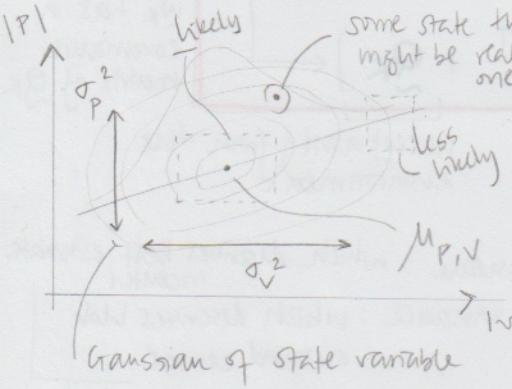
KALMAN FILTER

Bzarg 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



- $\hat{\underset{\sim}{X}}_K = [\vec{P}, \vec{v}]^T$ → usually we put $\hat{X}_K$, because X is an estimation (^): state vector of a system at moment k
  - $\vec{P}$: position vector
  - $\vec{v}$: velocity vector
  - } random Gaussian variables
- $\underset{\sim}{\mu}_{P,V}$: mean state value of Gaussian state
- σ_P^2, σ_V^2: variance of state (P: position, v: velocity)

$\underset{\sim}{P}_K$ (at moment k) = COVARIANCE MATRIX

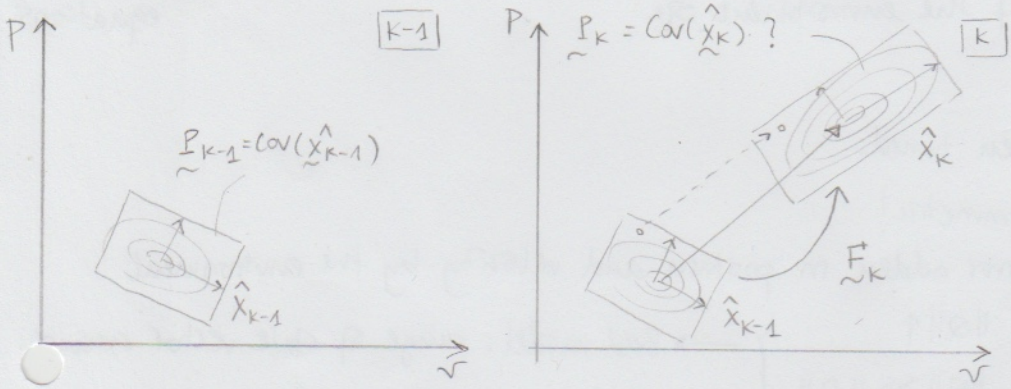
$$\begin{bmatrix} \sigma_{PP}^2 & \sigma_{PV}^2 \\ \sigma_{VP}^2 & \sigma_{VV}^2 \end{bmatrix} = \text{Cov}(\underset{\sim}{X}_K) = \underset{\sim}{\Sigma}$$

($\sigma_{PP}^2 = \sigma_P^2$; covariance σ_{VP})

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

- State $\underset{\sim}{X}$ can be built of different signals that can be noisy / uncertain → VARIANCE
- Different components of state can be correlated: $|\vec{v}| \uparrow \rightarrow |\vec{P}| \uparrow$

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



- Prediction step taken applying system's equations
- Every point in $\hat{X}_{k-1}$ is transformed into $\hat{X}_K$

$$P_K = P_{k-1} + \Delta t\, v_{k-1}$$
$$V_K = V_{k-1}$$

$$\begin{bmatrix} P_K \\ V_K \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}}_{\underset{\sim}{F}_K} \begin{bmatrix} P_{k-1} \\ V_{k-1} \end{bmatrix}$$

$\underset{\sim}{F}_K$: PREDICTION MATRIX

- Since: $\text{Cov}(\underset{\sim}{X}) = \underset{\sim}{\Sigma} \Leftrightarrow \text{Cov}(\underset{\sim}{A}\underset{\sim}{X}) = \underset{\sim}{A}\underset{\sim}{\Sigma}\underset{\sim}{A}^T$

- PREDICTION STEP:

$$\hat{\underset{\sim}{X}}_K = \underset{\sim}{F}_K \cdot \hat{\underset{\sim}{X}}_{K-1}$$ → state variable in next step is predicted

$$\underset{\sim}{P}_K = \underset{\sim}{F}_K \underset{\sim}{P}_{K-1} \underset{\sim}{F}_K^T$$ → Covariance matrix of state variable is predicted

Note: we need only $\underset{\sim}{F}_K$ to go from k-1 to k

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

③ External influence: control matrix

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

$$\hat{\underset{\sim}{X}}_K = \underset{\sim}{F}_K \hat{\underset{\sim}{X}}_{k-1} + \underbrace{\begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix}}_{\underset{\sim}{B}_K} \underbrace{a}_{\underset{\sim}{u}_K}$$

aka $\underset{\sim}{G}$

$$\hat{\underset{\sim}{X}}_K = \underset{\sim}{F}_K \hat{\underset{\sim}{X}}_{k-1} + \underset{\sim}{B}_K \underset{\sim}{u}_K$$

- $\underset{\sim}{B}_K$: CONTROL MATRIX
- $\underset{\sim}{u}_K$: CONTROL VECTOR, $a = f/m$

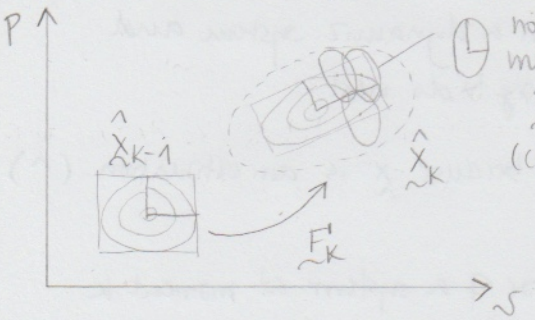
These are related to the forces we apply to control and we know them

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

Note:
- $\underset{\sim}{B}$ / $\underset{\sim}{G}$: matrix / vector
- $\underset{\sim}{u}$: vector / scalar

depending on problem

④ External uncertainty: noise (eg: wind, wheel slips, ...) → we could extend our model if we knew them, but we don't know them ...



noise with mean ∅: Q_k (covariance) → we add the Gaussian blob to our prediction

- We add to the prediction a noise component:
  - mean is ∅
  - variance is increased → covariance is updated

$$\hat{X}_k = F_k \cdot \hat{X}_{k-1} + B_k u_k + w_k$$

$$P_k = F_k P_{k-1} F_k^T + Q_k$$

w_k: mean ∅! (so it's not used); w_k has a covariance matrix of Q_k

Q_k: uncertainty from the environment

[1] PREDICTION STEP

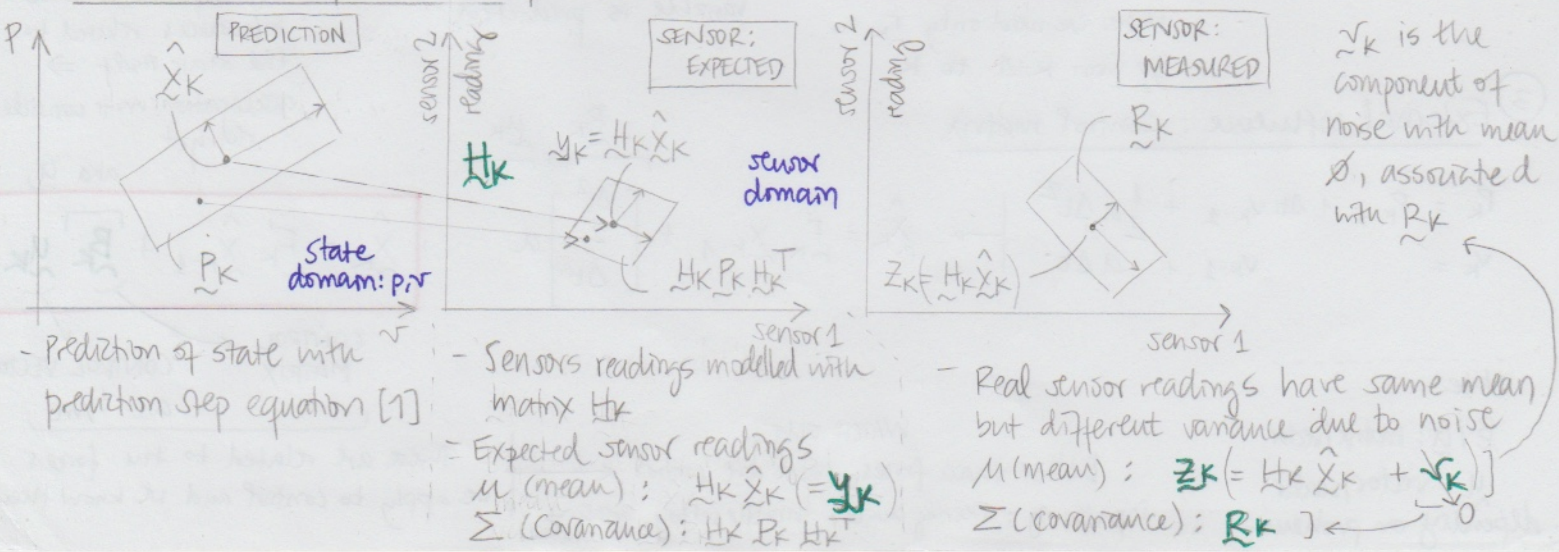
⑤ Insights from points 1-4

- 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

In other words, the distribution of a variable is defined with mean and variance equations

- NEW STATE ESTIMATE ($\hat{X}_k$) is a PREDICTION made from
  - previous state $\hat{X}_{k-1}$ estimate and system's (linear) model (F_k), and
  - correction for external influences / control ($B_k u_k = v_k$)
- NEW UNCERTAINTY (VARIANCE), (P_k) is a PREDICTION made from:
  - previous uncertainty P_{k-1} and system's (linear) model (F_k), and
  - additional uncertainty of the environment Q_k
- How to choose Q_k?
  - Usually hand-chosen, then tuned
  - $Q_k = \begin{bmatrix} PP & PV \\ VP & VV \end{bmatrix}$
    - Symmetric!
    - error added in position and velocity by the environment
    - if $\|Q\| \uparrow$
    - if $\|Q\| >> \|P\|$
    - → Bad model: range of state vector extends
    - if 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

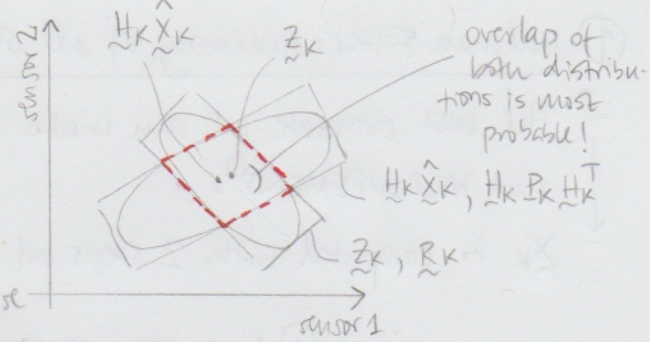


v_k is the component of noise with mean ∅, associated with R_k

- Prediction of state with prediction step equation [1]
- Sensors readings modelled with matrix H_k
  - = Expected sensor readings
  - μ (mean): $H_k \hat{X}_k$ (= y_k)
  - Σ (covariance): $H_k P_k H_k^T$
- Real sensor readings have same mean but different variance due to noise
  - μ (mean): $z_k (= H_k \hat{X}_k + v_k)$ (v_k: ∅)
  - Σ (covariance): R_k

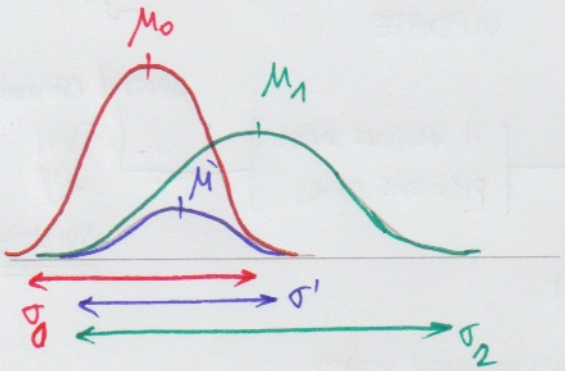
	PREDICTION	SENSOR: EXPECTED	SENSOR: MEASURED
μ	$\hat{x}_k$	$(y_k = H_k \hat{x}_k)$	$z_k (= H_k \hat{x}_k + v_k^{\to 0})$
Σ	P_k	$H_k P_k H_k^T$	R_k

expected and measured are usually not the same ← expected model | measured value → Covariance due to sensor noise



- We have 2 Gaussian blobs { expected: μ: $H_k \hat{x}_k$, Σ: $H_k P_k H_k^T$; measured: μ: z_k, Σ: 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 $N'(x, \mu', \sigma') \iff$ which are μ', σ'?

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

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

- It can be obtained:

1D dimension

$$\mu_{fused} = \mu' = \mu_0 + \frac{\sigma_0^2(\mu_1 - \mu_0)}{\sigma_0^2 + \sigma_1^2}$$

$$\sigma_{fused} = \sigma' = \sigma_0^2 - \frac{\sigma_0^4}{\sigma_0^2 + \sigma_1^2}$$

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

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

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

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

We extract constant K

n Dimensions

[2] FUSION

$$K = \Sigma_0 (\Sigma_0 + \Sigma_1)^{-1}$$

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

$$\Sigma' = \Sigma_0 - K\Sigma$$

Equivalent with matrices $\iff$ 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 { $(\mu_0, \Sigma_0) = (H_k \hat{x}_k, H_k P_k H_k^T)$: expected ; $(\mu_1, \Sigma_1) = (z_k, R_k)$: measured
- Applying Fusion eq.s [2] (in sensor domain):

- $\underbrace{H_k x_k'}_{\mu'} = \underbrace{H_k x_k}_{\mu_0} + K\left(\underbrace{z_k}_{\mu_1} - \underbrace{H_k x_k}_{\mu_0}\right)$
- $\underbrace{H_k P_k' H_k^T}_{\Sigma'} = \underbrace{H_k P_k H_k^T}_{\Sigma_0} - \underbrace{K H_k P_k H_k^T}_{\Sigma_0}$
- $K = \underbrace{H_k P_k H_k^T}_{\Sigma_0} \cdot \left(\underbrace{H_k P_k H_k^T}_{\Sigma_0} + \underbrace{R_k}_{\Sigma_1}\right)^{-1}$

[3] UPDATE ▾ update/ fused state!

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

$$P_k' = P_k - K' H_k P_k$$

$$K' = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1}$$

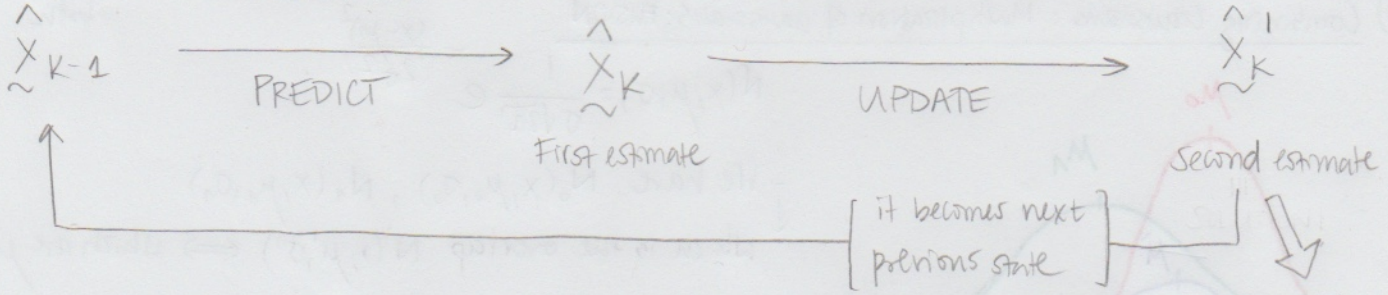
H_k is removed: we go from sensor to state domain

⑨ Kalman Filter: summary of all steps

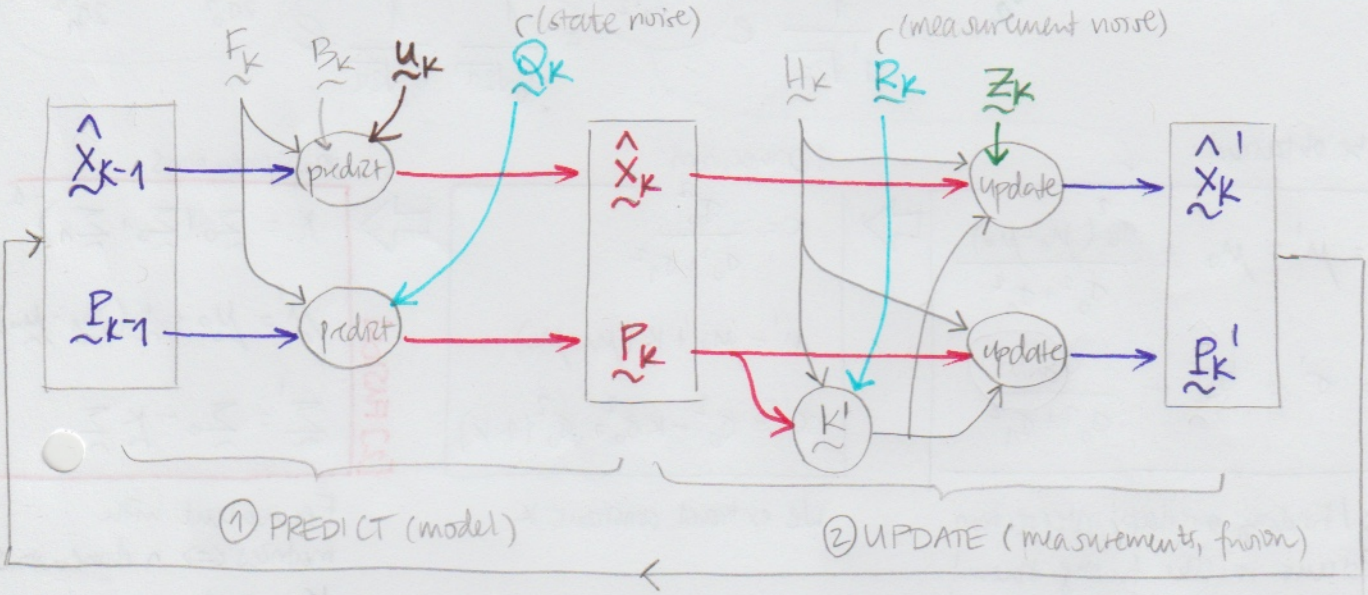
- "The best estimate we can make is provided by combining our knowledge from PREDICTION and MEASUREMENT".
- $\hat{x}_k$ is computed with 2 steps at each cycle:

① First estimate of next state is predicted with model equations taking into account uncertainties

② Second estimate of next state is obtained by fusing/overlapping first estimate in sensor domain & sensor readings



⑩ Complete kalman Filter Information Flow (Bzarg.com)



[NOTES]

- How can we obtain P, Q, R, H? Examples? ⟶ see Kalman Filter Tutorial: Rhudy, 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)

KALMAN FILTER

Tutorial for Undergraduates — Rhudy, Salguero, 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

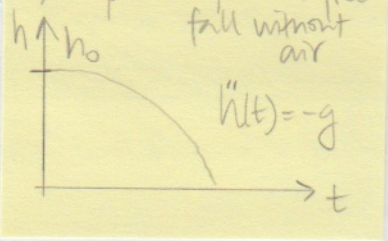
	① PREDICT	② UPDATE	
μ	$\hat{x}_k = F_k \hat{x}_{k-1} + B_k u_k + \cancel{w_k}^{\to 0}$	$y_k = H_k x_k + \cancel{v_k}^{\to 0}$	z_k
Σ	$P_k = F_k P_{k-1} F_k^T + Q_k$	$H_k P_k H_k^T$	R_k
		EXPECTED	MEASURED

$$\hat{x}'_k = \hat{x}_k + k'(z_k - H_k \hat{x}_k)$$

$$P'_k = (I - K' H_k) P_k$$

$$K' = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1}$$

Example: object in free-fall without air



Variable	Description	Dimension	Example	Comments/Notes
x_k	state vector	$n_x \times 1$	$x_k = [h \ \dot{h}]^T$	- Desired result! - It might include more variables that the really wanted if necessary. - STATE DOMAIN!
F_k	system matrix of state	$n_x \times n_x$	$F_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$ (in this case it's constant)	- Always square: $\dim(x) \times \dim(x)$ - Linear transformation applied to previous state to obtain current new state (without considering input/control variables) $x_k = \begin{bmatrix} h_k \\ \dot{h}_k \end{bmatrix} = \begin{bmatrix} h_{k-1} + \dot{h}_{k-1}\Delta t - \frac{1}{2} g \Delta t^2 \\ \dot{h}_{k-1} - g\Delta t \end{bmatrix}$ $= \underbrace{\begin{bmatrix} 1 & \Delta t \\ & 1 \end{bmatrix}}_{F_k} \underbrace{\begin{bmatrix} h_{k-1} \\ \dot{h}_{k-1} \end{bmatrix}}_{x_{k-1}} + \underbrace{\begin{bmatrix} -\frac{1}{2}\Delta t^2 \\ -\Delta t \end{bmatrix}}_{B_k} \underbrace{g}_{u_k}$
u_k	input or control vector	$n_u \times 1$	$u_k = g$ (in this case it's constant)	- It can be a vector or a scalar · if vector, B_k will be a matrix · if scalar, B_k will be a vector

(Cont.)

Variable	Description	Dimension	Example	Comments / Notes
$\underset{\sim}{u}_k$				- information related to the system dynamics - any other terms which appear in the filter that don't need to be estimated as states can be inputs - typical case: acceleration
$\underset{\sim}{B}_k$	system matrix of input/control	$n_x \times n_u$	$\underset{\sim}{B}_k = \begin{bmatrix} -\frac{\Delta t^2}{2} \\ -\Delta t \end{bmatrix}$ (in this case it's constant)	- It can be a vector or a matrix, depending on n_u (see $\underset{\sim}{u}_k$) - It contains the coefficients to apply to the input / control vector
$\underset{\sim}{y}_k$	output vector	$n_y \times 1$	$\underset{\sim}{y}_k = \underset{\sim}{H}_k \underset{\sim}{x}_k + \underset{\sim}{v}_k$ (noise) $= h_k$	- $\underset{\sim}{y}_k$ is the transformation of our state prediction to the sensor domain: it's the sensor measurement we'd expect!!
$\underset{\sim}{H}_k$	observation matrix	$n_y \times n_x$	$\underset{\sim}{H}_k = [1 \;\; 0]$ (in this case $\underset{\sim}{H}_k$ is constant)	- $\underset{\sim}{H}_k$ transforms the predicted state to the sensor domain - $\underset{\sim}{H}_k$ is a linear transformation that can SELECT: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_3 \end{bmatrix}$; SCALE: $2 \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix}$; COMBINE: $[1 \;\; 1]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 + x_2$ - IMPORTANT: sensor domain doesn't need to have same or equivalent dimensions as state domain!!!
$\underset{\sim}{w}_k$	process noise vector	$n_x \times 1$	mean $\varnothing$	- these noise vectors have mean $\varnothing$, thus they don't typically appear in the equations of interest - However, they have a variance component ⇒ the blob of the corresponding distribution is considered to be modified
$\underset{\sim}{v}_k$	measurement noise vector	$n_y \times 1$	mean $\varnothing$	
$\underset{\sim}{Q}_k$	covariance of the process noise	$n_x \times n_x$	$\underset{\sim}{Q}_k = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$	- Process noise ($\underset{\sim}{w}_k$, $\underset{\sim}{Q}_k$) is related to the error that might be in the model equations, e.g., we could consider the effects of wind or air resistance - BUT: if equations/model are considered accurate enough, we can set $\underset{\sim}{Q} = \varnothing$.
$\underset{\sim}{R}_k$	covariance of the measurement	$n_y \times n_y$	$\underset{\sim}{R}_k = 4$ (they can be constant)	

(cont.)

Variable	Description	Dimension	Example	Comments / Notes
$\underset{\sim}{Q}_K$ $\underset{\sim}{R}_K$				- Measurement covariance can be derived from sensor data sheet: if standard deviation of measurement error is assumed 2m $\rightarrow$ $\underset{\sim}{R}_K = 4m^2$ - IMPORTANT: $\underset{\sim}{Q}$ & $\underset{\sim}{R}$ affect considerably the performance of the filter; they are like weighting factors between the prediction (state) equations and the measurement (output) equations: $\lvert K \rvert \sim \frac{\lvert Q \rvert}{\lvert Q \rvert + \lvert R \rvert}$ $\underset{\sim}{Q}\uparrow$ larger uncertainty in state equations, correct more with measurement $\underset{\sim}{R}\uparrow$ larger uncertainty in measurement, correct less with measurement - The kalman gain controls the impact of the measurement correction: $\underset{\sim}{\hat{X}}_K' = \underset{\sim}{\hat{X}}_K + \underset{\sim}{K}'(\underset{\sim}{Z}_K - \underset{\sim}{H}_K \underset{\sim}{\hat{X}}_K)$ updated state; predicted first state; diff between measured and expected states in sensor domain
$\underset{\sim}{K}$	kalman gain	$n_x \times n_y$	Computed each step, see formula	
$\underset{\sim}{P}_0$	assumed initial state error cov. matrix	$n_x \times n_x$	$\underset{\sim}{P}_0 = \begin{bmatrix} 10 & 0 \\ 0 & 0.01 \end{bmatrix}$ (h, ḣ)	- $\underset{\sim}{P}_0$ is related to the assumed initial state $\underset{\sim}{x}_0$ and it affects the initial convergence of the filter. - It is updated to $\underset{\sim}{P}_K$ in the following steps - Its value is not that important, sometimes it's initialized to identity matrix - In our case the ball starts at $\underset{\sim}{x}_0 = [100\ 0]^T$, but we assume $\underset{\sim}{\hat{x}}_0 = [105\ 0]$ with a non-zero velocity of 0.1 m/s $\rightarrow$ $0.01\ m^2/s^2$
$\underset{\sim}{x}_0$	true initial state vector	$n_x \times 1$	$[100\ 0]^T$	Note: in the absence of info, we can guess initial values of $\underset{\sim}{P}$ by observing expected & measured values: if big diff, big values in $\underset{\sim}{P}$.
$\underset{\sim}{\hat{x}}_0$	assumed initial state vector	$n_x \times 1$	$[105\ 0]^T$	
Δt	step time or time increment	1×1	0.001s	

Notes on the covariance matrix

$$\underset{\sim}{X} = \begin{bmatrix} X_1 & \cdots & X_m \end{bmatrix}^T \Longrightarrow$$

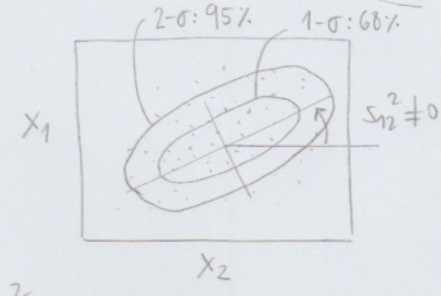
m: size/dim of $\underset{\sim}{X}$ = # of vars

n: # samples (i)

$$\underset{\sim}{X_1} \quad \cdots \quad \underset{\sim}{X_n}$$

$$\begin{bmatrix} X_{11} \\ X_{12} \\ | \\ X_{1m} \end{bmatrix} \quad \cdots \quad \begin{bmatrix} X_{n1} \\ X_{n2} \\ | \\ X_{nm} \end{bmatrix}$$

You can also compute your covariance matrix by performing measurements...



$$Cov(\underset{\sim}{X}) = \begin{bmatrix} s_1^2 & & s_{jk}^2 \\ & s_2^2 & \\ & & \\ sim & & s_m^2 \end{bmatrix}$$

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

$$s_j^2 = \frac{1}{n}\sum_{i=1}^{n}(\bar{X_j} - X_{ij})^2 \quad \text{of var. } j \text{ along all samples}$$

$$s_{jk}^2 = \frac{1}{n}\sum_{i=1}^{n}(\bar{X_j} - X_{ij})(\bar{X_k} - X_{ik}) \quad \text{between vars } j \text{ \& } k \text{ along all samples}$$

$$\bar{X_j} = \frac{1}{n}\sum_{i=1}^{n} X_{ij} \quad \text{mean of var } j$$

$$\underset{\sim}{X} = \begin{bmatrix} p & v \end{bmatrix}^T$$

$$Cov(\underset{\sim}{X}) = \begin{bmatrix} s_p^2 & s_{pv} \\ s_{vp} & s_v^2 \end{bmatrix}$$

$$s_p^2 = \frac{1}{n}\sum_{i=1}^{n}(\bar{p} - p_i)^2$$

$$s_v^2 = \frac{1}{n}\sum_{i=1}^{n}(\bar{v} - v_i)^2$$

$$s_{pv} = s_{vp} = \frac{1}{n}\sum_{i=1}^{n}(\bar{p} - p_i)(\bar{v} - v_i)$$

Correlation matrix

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

$$\underset{\sim}{R} = \begin{bmatrix} 1 & r_{12} & r_{13} & \cdots & r_{1m} \\ & 1 & r_{23} & \cdots & r_{2m} \\ & & 1 & & \\ & sim & & & | \\ & & & & 1 \end{bmatrix}$$

$$r_{jk} = \frac{s_{jk}^2}{s_j s_k} = \frac{\sum_{i=1}^{n}(\bar{X_j} - X_{ij})(\bar{X_k} - X_{ik})}{\sqrt{\sum_{i=1}^{n}(\bar{X_j} - X_{ij})^2 \sum_{i=1}^{n}(\bar{X_k} - X_{ik})^2}}$$

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

Rhudy, Salguero, Holappa 2017

```
%% 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 intial 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

%% 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;
    % Predict the covariance
    P = F*P*F' + Q;
    % Calculate the Kalman gain matrix
    K = P*H'/(H*P*H' + R);
    % 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('\Deltax_1 (m)'); grid on;
subplot(212);
plot(t, x(2,:)-xt(2,:), 'm', 'LineWidth', 2)
xlabel('t (s)'); ylabel('\Deltax_2 (m/s)'); grid on;
```

$G = B$

PREDICTION (along all time steps with dt)

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

1. PREDICT

2. UPDATE

plots on next page

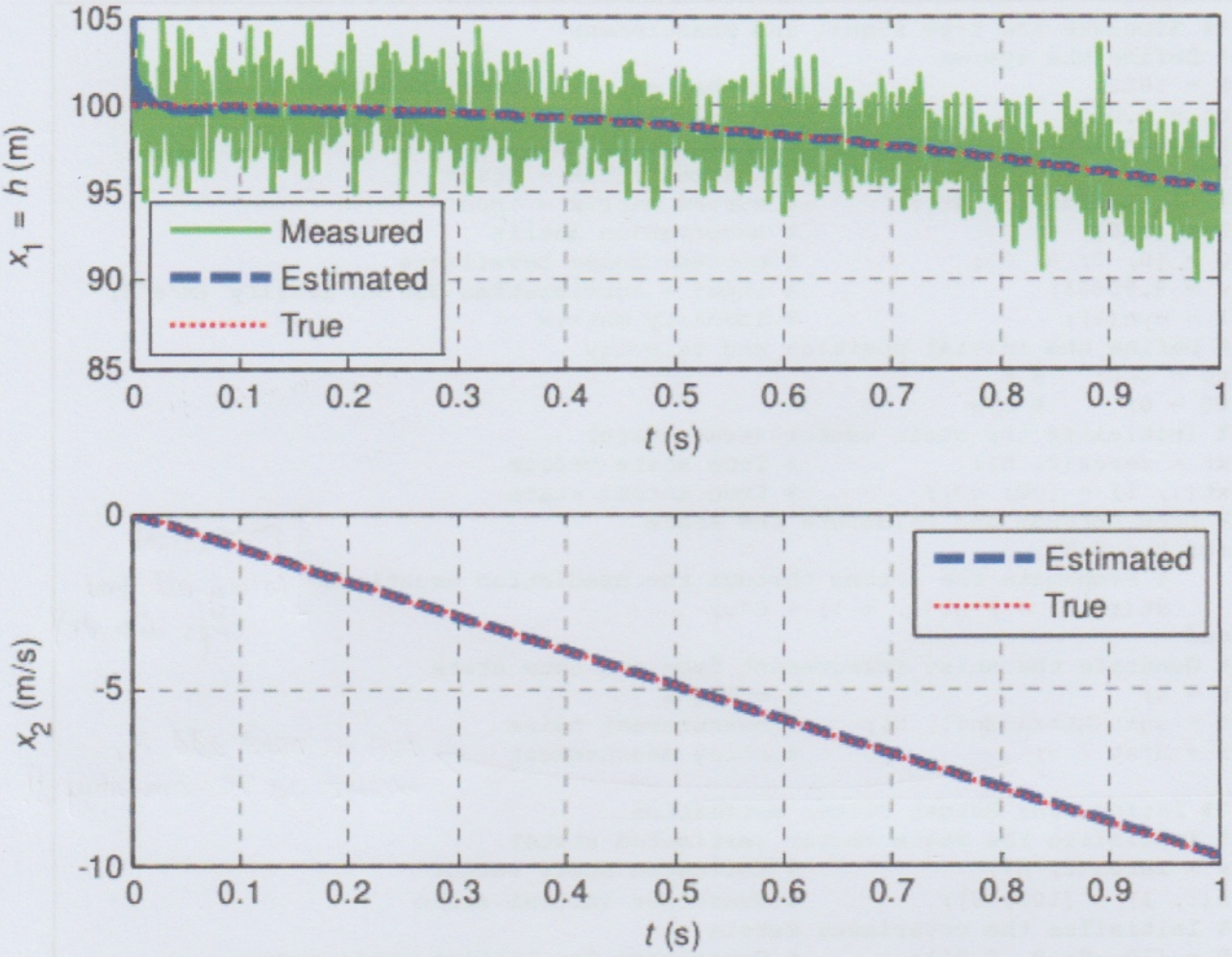
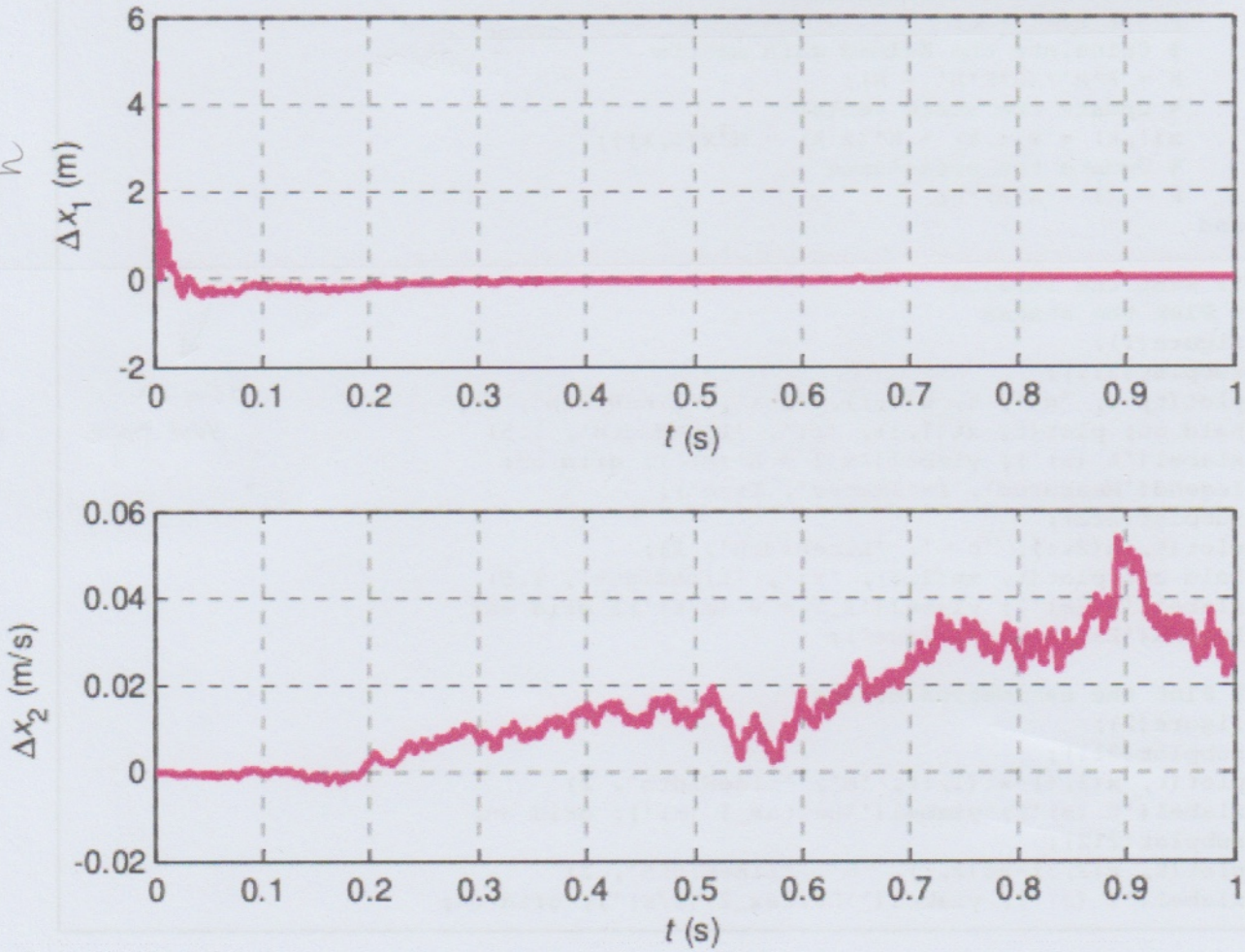


Figure 3. Kalman Filtering Example Estimated and True States



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 x_k given previous
  - the Jacobian of f gives F
  - similarly a function h is used; its Jacobian yields 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

$$\hat{x}_k = F_k \hat{x}_{k-1} + B_k u_k + \cancel{w_k}^{\,0}$$

$$\hat{y}_k = H_k \hat{x}_k + \cancel{v_k}^{\,0}$$

$\Rightarrow$

EXTENDED / NON-LINEAR CASE

$$\hat{x}_k = f(\hat{x}_{k-1}, u_k) + \cancel{w_k}^{\,0}$$

$$\hat{y}_k = h(\hat{x}_k) + \cancel{v_k}^{\,0}$$

1. Predict

$$F_k = \left.\frac{\partial f}{\partial x}\right|_{\hat{x}_{k-1}} \qquad H_k = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_k}$$

(Jacobi matrices)

$$P_k = F_k P_{k-1} F_k^T + Q_k$$

$$K = P_k H_k^T \left(H_k P_k H_k^T + R_k\right)^{-1}$$

} same as before

2. Update

$$\hat{x}'_k = \hat{x}_k + K\left(z_k - h(\hat{x}_k)\right)$$

$$P'_k = (I - K H_k) P_k$$

example

Tracking in 2D of object:

state: $x = [x, y]^T$

measurement: $y = [r, \theta]^T$

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

$$\theta = \operatorname{atan}(y/x)$$

$$H = \frac{\partial h}{\partial x} = \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} \\ \frac{\partial \theta}{\partial x} & \frac{\partial \theta}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} \\ \frac{-y}{x^2+y^2} & \frac{x}{x^2+y^2} \end{bmatrix}$$

→ this must be evaluated at each time step !!!

KALMAN FILTER — Udemy Course: Exercises with simulators

Kalman Filter in a nutshell:

Variables: 5+3 matrices

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

Algorithm in 2+1 steps:

0) Define initial values of 5+3 matrices

1) Predict next state:

$$\hat{\underset{\sim}{x}}_k = \underset{\sim}{F} \cdot \underset{\sim}{x}_{k-1} \left(+ \underset{\sim}{B}\underset{\sim}{u}_k \right)$$

$$\hat{\underset{\sim}{P}}_k = \underset{\sim}{F}\,\underset{\sim}{P}_k\,\underset{\sim}{F}^T \left(+ \underset{\sim}{Q} \right)$$

2) Measure and update state with measurement information

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

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

$$\underset{\sim}{S} = \underset{\sim}{H}\hat{\underset{\sim}{P}}_k\underset{\sim}{H}^T + \underset{\sim}{R}$$

$$\underset{\sim}{K} = \underset{\sim}{P}\underset{\sim}{H}^T\underset{\sim}{S}^{-1}$$

$$\hat{\underset{\sim}{x}}'_k = \hat{\underset{\sim}{x}}_k + \underset{\sim}{K}\underset{\sim}{y}$$

$$\hat{\underset{\sim}{P}}'_k = \left(\underset{\sim}{I} - \underset{\sim}{K}\underset{\sim}{H}\right)\hat{\underset{\sim}{P}}_k$$

that's the output! they become next $\underset{\sim}{x}_{k-1}$ & $\underset{\sim}{P}_{k-1}$

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

Dummy example, without kalman equations yet

```
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
```

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

] -> We update $\dot{x}$ with what we have measured

```
sim_run(options,KalmanFilterToy)
```

weight that determines how much we rely on previous measurement

usual range: [0.5 ⟷ 1]

more filtering ← slower response

less filtering, faster response →

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]])
        # Uncertainity Matrix: state model uncertainty
        self.P = np.matrix([[1000.0, 0.0],
                            [0.0, 1000.0]])
        # Next State Function: transtion 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)
```

$\underset{\sim}{x} = [x\ \dot{x}]^T$

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

$+\underset{\sim}{B}\underset{\sim}{u}$ not used

$+\underset{\sim}{Q}$ not used

manual tuning (why?): $\underset{\sim}{P}$ 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.]])
        # Uncertainity 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

$$x_K = x_{K-1} + \Delta t\,\dot{x}_{K-1}$$
$$y_K = y_{K-1} + \Delta t\,\dot{y}_{K-1}$$
$$\dot{x}_K = \dot{x}_{K-1}$$
$$\dot{y}_K = \dot{y}_{K-1}$$

$$\begin{matrix} x & y & \dot{x} & \dot{y} \end{matrix}$$
$$\begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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 : $[x\ y\ \psi\ \theta\ \dot{\theta}]^T$

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.]])
        # Uncertainity 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]]
```

input/control variable used here

```
    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]]

    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 doesnt 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

→ manual override of Δt for prediction

u is directly Bu

explanation

→ acceleration

→ manual override of Δt for prediction

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