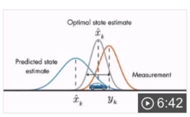
Localization and sensor fusion

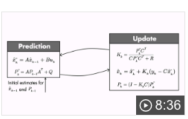
Table of Contents

Product of two PDF's 1
Stimating the tricycle pose 2
Using the tricycle model. Involved equations........ 2
Pose without Noise........ 3
With noise........ 4
Driving and running the simulation 5
Simulating to get the Taylor + Ricatti........ 12
Plotting the results........ 12
Map-based localization........ 13
Build the Map 13
Sensor data........ 14
Sensor on the vehiche........ 14
Reading the sensor - 'errors ?' 15
Laser innovation & making the fusion........ 15
Prediction: Taylor and Riccati........ 15
Updating........ 15
Plotting trajectories and ellipse errors........ 17
Plotting statistics........ 18

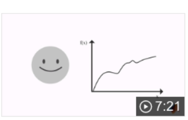
For better understanding how Kalman Filter aplies to stimate Robot position See the video Part3, Prt4 and Part5 at URL: https://es.mathworks.com/videos/series/understanding-kalman-filters.html



Part 3: An Optimal State Estimator
Learn how Kalman filters work. Kalman filters combine two sources of information, the predicted states and noisy measurements, to produce optimal, unbiased state estimates.



Part 4: An Optimal State Estimator Algorithm
Discover the set of equations you need to implement the Kalman filter algorithm.



Part 5: Nonlinear State Estimators
This video explains the basic concepts behind nonlinear state estimators, including extended Kalman filters, unscented Kalman filters, and particle filters.

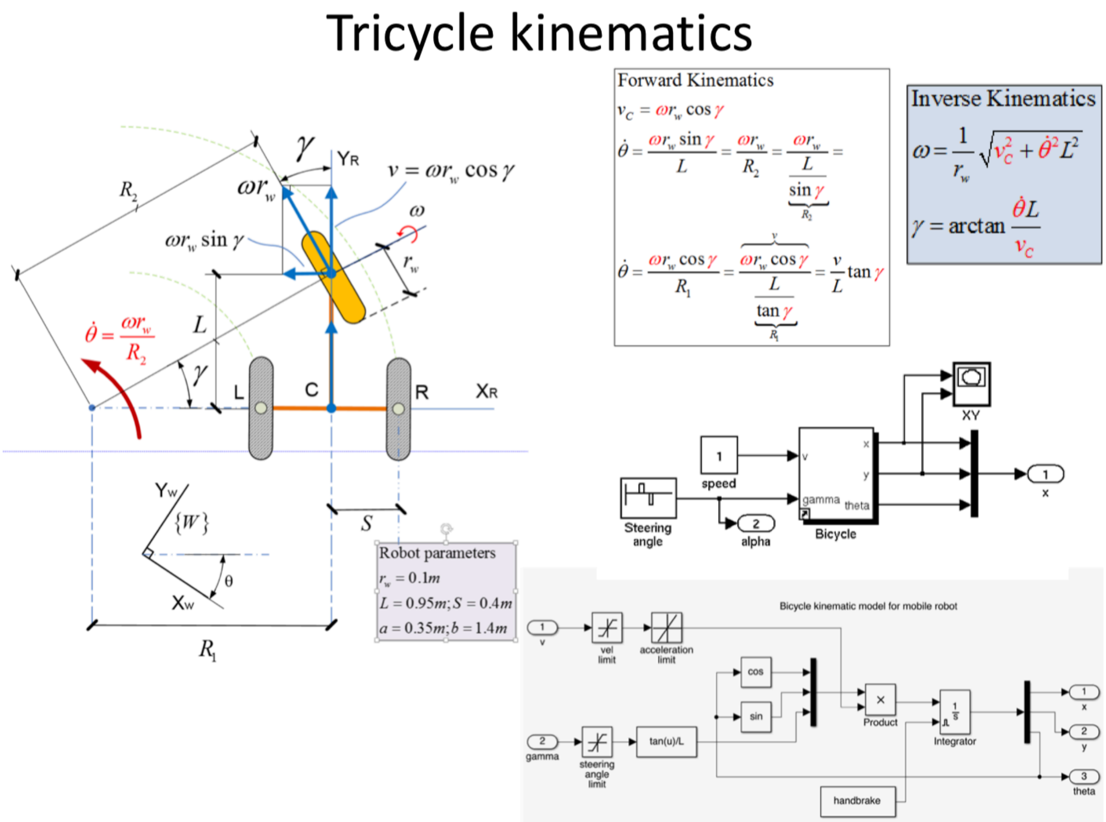
Product of two PDF's

Run the Live Script: ProductPDFs.mlx to check the sensor fusion concept

```
format compact
close all
clear
clc
```

Stimating the tricycle pose

Using the tricycle model. Involved equations



The Robotics ToolBox has some methods

```
V = diag([0.02, 0.5*pi/180].^2) % covariance matrix of noisy odometri
```

```
V = 2×2
10^-3 ×
    0.4000         0
         0    0.0762
```

```
P0=0.0001*eye(3) % Probability Density Function (PDF) of Initial position
```

```
P0 = 3×3
10^-4 ×
    1.0000         0         0
         0    1.0000         0
         0         0    1.0000
```

```
veh = Bicycle('covar', V) % Vehicle constructor
```

```
veh =
Bicycle object
  L=1, steer.max=0.5, accel.max=Inf
  Superclass: Vehicle
    max speed=1, dT=0.1, nhist=0
    V=(0.0004, 7.61544e-05)
    configuration: x=0, y=0, theta=0
```

```
randinit % initialization of randon numbers
vel=1 % velocity
```

```
vel = 1
```

```
gamma=0.3 % steering angle
```

```
gamma = 0.3000
```

You think the odometri is:

$$\delta_d = v_c \Delta t = 1 \cdot \cos(\gamma) \cdot 0.1 \approx 0.1[m]$$

$$\delta_\theta = \dot{\theta} \Delta t = \frac{v}{L} \tan(\gamma) \cdot 0.1 \approx 0.3 \,[\mathrm{rad}]$$

But noise is there:

```
odo = veh.step(vel, gamma)
```

```
odo = 1×2
    0.1108    0.0469
```

Pose without Noise

You think the odometri is:

$$\delta_d = v_c \Delta t = 1 \cdot \cos(\gamma) \cdot 0.1 \approx 0.1[m]$$

$$\delta_\theta = \dot{\theta} \Delta t = \frac{v}{L} \tan(\gamma) \cdot 0.1 \approx 0.3 \,[\mathrm{rad}]$$

Pose: $\xi_k = transl_x(\delta_d) trot_z(\delta_\theta)$

Or

$$\dot{\theta} = \frac{v_c}{L} \tan(\gamma) \rightarrow \theta = \int_0^t \dot{\theta} dt$$

$$\dot{x} = v_c \cos(\theta) \rightarrow x = \int_0^t \dot{x} dt$$

$$\dot{y} = v_c \sin(\theta) \rightarrow y = \int_0^t \dot{y} dt$$

By hand

```
theta_d=vel*tan(gamma)/veh.L
```

```
theta_d = 0.3093
```

```
theta=theta_d*veh.dt
```

```
theta = 0.0309
```

```
next_pose_h=transl(vel*veh.dt,0,0)*trotz(theta)
```

```
next_pose_h = 4×4
    0.9995   -0.0309         0    0.1000
    0.0309    0.9995         0         0
         0         0    1.0000         0
         0         0         0    1.0000
```

```
theta_h=tr2rpy(next_pose_h)
```

```
theta_h = 1×3
         0         0    0.0309
```

Using the RTB (See method of 'veh' object)

```
Pose_t=veh.x'
```

```
Pose_t = 1×3
    0.1000         0    0.0309
```

With noise

$$\text{f}=\xi(k+1)=\begin{pmatrix} x(k)+(\delta_d+\upsilon_d)\cos(\theta(k)+\delta_\theta+\upsilon_\theta) \\ y(k)+(\delta_d+\upsilon_d)\sin(\theta(k)+\delta_\theta+\upsilon_\theta) \\ \theta(k)+\delta_\theta+\upsilon_\theta \end{pmatrix}$$

Using the RTB method (See Bicycle object/method '.f').

```
next_pose=veh.f([0 0 0], odo) % odo comes with noise
```

```
next_pose = 1×3
    0.1106    0.0052    0.0469
```

By hand

```
x_next=0+odo(1)*cos(0+odo(2))
```

```
x_next = 0.1106
```

```
y_next=0+odo(1)*sin(0+odo(2))
```

```
y_next = 0.0052
```

```
theta_next=0+odo(2)
```

```
theta_next = 0.0469
```

```
next_pose=veh.f([1 2 0.1], odo) % odo comes with noise
```

```
next_pose = 1×3
    1.1096    2.0162    0.1469
```

Driving and running the simulation

The vehicle is in a [-10 10 -10 10] enviroment and it move activating the throtel and the steering wheel

```
Ts=0.1
```

```
Ts = 0.1000
```

```
t=(0:Ts:50-Ts)' % aceleration time
```

```
t = 500×1
         0
    0.1000
    0.2000
    0.3000
    0.4000
    0.5000
    0.6000
    0.7000
    0.8000
    0.9000
     ⋮
```

```
vel=1
```

```
vel = 1
```

```
v1= tpoly(0,vel,50)%t(1:50))
```

```
v1 = 50×1
         0
    0.0001
    0.0006
    0.0021
    0.0048
    0.0091
    0.0152
    0.0233
    0.0336
    0.0461
     ⋮
```

```
v2= vel*ones(400,1)
```

```
v2 = 400×1
     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
     ⋮
```

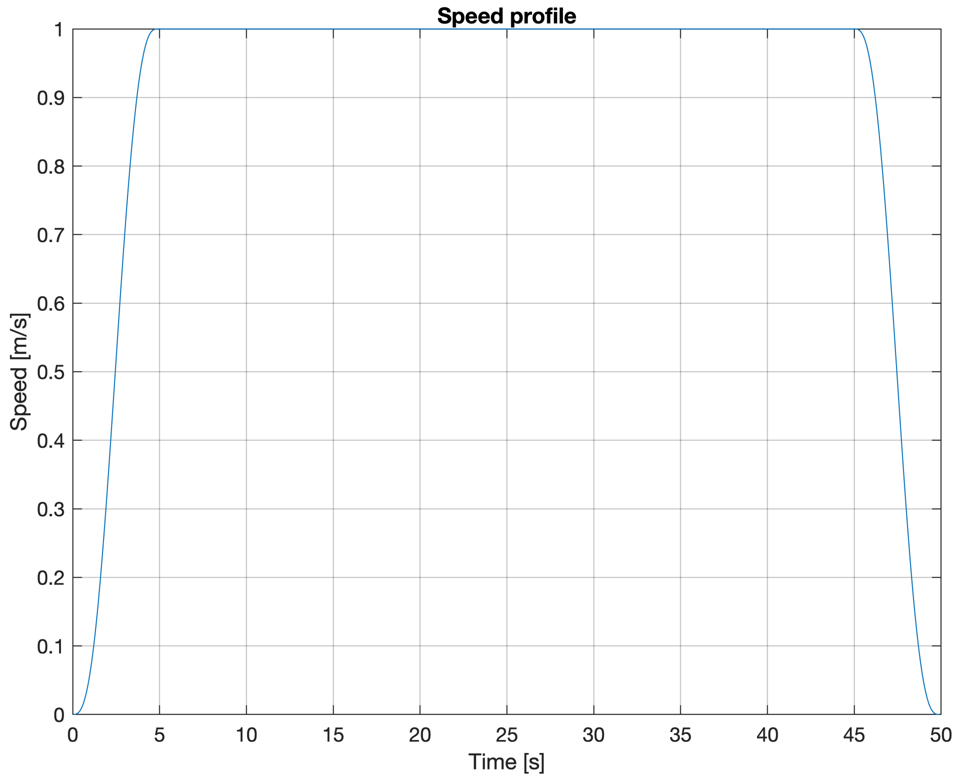
```
v3= tpoly(vel,0,50)%t(451:500))
```

```
v3 = 50×1
    1.0000
    0.9999
    0.9994
    0.9979
    0.9952
    0.9909
    0.9848
    0.9767
    0.9664
    0.9539
      ⋮
```

```
vel=[v1;v2;v3]
```

```
vel = 500×1
         0
    0.0001
    0.0006
    0.0021
    0.0048
    0.0091
    0.0152
    0.0233
    0.0336
    0.0461
      ⋮
```

```
plot(t,vel)
grid on
title('Speed profile')
xlabel('Time [s]')
ylabel ('Speed [m/s]')
```



Running the simulation, which repeatedly calls the step method. The 'veh' object maintains a history of the true state of the vehicle.

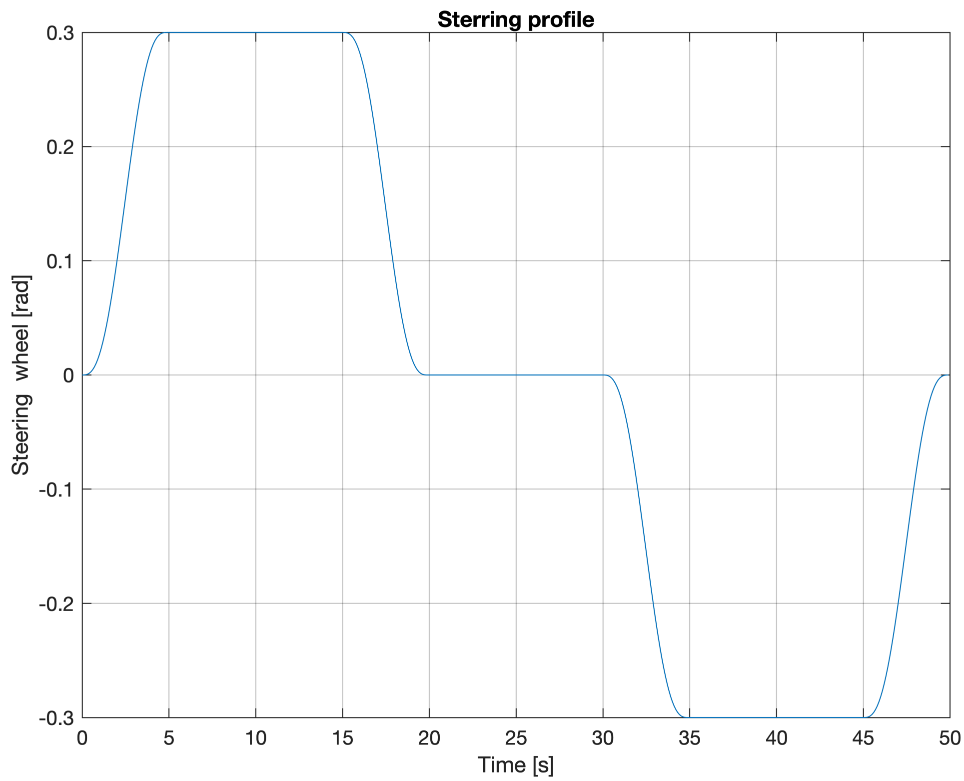
```
gg=0.3
```

```
gg = 0.3000
```

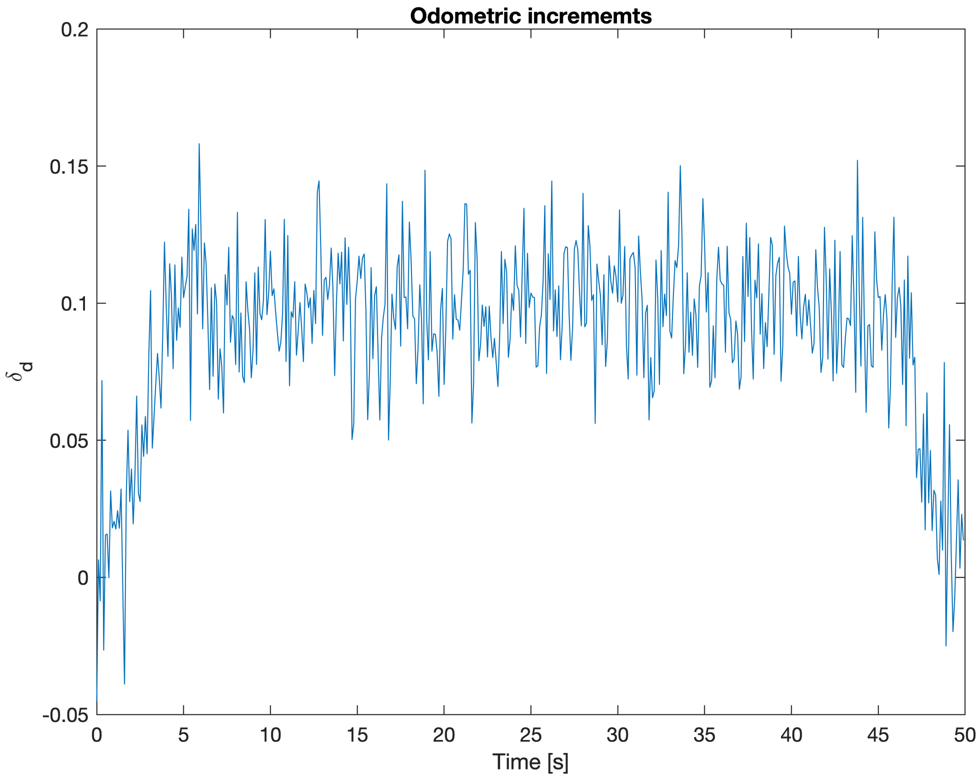
```
g1= tpoly(0,gg,50); % start steering 'cw'
g2= gg*ones(100,1); % been stered
g3= tpoly(gg,0,50);% Going back to no steered
g4= zeros(100,1); % going straight
g5= tpoly(0,-gg,50); % start steering 'ccw'
g6= -gg*ones(100,1); % been stered
g7= tpoly(-gg,0,50);% Going back to no steered
gamma=[g1;g2;g3;g4;g5;g6;g7]
```

```
gamma = 500×1
         0
    0.0000
    0.0002
    0.0006
    0.0014
    0.0027
    0.0045
    0.0070
    0.0101
    0.0138
     ⋮
```

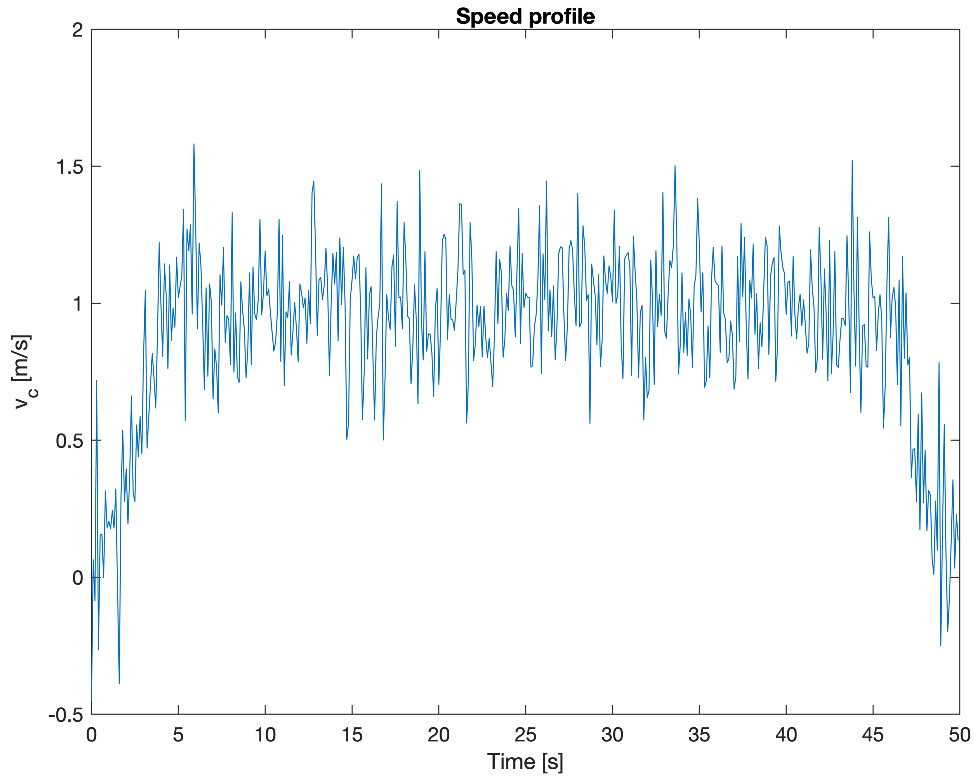
```
plot(t,gamma)
grid on
title('Sterring profile')
xlabel('Time [s]')
ylabel ('Steering  wheel [rad]')
```



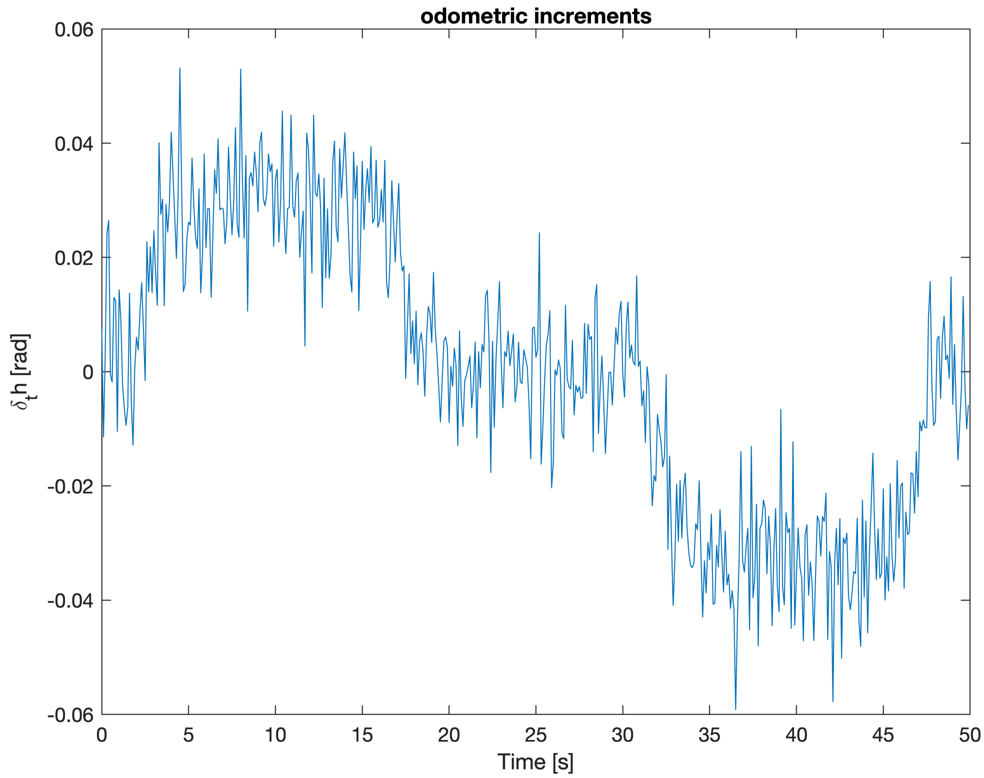

```
for i=1:500
odo(i,:) = veh.step(vel(i), gamma(i));
end
plot(t,odo(:,1));
title('Odometric incrememts')
xlabel('Time [s]')
ylabel('\delta_{d}')
```



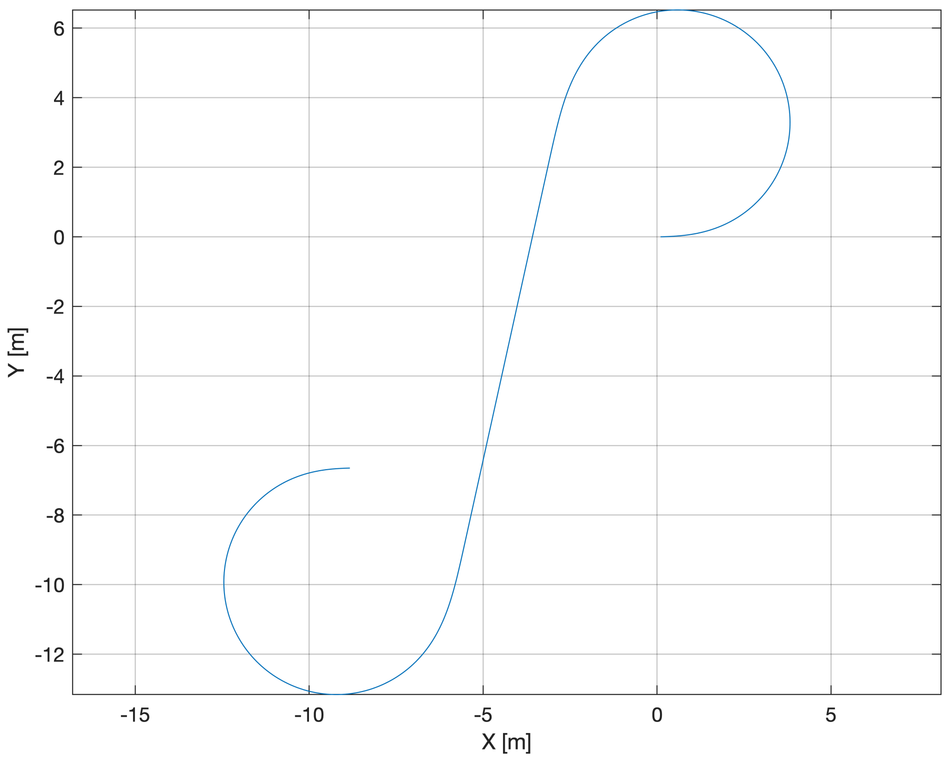

```
plot(t,odo(:,1)/Ts)
title('Speed profile')
xlabel('Time [s]')
ylabel('v_c [m/s]')
```




```
plot(t,odo(:,2));
title('odometric increments')
xlabel('Time [s]')
ylabel('\delta_th [rad]')
```




```
plot(veh.x_hist(:,1), veh.x_hist(:,2))
grid on
axis equal
xlabel('X [m]')
ylabel('Y [m]')
```



Simulating to get the Taylor + Ricatti

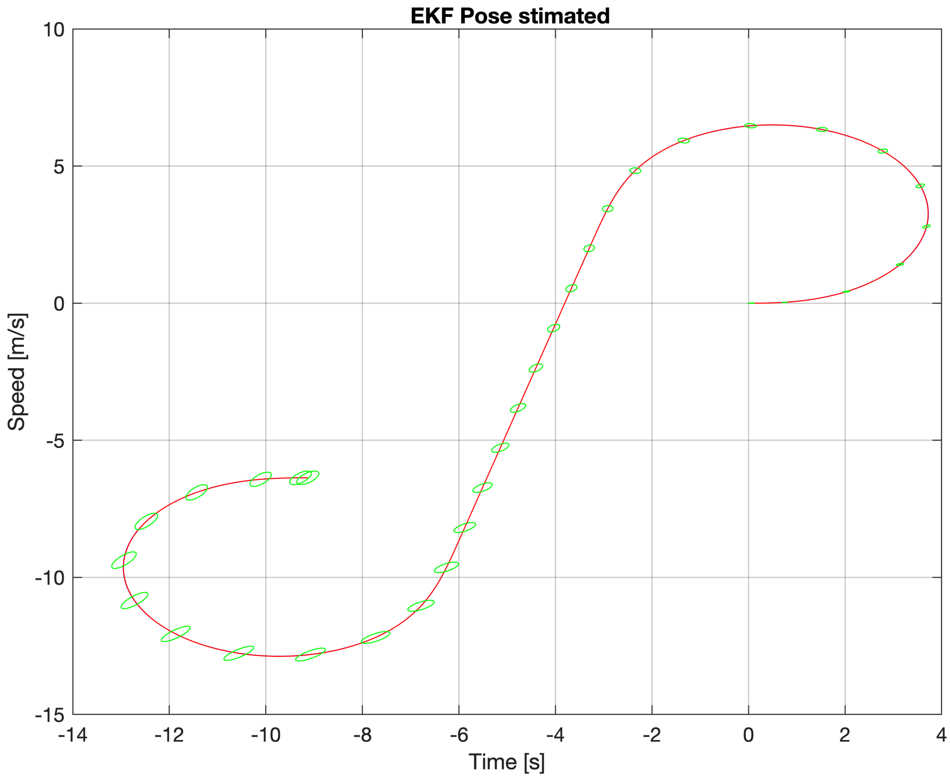
```
sim("Tricycle_EKF_Pose_estimation.slx")
```

```
Ts = 0.1000
```

Plotting the results

```
plot (Pose_t.Data(:,1),Pose_t.Data(:,2),'b');
grid on
title('Speed profile')
xlabel('Time [s]')
ylabel ('Speed [m/s]')
hold on
plot (Pose_est.Data(:,1),Pose_est.Data(:,2),'r');

len=length(Pose_t.Data);
for  i=1:15:len
    plot_ellipse(Pk.signals.values(1:2,1:2,i),[Pose_est.Data(i,1), Pose_est.Data(i,2)],
    title(' EKF Pose stimated');
end
```



Map-based localization

I had a lot of problems to run this Live Script section into Matlab Online.

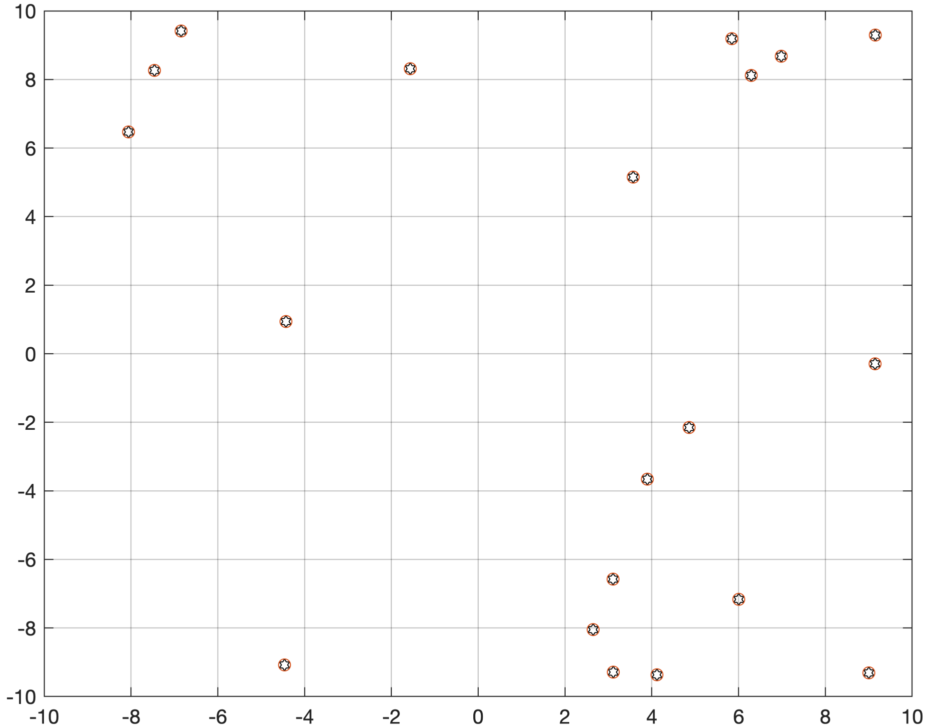
I am not the only one, many other theacher report problems with method 'run' of the EKF object. Check you in the RTB forum of Peter Corke.

After fighting agains all possible RTB's and Matlab 2019b and 2020b I succed with Matlab 2019a with the RTB_10_3.

Build the Map

Known landmarks position randonly distributed

```
randinit % reset random number stream.
map = LandmarkMap(20,10); %N = 20 landmarks uniformly randomly spread over a region spa
map.plot()
scatter(map.map(1,:), map.map(2,:))
```



Sensor data

$z = h(x, p_i)$; $x = (x_v\, y_v\, \theta_v)$ is the vehicle state and $p_i = (x_i\, y_i)$ is is the known location of the ith landmark in the world frame.

$$\boldsymbol{z} = \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{p}_i) = \begin{pmatrix} \sqrt{(y_i - y_v)^2 + (x_i - x_v)^2} \\ \tan^{-1}(y_i - y_v)/(x_i - x_v) - \theta_v \end{pmatrix} + \begin{pmatrix} w_r \\ w_\beta \end{pmatrix}$$

z = (r, β)' r is the distance to the landmark an β is bearing the robot see the landmark

w = (wr, wβ)T is a zero mean Gaussian random variable that models errors in the sensor

```
W = diag([0.1, 1*pi/180].^2); % Noise measurement
```

Sensor on the vehiche

The Robotics ToolBox has the sensor Object with 'Methods'

```
sensor = RangeBearingSensor(veh, map, 'covar', W)
```

```
sensor =
RangeBearingSensor sensor class:
LandmarkMap object
  20 landmarks
  dimension 10.0
W = [0.01 0;0 0.000305]
interval 1 samples
```

Reading the sensor - 'errors ?'

The landmark is chosen randomly from the set of visible landmarks. z = distance and bearing and i the Landmark ID

```
[z,i] = sensor.reading() % return distance and angle
```

```
z = 2×1
    5.1804
   -0.5777
i = 17
```

```
map.landmark(17) % position of Land Mark (n)
```

```
ans = 2×1
   -4.4615
   -9.0766
```

The robot can estimate the range and bearing angle to the landmark based on its own estimated position and the known position of the landmark from the map.

Any difference between the observation z# and the estimated observation indicates an error in the robot's pose estimate ' – it isn't where it thought it was.

$$\nu = z^{\#}\langle k+1\rangle - h\left(\hat{x}^{+}\langle k+1\rangle, p_i\right) \begin{cases} z^{\#}\langle k+1\rangle \rightarrow Sensor\ reading \\ h\left(\hat{x}^{+}\langle k+1\rangle, p_i\right) \rightarrow Robot\ stimate \end{cases}$$

ν is the innovation

Laser innovation & making the fusion

```
randinit
map = LandmarkMap(20,10);
veh = Bicycle('covar', V);
veh.add_driver( RandomPath(map.dim) );
sensor = RangeBearingSensor(veh, map, 'covar', W, 'angle', ...
[-2*pi/3 2*pi/3], 'range', 4, 'animate');
```

Prediction: Taylor and Riccati

$$\hat{x}_{k+1} = \hat{x}_k + F_x(x_k - \hat{x}_k) + F_v \nu_k$$

$$P_{k+1} = F_{xk} P_k F_{xk}^T + F_{vk} V F_{vk}^T$$

Updating

When posssible if land marks are visibles the innovation allows to update

$$\hat{\boldsymbol{x}}^{+}\langle k+1\rangle = \boldsymbol{f}(\hat{\boldsymbol{x}}\langle k\rangle, \hat{\boldsymbol{u}}\langle k\rangle)$$

predict state one step ahead

$$\hat{P}^{+}\langle k+1\rangle = F_x \hat{P}\langle k\rangle F_x^T + F_v V F_v^T$$

project covariance one step ahead

prediction phase

$$\nu = \boldsymbol{z}^{\#}\langle k+1\rangle - h(\hat{\boldsymbol{x}}^{+}\langle k+1\rangle, \boldsymbol{p}_i)$$

new information – innovation

$$K = P^{+}\langle k+1\rangle H_x^T \left(H_x P^{+}\langle k+1\rangle H_x^T + H_w W H_w^T\right)^{-1}$$

how to distribute the innovation to states – Kalman gain

$$\hat{\boldsymbol{x}}\langle k+1\rangle = \hat{\boldsymbol{x}}^{+}\langle k+1\rangle + K\nu$$

state updated with innovation

$$\hat{P}\langle k+1\rangle = \hat{P}^{+}\langle k+1\rangle - K H_x \hat{P}^{+}\langle k+1\rangle$$

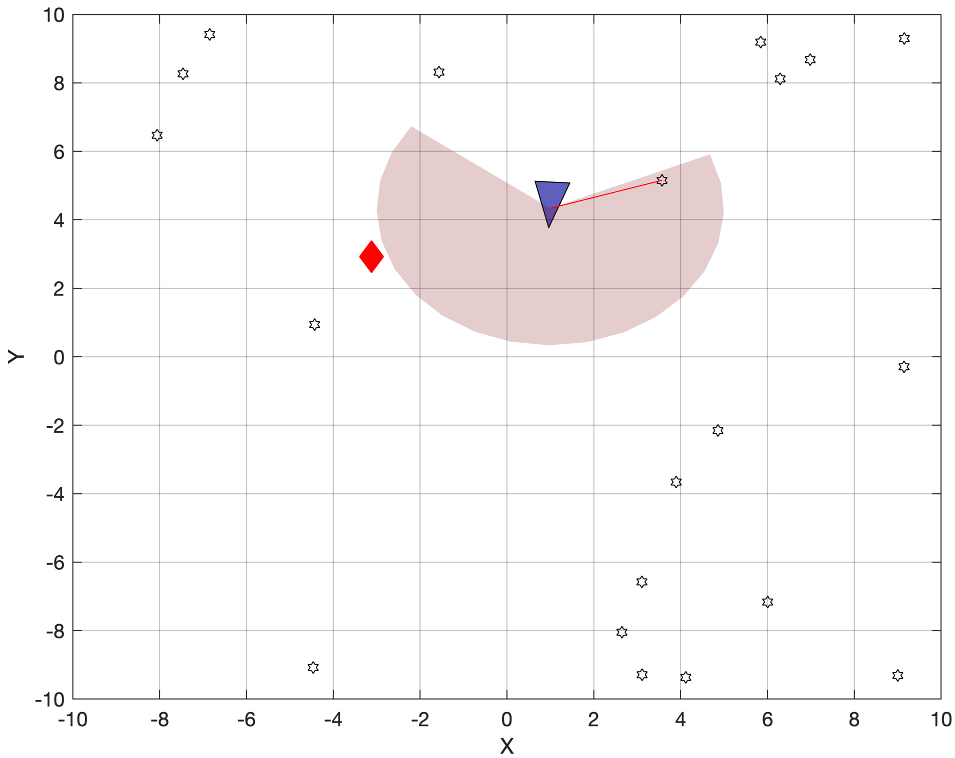
updated covariance

update phase

$$\boldsymbol{H}_x = \left.\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{w}=0} = \begin{pmatrix} -\dfrac{x_i - x_v}{r} & -\dfrac{y_i - y_v}{r} & 0 \\ \dfrac{y_i - y_v}{r^2} & -\dfrac{x_i - x_v}{r^2} & -1 \end{pmatrix}$$

$$\boldsymbol{H}_w = \left.\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{w}}\right|_{\boldsymbol{w}=0} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
ekf=EKF(veh, V, P0, sensor, W, map); % constructor of the EKF object
ekf.run(1000); % run the simulation Predicting and updating.
```



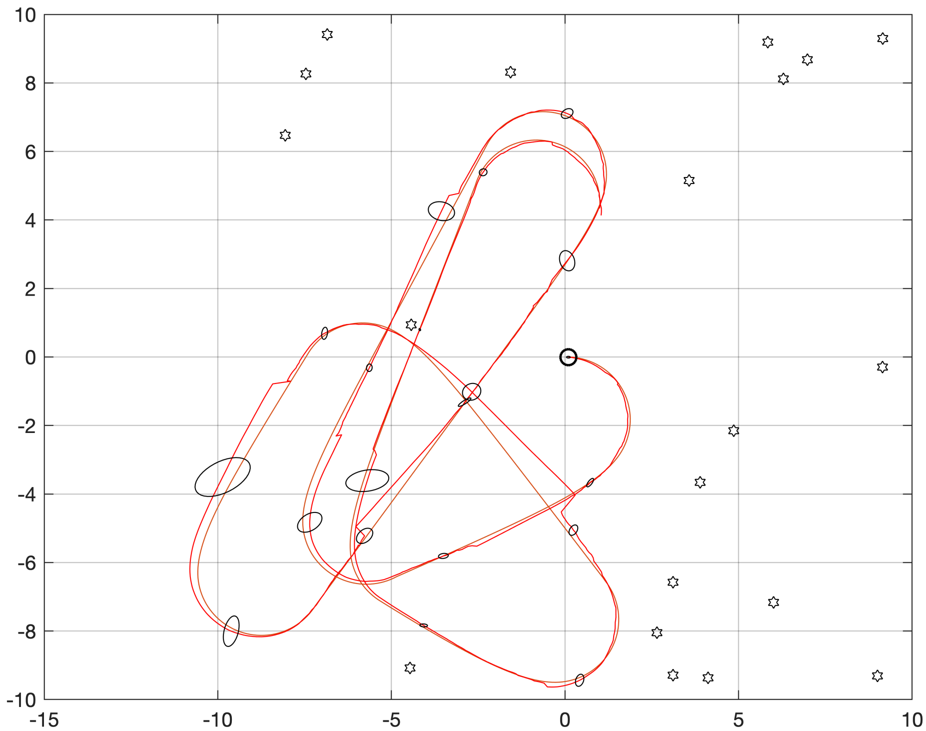
In case your enviroment fails:

See video:https://drive.google.com/file/d/1PIg86QQqi5GCic19OeUx_2UzZMGcMqZs/view?usp=sharing

Load the Workspace (EKF_Section.mat) to plot results. Check the pdf 'Senssor_fusion and _Localization.pdf'

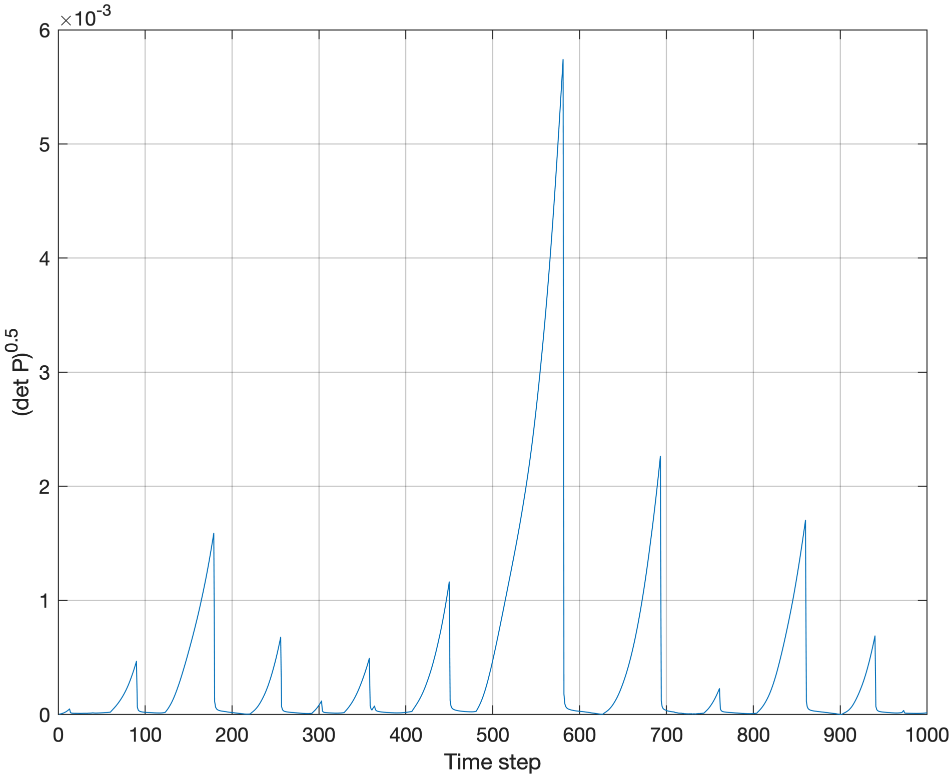
Plotting trajectories and ellipse errors

```
map.plot() % plot the map with the land marks
veh.plot_xy(); % Theoric trajectory
ekf.plot_xy('r'); % updated trajectory
ekf.plot_ellipse('k') % viasualizing the 2D ellipses
```



Plotting statistics

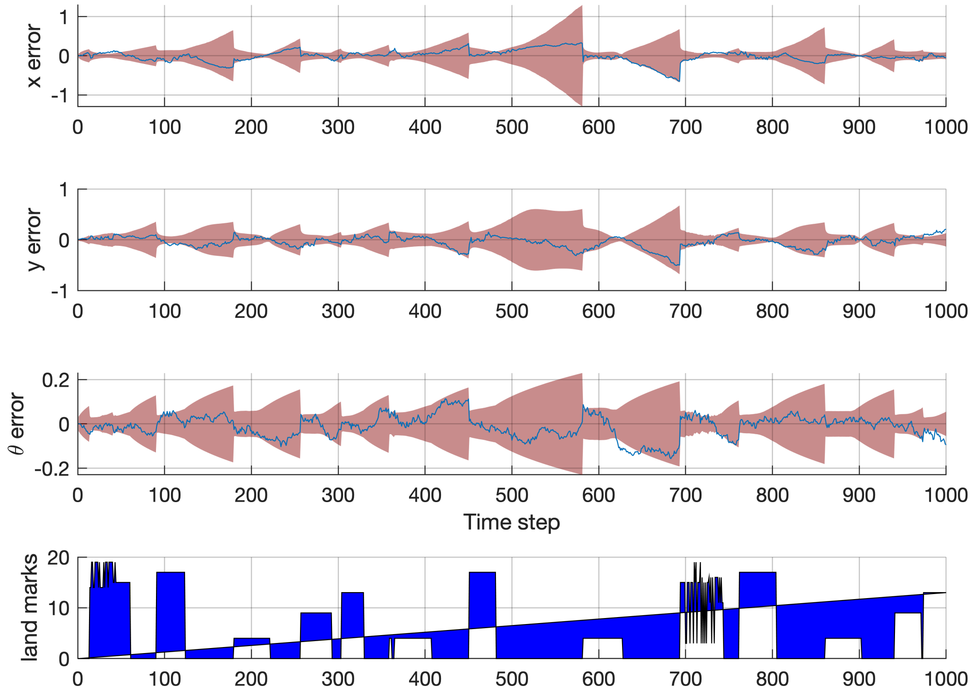
```
clf
ekf.plot_P() % compact way to see the coovariance
xlabel('Time step'); ylabel('(det P)^{0.5}'); grid
```



Covariance magnitude as a function of time. Overall uncertainty is given by $\sqrt{\det(P_k)}$ and shows that uncertainty does not

continually increase with time.

```
clf
ekf.plot_error('confidence', 0.95, 'nplots', 4) % errors x,y,theta
plot_poly([1:1000; sensor.landmarklog], 'fill', 'b')% which landmark
ylabel('land marks')
grid
```



Top: pose estimation error with 95% confi dence bound shown in pink; bottom: observed landmarks the bar indicates which landmark is seen at each time step, 0 means no observation.