COLUMBIA UNIVERSITY EEME E6911 FALL ‘25

TOPICS IN CONTROL : PROBABILISTIC ROBOTICS

KALMAN FILTER

Instructor: Ilija Hadzic

Recall (Bayes Filter – Previous Class)

- Prediction:

$$\overline{bel}\,(x_i[n]) = \sum_k P\big(x[n] = x_i \big| x[n-1] = x_k, u[n] = u_j\big) bel(x_k[n-1])$$

- Innovation:

$$bel(x_i[n]) = \eta P\big(z[n] = z_j \big| x[n] = x_i\big)\overline{bel}\,(x_i[n])$$

$$\eta \sum_i P\big(z = z_j \big| x = x_i\big)\overline{bel}\,(x_i) = \sum_i bel(x_i[n]) = 1$$

Assumptions and Objectives

- System model is available (linear or linearized).
- Signal values are continuous.
- Gaussian uncertainty.
- No bias.

- Objective: Extract maximum information from system input signals and measurements to estimate the state.

Gaussian Distribution

Covariance

$$\boldsymbol{x}: \mathcal{N}(\boldsymbol{x}, \boldsymbol{\Sigma})$$

$$f(\boldsymbol{x}) = \frac{1}{(2\pi)^{\frac{k}{2}}\sqrt{|\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)$$

Mean

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_{11}^2 & \cdots & \rho_{1k}\sigma_{11}\sigma_{kk} \\ \vdots & \ddots & \vdots \\ \rho_{1k}\sigma_{11}\sigma_{kk} & \cdots & \sigma_{kk}^2 \end{bmatrix}$$

Correlation coefficient

Individual variance

Bayesian Inference

- What are we looking for?
- What are we measuring?
- How do we represent PDFs?
- What do we know?

Bayesian Inference

- What are we looking for: $f(\boldsymbol{x} \mid \boldsymbol{u}, \boldsymbol{z})$
- What are we measuring: $f(\boldsymbol{z}\ ), f(\boldsymbol{u})$
- How do we represent PDFs: mean and covariance
- What do we know: system model
  - How the input reflects on state: $f(\boldsymbol{x}^{+} \mid \boldsymbol{x}, \boldsymbol{u})$
  - What we expect to measure given the state: $f(\boldsymbol{z} \mid \boldsymbol{x})$

System Model

$$\boldsymbol{x}[n] = \mathbf{A}\boldsymbol{x}[n-1] + \mathbf{B}\boldsymbol{u}[n]$$

State transition model: given input u, how the state changes

$$\mathbf{z}[n] = \boldsymbol{C}\boldsymbol{x}[n]$$

Measurement model: given state x, what should the sensor measure

- $\boldsymbol{u}[n]$ is the system input, but can be the measurement from another sensor (e.g. odometry).
- $\boldsymbol{z}[n]$ is the system output, but can be any observable signal.

What is Kalman Filter Intuitively?

- Fancy weighted mean.
- Weights proportional to confidence.
- Weights dynamically tracked.
- Assume we know uncertain state: $\boldsymbol{x}, \boldsymbol{\Sigma}_x$.
- We measure (or know) uncertain input: $\boldsymbol{u}, \boldsymbol{\Sigma}_u$.
- We measure uncertain sensor: $\boldsymbol{z}, \boldsymbol{\Sigma}_z$.

Prediction

$$\overline{\boldsymbol{x}}[n] = \mathbf{A}\boldsymbol{x}[n-1] + \mathbf{B}\boldsymbol{u}[n]$$

Update the state using the model

$$\overline{\boldsymbol{\Sigma}}_x[n] = \mathbf{A}\boldsymbol{\Sigma}_x[n-1]\mathbf{A}^{\mathrm{T}} + \mathbf{B}\boldsymbol{\Sigma}_u[n]\mathbf{B}^{\mathrm{T}}$$

Propagate the covariance through the model.

- Model predicts what should happen given the input.
- We are *less* certain about the predicted state.
- If the model is non-linear, we can use Jacobians for covariance propagation.
- Side-note: in the textbook $\boldsymbol{R} = \boldsymbol{B}\boldsymbol{\Sigma}_u[n]\boldsymbol{B}^T$

Kalman Gain

No pre-multiply with C to stay in state space

$$\mathbf{K} = \bar{\mathbf{\Sigma}}_x[n]\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\bar{\mathbf{\Sigma}}_x[n]\mathbf{C}^{\mathrm{T}} + \mathbf{\Sigma}_z[n]\right)^{-1}$$

Prediction uncertainty transformed to measurement space

Measurement uncertainty

- This will be our “fancy weight factor”
- Notice that it is just a vectorized “cousin” of $\frac{a}{a+b}$

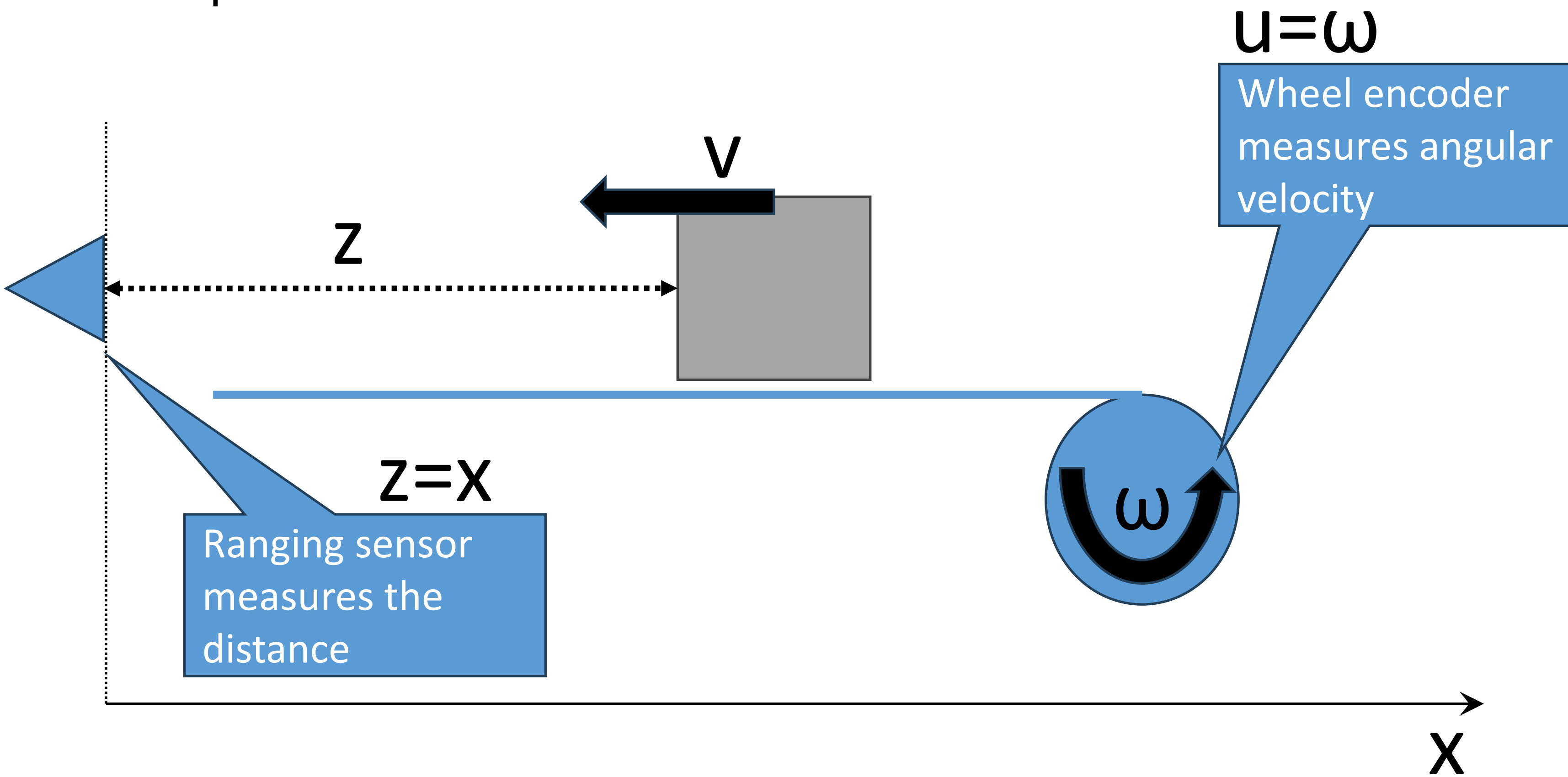
Innovation

$$\boldsymbol{x}[n] = \overline{\boldsymbol{x}}[n] + \mathbf{K}(\boldsymbol{z}[n] - \mathbf{C}\overline{\boldsymbol{x}}[\mathrm{n}])$$

$$\boldsymbol{\Sigma}_x[n] = (\mathbf{I} - \mathbf{KC})\overline{\boldsymbol{\Sigma}}_x[n]$$

- The state moves towards the measurement.
- But only proportionally to the relative confidence.
- This step *reduces* uncertainty.

Example: Two 1D Sensors



- Objective: estimate box position on conveyor belt

Example: Two 1D Sensors, Model

$$x[n] = x[n-1] - r\Delta t \cdot u[n]$$

$$\mathrm{z}[n] = x[n]$$

$$A = 1$$

$$B = -r\Delta t$$

$$C = 1$$

Example: Two 1D Sensors, Prediction

$$\bar{x}[n] = x[n-1] - r\Delta t \cdot u[n]$$

$$\bar{\sigma}_x^2\,[n] = \sigma_x^2[n-1] + r^2\Delta t^2\sigma_u^2[n]$$

Example: Two 1D Sensors, Gain

$$K = \frac{\bar{\sigma}^2{}_x[n]}{\bar{\sigma}_x^2[n] + \sigma_z^2[n]}$$

$$1 - K = \frac{\sigma_z^2[n]}{\bar{\sigma}_x^2[n] + \sigma_z^2[n]}$$

Example: Two 1D Sensors, Innovation

$$x[n] = (1 - K)\bar{x}[n] + Kz[n]$$

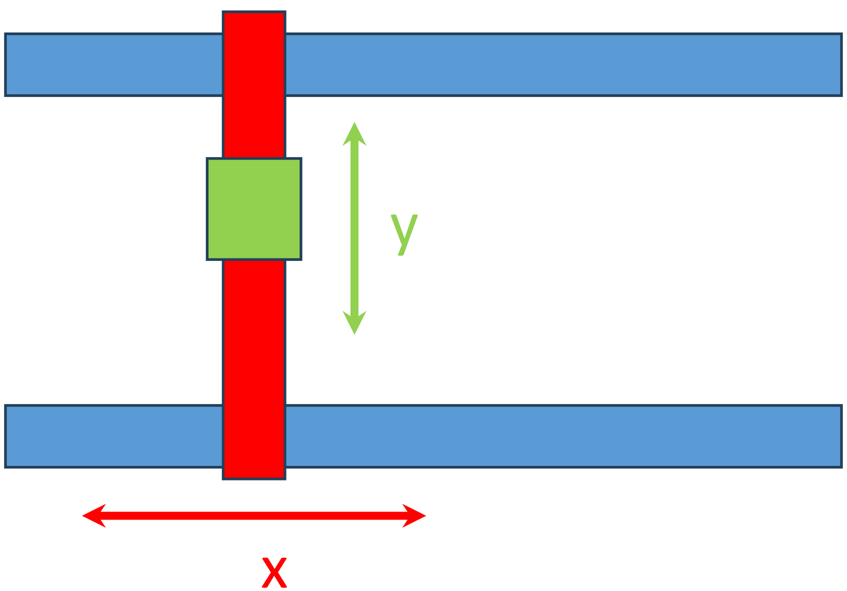
$$\sigma_x^2[n] = \bar{\sigma}_x^2[n](1 - K) = \frac{\bar{\sigma}_x^2[n]\sigma_z^2[n]}{\bar{\sigma}_x^2[n] + \sigma_z^2[n]}$$

Summary

- Extracts maximum information from sensors.
- Requires the knowledge of system model.
- Requires uncertainty estimate.
- Optimal under no-bias/Gaussian assumption.
- Assumes linear (or linearized) system.
- The resulting uncertainty is smaller than the smallest individual uncertainty!

Example: 2D Cartesian Robot

- Motor gear drives linear slider
- Camera mounted on end-effector overlooks the scene from above
- Apriltag markers on the floor seen by the camera



Motor Dynamics

- Torque constant: K
- Rotor inertia: J
- Friction constant: b
- Armature resistance: R
- Armature inductance: L
- Assume no load torque

$$\begin{bmatrix} \frac{d\omega}{dt} \\ \frac{di}{dt} \end{bmatrix} = \begin{bmatrix} -\frac{b}{J} & \frac{K}{J} \\ -\frac{K}{L} & -\frac{R}{L} \end{bmatrix} \begin{bmatrix} \omega \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} v$$

Motor Dynamics – Discrete time

$$\begin{bmatrix} \omega[n] \\ i[n] \end{bmatrix} = \begin{bmatrix} 1 - \Delta t \frac{b}{J} & \Delta t \frac{K}{J} \\ -\Delta t \frac{K}{L} & 1 - \Delta t \frac{R}{L} \end{bmatrix} \begin{bmatrix} \omega[n-1] \\ i[n-1] \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{\Delta t}{L} \end{bmatrix} v[n]$$

Guiderail Kinematics

- Wheel radius: r

$$p[n] = p[n-1] + r\Delta t\omega[n]$$

Full State-Space Model

- Six state variables (three per guide rail).
- System is linear.
- Axes are decoupled.

$$\begin{bmatrix} p_x[n] \\ \omega_x[n] \\ i_x[n] \\ p_y[n] \\ \omega_y[n] \\ i_y[n] \end{bmatrix} = \begin{bmatrix} 1 & r\Delta t & 0 & 0 & 0 & 0 \\ 0 & 1-\Delta t\dfrac{b}{J} & \Delta t\dfrac{K}{J} & 0 & 0 & 0 \\ 0 & -\Delta t\dfrac{K}{L} & 1-\Delta t\dfrac{R}{L} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & r\Delta t & 0 \\ 0 & 0 & 0 & 0 & 1-\Delta t\dfrac{b}{J} & \Delta t\dfrac{K}{J} \\ 0 & 0 & 0 & 0 & -\Delta t\dfrac{K}{L} & 1-\Delta t\dfrac{R}{L} \end{bmatrix} \begin{bmatrix} p_x[n-1] \\ \omega_x[n-1] \\ i_x[n-1] \\ p_y[n-1] \\ \omega_y[n-1] \\ i_y[n-1] \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \dfrac{\Delta t}{L} & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & \dfrac{\Delta t}{L} \end{bmatrix} \begin{bmatrix} v_x[n] \\ v_y[n] \end{bmatrix}$$

Measurement Model

- Position measured directly from image processing
- There could be cross-terms in the covariance

$$\begin{bmatrix} z_x[n] \\ z_y[n] \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_x[n] \\ \omega_x[n] \\ i_x[n] \\ p_y[n] \\ \omega_y[n] \\ i_y[n] \end{bmatrix}$$

Implementation – constructor

```
import numpy
import math

default_var = 0.25
default_time_step = 1 / 50

class CartesianBotKF:
    def __init__(self, torque_const, rotor_inertia, friction_const,
                 armature_res, armature_ind, wheel_radius):
        self.K = torque_const
        self.J = rotor_inertia
        self.b = friction_const
        self.R = armature_res
        self.L = armature_ind
        self.r = wheel_radius
        self.state = numpy.zeros((6, 1))
        self.state_cov = numpy.identity(6) * default_var
        self.predicted_state = numpy.zeros((6, 1))
        self.predicted_state_cov = numpy.identity(6) * default_var
        self.time = None
```

Implementation – initialization

```
def set_state(self, timestamp,
              position_x = 0, position_y = 0,
              wheel_ang_vel_x = 0, wheel_ang_vel_y = 0,
              motor_current_x = 0, motor_current_y = 0):
    self.time = timestamp
    self.state[0, 0] = position_x
    self.state[1, 0] = wheel_ang_vel_x
    self.state[2, 0] = motor_current_x
    self.state[3, 0] = position_y
    self.state[4, 0] = wheel_ang_vel_y
    self.state[5, 0] = motor_current_y
    self.state_cov = numpy.identity(6) * default_var
```

Implementation – model

```
def A_matrix(self, delta_t = default_time_step):
    A = numpy.identity(6)
    A[0, 1] = A[3, 4] = self.r * delta_t
    A[1, 1] = A[4, 4] = 1 - delta_t * self.b / self.J
    A[1, 2] = A[4, 5] = delta_t * self.K / self.J
    A[2, 1] = A[5, 4] = -delta_t * self.K / self.L
    A[2, 2] = A[5, 5] = 1 - delta_t * self.R / self.L
    return A

def B_matrix(self, delta_t = default_time_step):
    B = numpy.zeros((6, 2))
    B[2, 0] = B[5, 1] = delta_t / self.L
    return B

def C_matrix(self):
    C = numpy.zeros((2,6))
    C[0, 0] = 1
    C[1, 3] = 1
    return C
```

Implementation – prediction

```
def _predict(self, timestamp, v_x, v_y,
             var_vx = default_var, var_vy = default_var):
    if self.time is None:
        raise RuntimeError('uninitialized filter')
    delta_t = timestamp - self.time
    self.time = timestamp
    u = numpy.array([[ v_x ],
                     [ v_y ]])
    sigma_u = numpy.array([[var_vx, 0],
                           [0, var_vy]])
    self.predicted_state = self.A_matrix(delta_t) @ \
        self.state + self.B_matrix(delta_t) @ u
    self.predicted_state_cov = \
        self.A_matrix(delta_t) @ self.state_cov @ \
        self.A_matrix(delta_t).transpose() + \
        self.B_matrix(delta_t) @ sigma_u @ \
        self.B_matrix(delta_t).transpose()
```

Implementation – measurement

```
def _measure(self, z_x, z_y,
             var_zx = default_var, var_zy = default_var,
             rho_zxy = 0):
    sigma_z = numpy.array([
        [var_zx, rho_zxy * math.sqrt(var_zx) * math.sqrt(var_zy)],
        [rho_zxy * math.sqrt(var_zx) * math.sqrt(var_zy), var_zy]])
    z = numpy.array([[z_x],
                     [z_y]])
    SigmaXCT =  self.predicted_state_cov @ self.C_matrix().transpose()
    CSigmaXCT = self.C_matrix() @ SigmaXCT
    K = SigmaXCT @ numpy.linalg.inv(CSigmaXCT + sigma_z)
    self.state = self.predicted_state + K @ \
        (z - self.C_matrix() @ self.predicted_state)
    self.state_cov = (numpy.identity(6) - K @ self.C_matrix()) @ \
        self.predicted_state_cov
```

Implementation – top level

```
def advance_filter(self, timestamp, v_x, v_y, z_x, z_y,
                   var_vx = default_var, var_vy = default_var,
                   var_zx = default_var, var_zy = default_var,
                   rho_zxy = 0):
    self._predict(timestamp, v_x, v_y, var_vx, var_vy)
    self._measure(z_x, z_y, var_zx, var_zy, rho_zxy)

def get_estimate(self):
    z = self.C_matrix() @ self.state
    z_cov = self.C_matrix() @ self.state_cov @ self.C_matrix().transpose()
    return z, z_cov

def get_state(self):
    return self.state, self.state_cov
```

Implementation – utilities

```
def _skip_measure(self):
    self.state = self.predicted_state
    self.state_cov = numpy.identity(6) * default_var

def peek_pos(self):
    return self.state[0, 0], self.state[3, 0]

def peek_omega(self):
    return self.state[1, 0], self.state[4, 0]

def peek_current(self):
    return self.state[2, 0], self.state[5, 0]

def simulate_system(self, timestamp, v_x, v_y):
        self._predict(timestamp, v_x, v_y)
        self._skip_measure()
```

Example Motor Parameters

$K = 0.01\ N \cdot m/A$

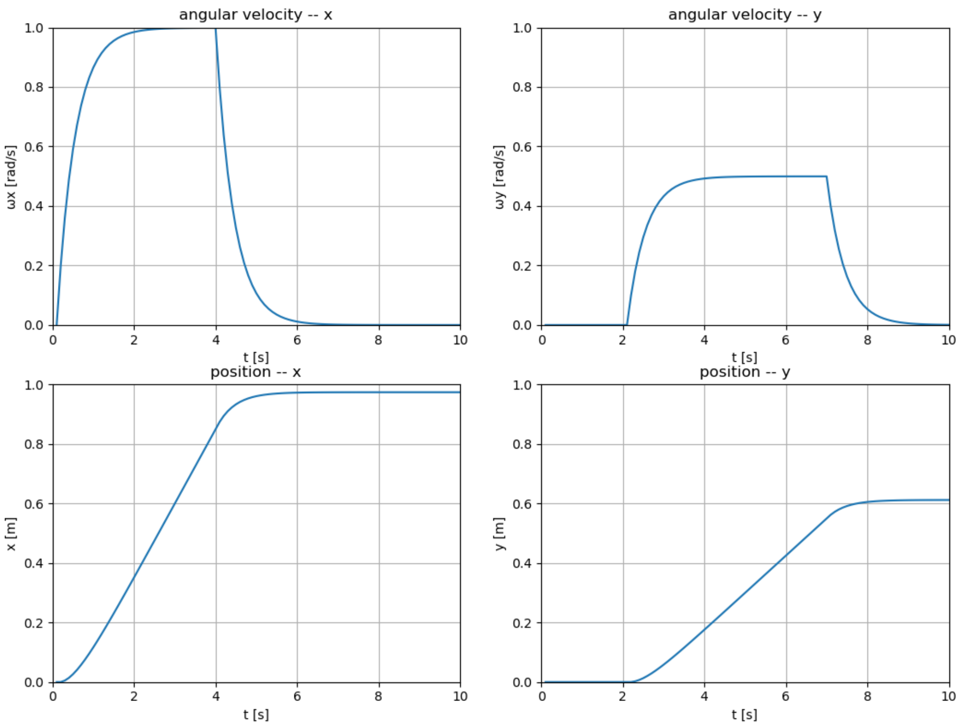
$J = 0.01\ kg \cdot m^2$

$b = 0.1\ N \cdot m \cdot s$

$R = 1\ \Omega$

$L = 0.5\ H$

Example Motion



- x-motor input held at 10 V from 0s to 5s
- y-motor input held at 5V from 2s to 7s

Usage – simulation

```
# motor parameters
K = 0.01
J = 0.01
b = 0.1
R = 1
L = 0.5

# wheel radius
r = 0.25

# time stemp
t_step = 0.1

bot_model = CartesianBotKF(K, J, b, R, L, r)
bot_model.set_state(0)

estimator = CartesianBotKF(K, J, b, R, L, r)
estimator.set_state(0)
vx_variance = 0.1
vy_variance = 0.2
t = t_step
all_t = [i * t_step for i in range(1, 101)]
```

Example Estimation

- System input voltages are noisy, so system output is noisy.
- Estimator does not know exact inputs, so it assumes nominal voltages during prediction.
- Estimator knows input noise properties (variance).
- Input noise process is time-invariant statistics.
- Estimator measures x, y (output) with error.
- Estimator does not know the instantaneous error, but it knows the measurement error properties (2D covariance matrix).
- Measurement covariance changes with each measurement (image processing system estimates it with each measurement).
- Initial state is known and is all-zero.

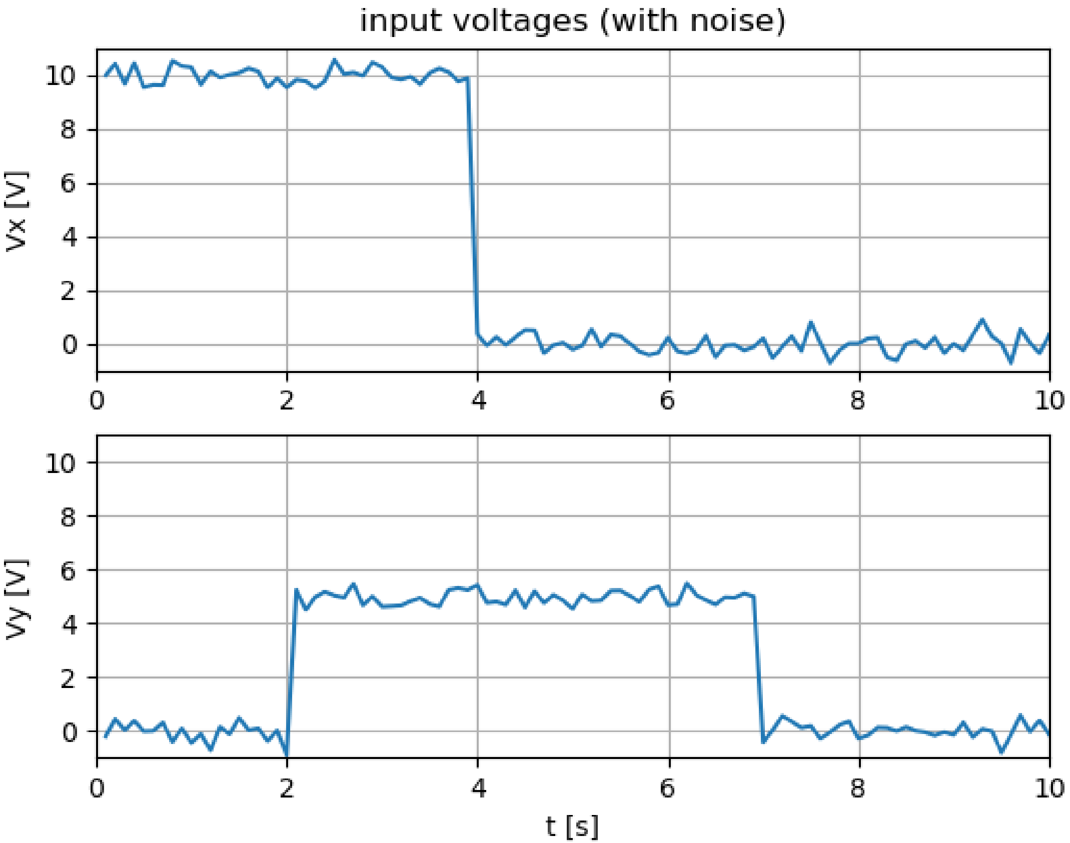
Simulation

```
# input x voltage with noise
vx_nominal = [10 if t < 4 else 0 for t in all_t]
vx = [v+n for v, n in zip(vx_nominal, list(numpy.random.normal(0,
math.sqrt(vx_variance), len(all_t))))]

# input y voltage with noise
vy_nominal = [5 if t < 7 and t > 2 else 0 for t in all_t]
vy = [v+n for v, n in zip(vy_nominal, list(numpy.random.normal(0,
math.sqrt(vx_variance), len(all_t))))]

# uncertainty range and correlation for measurements
min_var_z = 0.01
max_var_z = 0.2
min_rho_z = 0
max_rho_z = 0.7

x_ground_truth = []
y_ground_truth = []
x_estimate = []
y_estimate = []
```

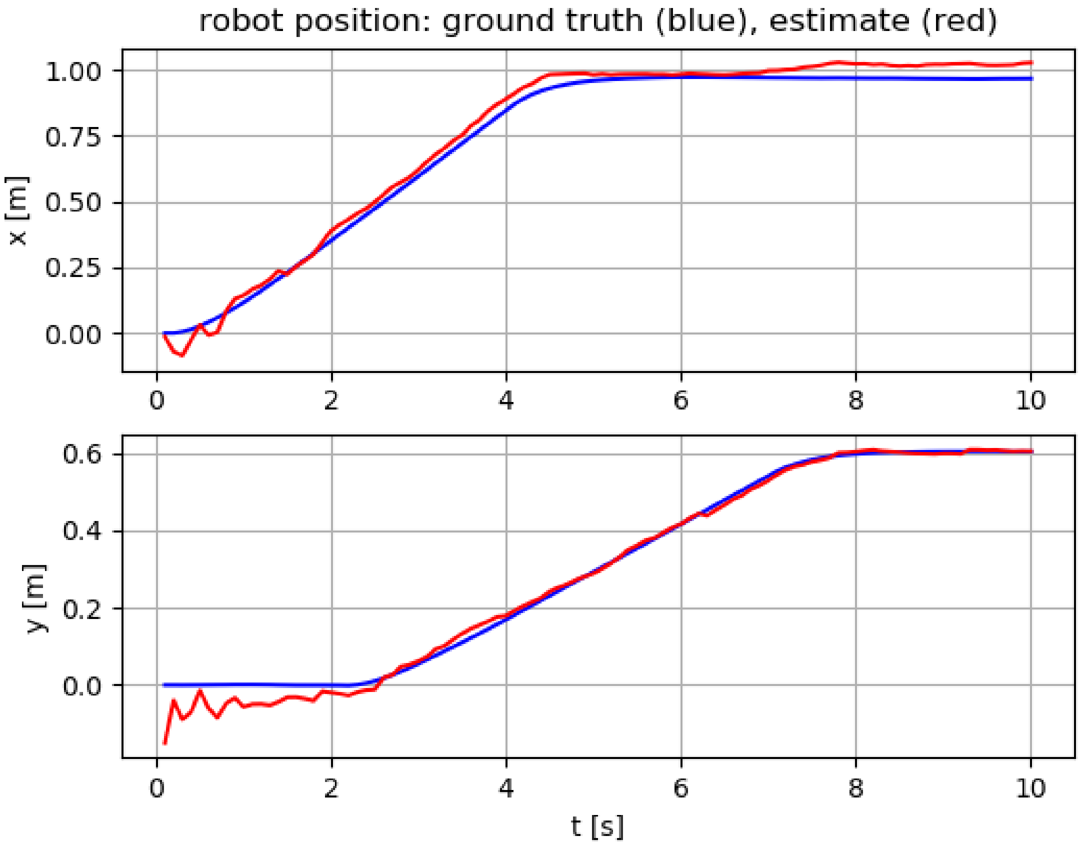
Simulation

```
def generate_measurement(xm, ym):
    var_x = random.uniform(min_var_z, max_var_z)
    var_y = random.uniform(min_var_z, max_var_z)
    rho = random.uniform(min_rho_z, max_rho_z)
    var_xy = rho * math.sqrt(var_x) * math.sqrt(var_y)
    cov = numpy.array([[var_x, var_xy],
                       [var_xy, var_y]])
    xz, yz = numpy.random.multivariate_normal([xm, ym], cov)
    return xz, yz, var_x, var_y, rho

for i in range(len(all_t)):
    # move the system under simulation using noisy input
    bot_model.simulate_system(all_t[i], vx[i], vy[i])
    xgt, ygt = bot_model.peek_pos()
    x_ground_truth.append(xgt)
    y_ground_truth.append(ygt)
    zx, zy, var_zx, var_zy, rho_zxy = generate_measurement(xgt, ygt)
    estimator.advance_filter(all_t[i], vx[i], vy[i], zx, zy,
                             vx_variance, vy_variance,
                             var_zx, var_zy, rho_zxy)
    est_xy, est_xy_cov = estimator.get_estimate()
    est_x, est_y = est_xy[0, 0], est_xy[1, 0]
    x_estimate.append(est_x)
    y_estimate.append(est_y)
```

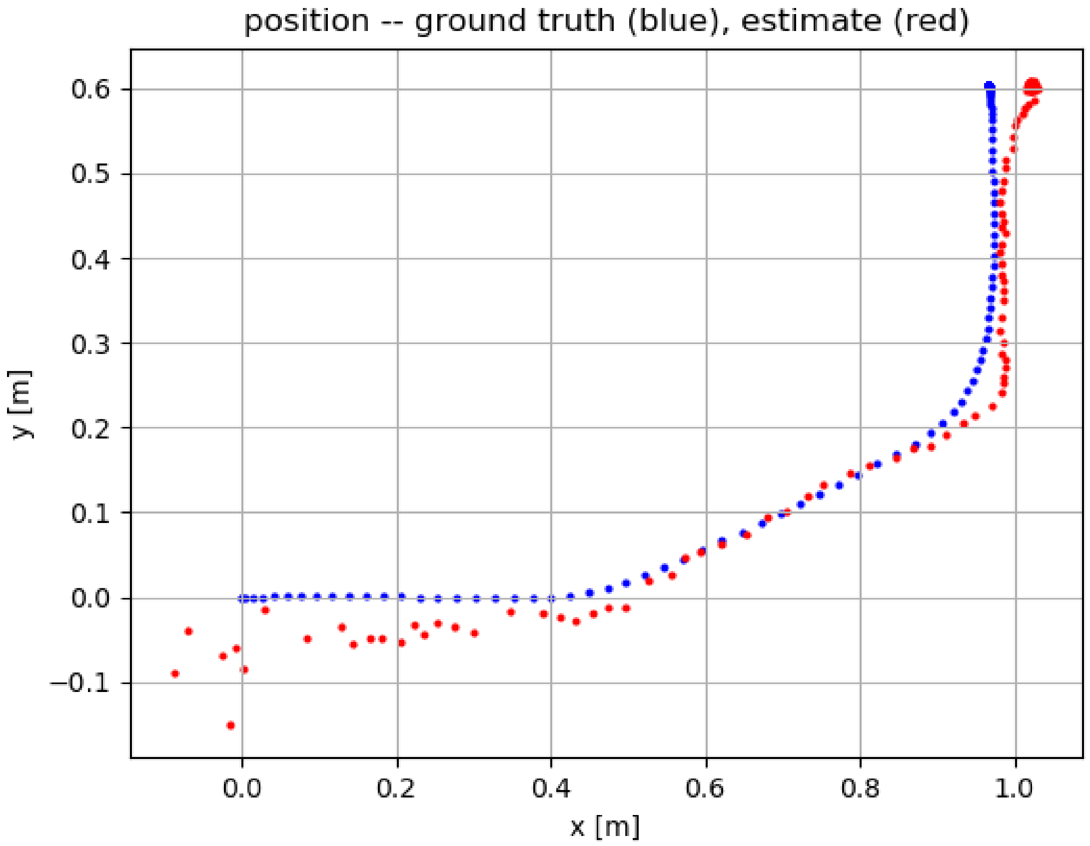
Results



Simulation Results



Results



position -- ground truth (blue), estimate (red)
y [m]
x [m]
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
0.0
0.2
0.4
0.6
0.8
1.0