] = 1.5
    F_new[0,2] = 1
    F_new[1,3] = 1
    x_new = F_new * self.x + u_new
    F_new[0,2] = light_duration - 1
    F_new[1,3] = light_duration - 1
    x_new = F_new * x_new
    if x_new[0] < light_location:
        return [False, x_new[0]]
    else:
        return [True, x_new[0]]

```

explanation

```

for i in range(0,5):
    sim_run(options, KalmanFilter, i)

```