An Intuitive Tutorial on Bayesian Filtering

Sunglok Choi, Assistant Professor, Ph.D.
Computer Science and Engineering Department, SeoulTech
sunglok@seoultech.ac.kr | https://mint-lab.github.io/

Getting Started from Average

- **Example) Merging two weight measurements**
  - Given: Two measurements from two weight scales, 72 kg and 74 kg
  - Target: The true weight
  - Solution: Average

$$\frac{1}{2}(72 + 74) = 73$$

Image: [Wikipedia]

Getting Started from Weighted Average

- **Example) Merging two weight measurements with their variance**
  - Given: Two measurements from two weight scales
    - $x_1 = 72$ kg from a weight scale whose variance $\sigma_1^2 = 1$
    - $x_2 = 74$ kg from a weight scale whose variance $\sigma_2^2 = 4$
    - Note) Two scales were zero-adjusted so that had no bias error.
  - Target: The true weight $\bar{x}$
  - Solution: **Inverse-variance weighted average**

$$\bar{x} = \frac{\sum_i x_i / \sigma_i^2}{\sum_i 1/\sigma_i^2} = \left(\frac{x_1}{\sigma_1^2} + \frac{x_2}{\sigma_2^2} \right) \Big/ \left(\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2} \right) = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2} x_1 + \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2} x_2 = \frac{4}{5} \cdot 72 + \frac{1}{5} \cdot 74 = 72.4$$

$$\bar{\sigma} = \frac{1}{\sum_i 1/\sigma_i^2} = \frac{\sigma_1^2 \, \sigma_2^2}{\sigma_1^2 + \sigma_2^2} = \frac{4}{5} = 0.8$$

Image: Wikipedia

Table of Contents

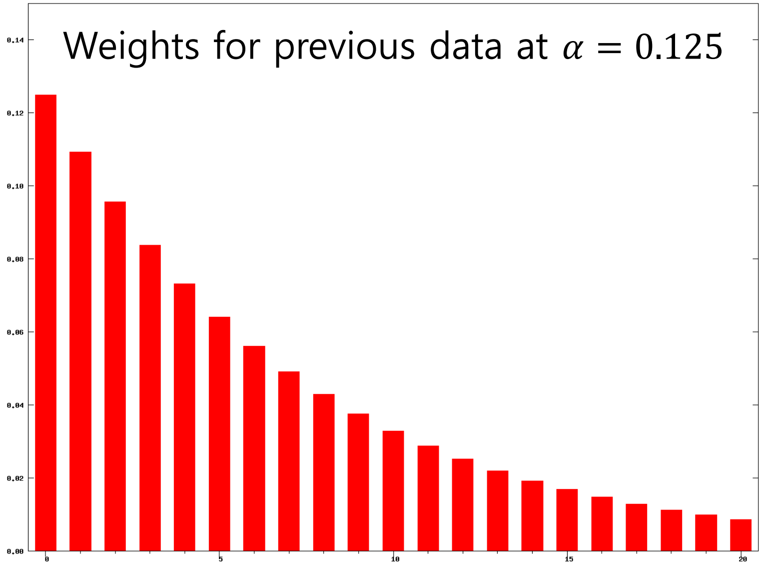
Note) These slides and examples are available at https://github.com/mint-lab/filtering_tutorial.

Weighted Average

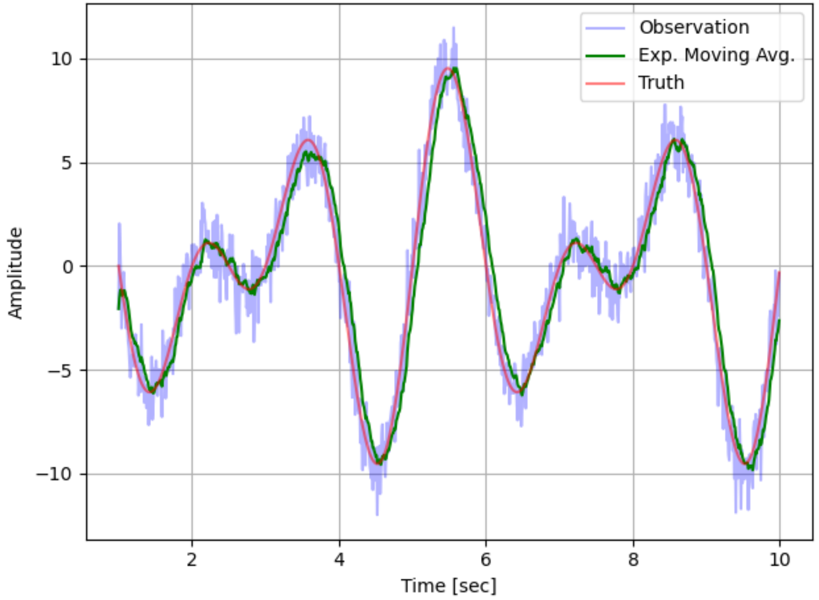
- **Weighted average**
  - Formulation #1: $\bar{x} = \frac{\sum_i w_i x_i}{\sum_i w_i}$ with non-negative weights
  - Formulation #2: $\bar{x} = \sum_i w_i' x_i$ with non-negative **normalized** weights $\sum_i w_i' = 1$
- **Inverse-variance weighted average**
  - Given: Independent measurements x_i with their variances σ_i^2
  - Formulation #1: $\bar{x} = \frac{\sum_i w_i x_i^2}{\sum_i w_i}$ with $w_i = \frac{1}{\sigma_i^2}$
  - Formulation #2: $\bar{x} = \sum_i w_i' x_i$ with $w_i' = \frac{1/\sigma_i^2}{\sum_j 1/\sigma_j^2}$
  - Variance: $\mathrm{Var}(\bar{x}) = \sum_i w_i'^2 \sigma_i^2 = \frac{1}{\sum_i 1/\sigma_i^2}$
    - Note) The variance is the **least variance** among all available weighted averages.
  - Derivation
    - Objective function with Lagrange multiplier λ: $\mathcal{L}(\mathbf{w}', \lambda) = \sum_i w_i'^2 \sigma_i^2 + \lambda(1 - \sum_i w_i')$
    - Finding its minima $\frac{\partial}{\partial w_i'} \mathcal{L}(\mathbf{w}', \lambda) = 0$: $2w_i' \sigma_i^2 - \lambda = 0 \rightarrow w_i' = \frac{\lambda/2}{\sigma_i^2} \rightarrow w_i' = \frac{1/\sigma_i^2}{\sum_j 1/\sigma_j^2}$

Moving Average

- **Exponential moving average** (a.k.a. exponential smoothing, alpha filter)
  - Given: Sequential data
    - x_t: The current measurement at time t
    - $\bar{x}_{t-1}$: The previous averaged value at time $t-1$
  - Formulation (α: weight)
    - $\bar{x}_0 = x_0$
    - $\bar{x}_t = \alpha x_t + (1-\alpha)\bar{x}_{t-1} = \bar{x}_{t-1} + \alpha(x_t - \bar{x}_{t-1})$
      - Note) $(x_t - \bar{x}_{t-1})$ is called as measurement residual (or innovation).



- **Example) 1-D noisy signal filtering**
  - Given: Noisy time-series signal
  - Target: Smooth signal (without noise)
  - Solution: Exponential moving average
    - Parameter: $\alpha = 0.125$

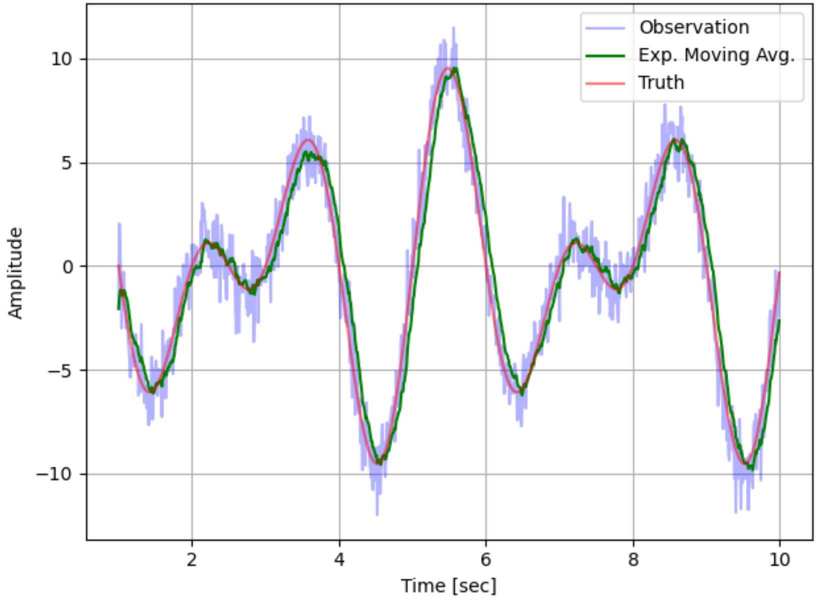


- **Example) 1-D noisy signal filtering** (ema_1d_signal.py)

```python
import numpy as np
import matplotlib.pyplot as plt

if __name__ == '__main__':
    # Prepare a noisy signal
    true_signal = lambda t: 10 * np.sin(2*np.pi/2*t) * np.cos(2*np.pi/10*t)
    times = np.arange(1, 10, 0.01)
    truth = true_signal(times)
    obs_signal = truth + np.random.normal(scale=1, size=truth.size)

    # Perform exponential moving average
    alpha = 0.125
    xs = []
    for z in obs_signal:
        if len(xs) == 0:
            xs.append(z)
        else:
            xs.append(xs[-1] + alpha * (z - xs[-1]))

    # Visualize the results
    plt.figure()
    plt.plot(times, obs_signal, 'b-', label='Observation', alpha=0.3)
    plt.plot(times, xs,         'g-', label='Exp. Moving Avg.')
    plt.plot(times, truth,      'r-', label='Truth', alpha=0.5)
    plt.xlabel('Time [sec]')
    plt.ylabel('Amplitude')
    plt.grid()
    plt.legend()
    plt.show()
```

Q) How to select the parameter α?

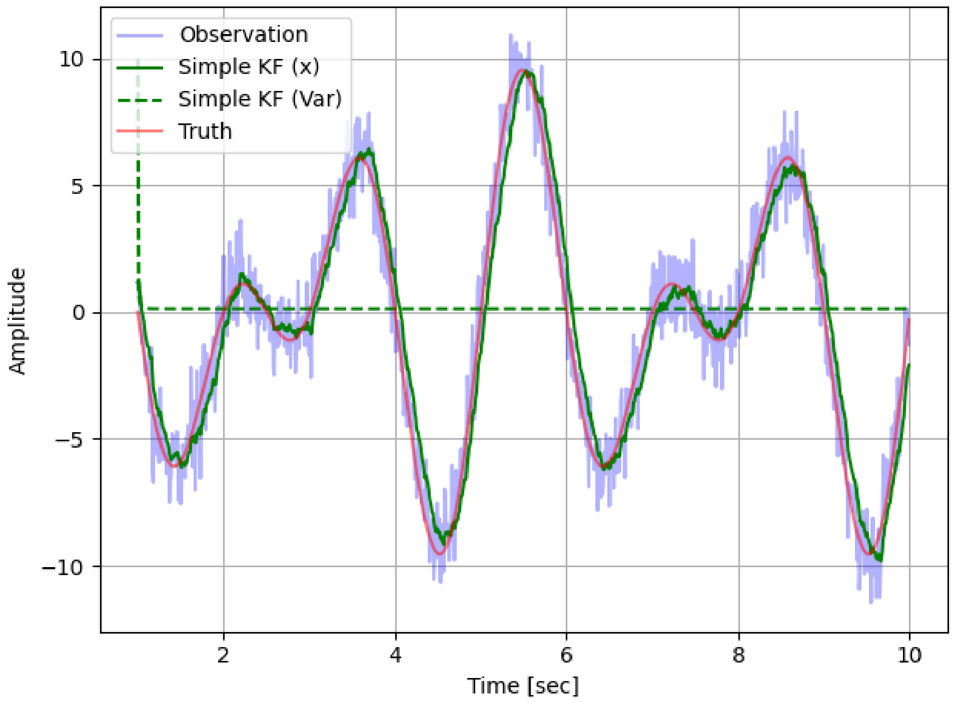
$$\bar{x}_0 = z_0$$
$$\bar{x}_t = \bar{x}_{t-1} + \alpha(z_t - \bar{x}_{t-1})$$



Simple 1-D Kalman Filter

- **Simple 1-D Kalman filter**
  - Idea: Exponential moving average with inverse-variance weight
  - Formulation (z_k: signal at time index k)
    - Initialization: $x_0 = z_0$ σ_0^2
    - Signal prediction: $\hat{x}_k = x_{k-1}$ $\hat{\sigma}_k^2 = \sigma_{k-1}^2 + \sigma_Q^2$ (due to signal change)
    - Signal correction: $x_k = \alpha z_k + (1-\alpha)\hat{x}_k = \hat{x}_k + \alpha(z_k - \hat{x}_k)$ $\sigma_k^2 = \frac{\hat{\sigma}_k^2 \sigma_R^2}{\hat{\sigma}_k^2 + \sigma_R^2} = (1-\alpha)\hat{\sigma}_k^2$
      - Weight $\alpha = \frac{\hat{\sigma}_k^2}{\hat{\sigma}_k^2 + \sigma_R^2}$

- **Example) 1-D noisy signal filtering**
  - Given: Noisy sequential data
  - Target: Smooth sequential data (with noise filtering)
  - Solution: Simple 1-D Kalman filter
    - Parameters: $\sigma_0^2 = 10$, $\sigma_Q^2 = 0.02$, $\sigma_R^2 = 1$
    - Note) The weight α was initially 0.91 and converged to 0.13.



Simple 1-D Kalman Filter

- **Example) 1-D noisy signal filtering** (simple_kf_1d_signal.py)

```python
import numpy as np
import matplotlib.pyplot as plt

if __name__ == '__main__':
    # Prepare a noisy signal
    ...

    # Perform the simple 1-D Kalman filter
    var_init, var_q, var_r = 10, 0.02, 1
    xs, var = [], []
    for z in obs_signal:
        if len(xs) == 0:
            xs.append(z)
            var.append(var_init)
        else:
            # Predict signal change
            pred_x   = xs[-1]
            pred_var = var[-1] + var_q

            # Correct signal change
            alpha = pred_var / (pred_var + var_r)
            xs.append(pred_x + alpha * (z - pred_x))
            var.append((1 - alpha) * pred_var)

    # Visualize the results
    ...
```

	State x	Variance σ^2
Initialization	$x_0 = z_0$	σ_0^2
Prediction	$\hat{x}_k = x_{k-1}$	$\hat{\sigma}_k^2 = \sigma_{k-1}^2 + \sigma_Q^2$
Correction	$x_k = \hat{x}_k + \alpha(z_k - \hat{x}_k)$	$\sigma_k^2 = (1-\alpha)\hat{\sigma}_k^2$

$$\alpha = \frac{\hat{\sigma}_k^2}{\hat{\sigma}_k^2 + \sigma_R^2}$$

Kalman Filter

- Kalman filter is the **optimal** **recursive estimator** for **linear dynamic systems** with unbiased Gaussian noise.
  - Linear dynamic system
    - State variable: $\mathbf{x}$
    - State transition function: $\mathbf{x}_k = f(\mathbf{x}_{k-1}; \mathbf{u}_k) = \mathrm{F}_k \mathbf{x}_{k-1} + \mathrm{B}_k \mathbf{u}_k + \mathbf{w}_k$ where transition noise $\mathbf{w}_k \sim N(0, \mathrm{Q}_k)$
      - Control input: $\mathbf{u}$
    - Observation function: $\mathbf{z}_k = h(\mathbf{x}_k) = \mathrm{H}_k \mathbf{x}_k + \mathbf{v}_k$ where observation noise $\mathbf{v}_k \sim N(0, \mathrm{R}_k)$
      - Observation: $\mathbf{z}$
  - Recursive estimator (P: state covariance)
    - **Prediction**: $\hat{\mathbf{x}}_k = \mathrm{F}_k \mathbf{x}_{k-1} + \mathrm{B}_k \mathbf{u}_k$ $\widehat{\mathrm{P}}_k = \mathrm{F}_k \mathrm{P}_{k-1} \mathrm{F}_k^\top + \mathrm{Q}_k$
    - **Correction**: $\mathbf{x}_k = \hat{\mathbf{x}}_k + \mathrm{K}_k (\mathbf{z}_k - \mathrm{H}_k \hat{\mathbf{x}}_k)$ $\mathrm{P}_k = (1 - \mathrm{K}_k \mathrm{H}_k) \widehat{\mathrm{P}}_k$
      - **Kalman gain**: $\mathrm{K}_k = \widehat{\mathrm{P}}_k \mathrm{H}_k^\top \left(\mathrm{H}_k \widehat{\mathrm{P}}_k \mathrm{H}_k^\top + \mathrm{R}_k \right)^{-1}$
    - Note) Derivation
  - Optimality assumption
    1) The system transition and observation are linear and known.
    2) The noise $\mathbf{w}_k$ and $\mathbf{v}_k$ are unbiased Gaussian noise.

Kalman Filter

- **Review) The simple 1-D Kalman filter**
  - Linear dynamic system
    - **State variable**: $\mathbf{x} = x \quad (\mathrm{P} = \sigma^2)$
    - **State transition function**: $\mathbf{x}_k = f(\mathbf{x}_{k-1}; \mathbf{u}_k) = \mathbf{x}_{k-1} \quad (\mathrm{F}_k = 1,\ \mathrm{B}_k = 0)$ / **State transition noise**: $\mathrm{Q}_k = \sigma_Q^2$
    - **Observation function**: $\mathbf{z}_k = h(\mathbf{x}_k) = \mathbf{x}_k \quad (\mathrm{H}_k = 1)$ / **Observation noise**: $\mathrm{R}_k = \sigma_R^2$
    - Note) The above five definitions are important to design and analyze Bayesian filtering.
  - Recursive estimator

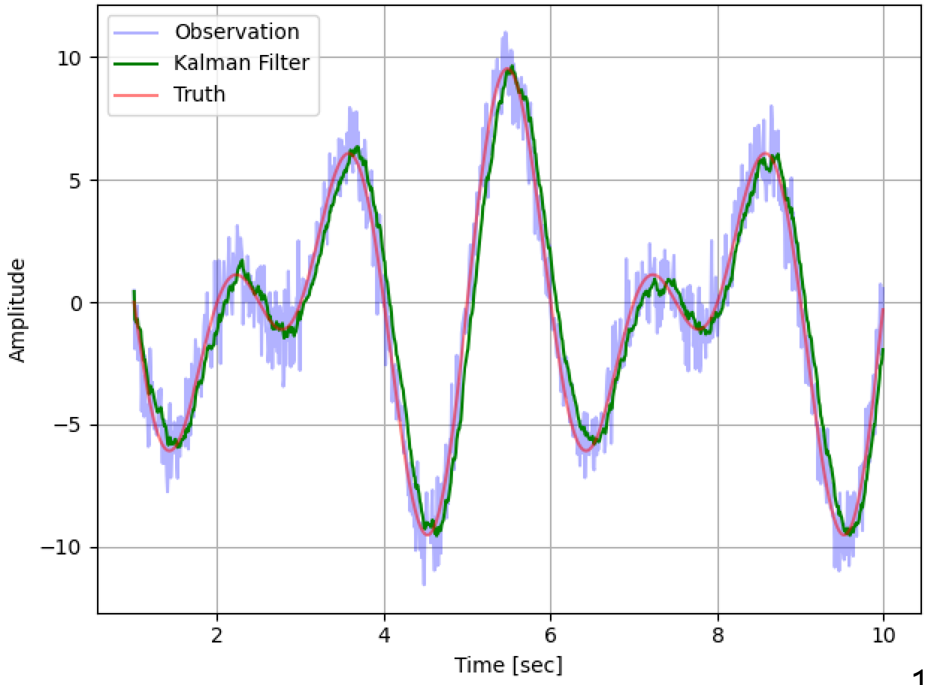
	State $\mathbf{x}$	Covariance P
Prediction	$\hat{\mathbf{x}}_k = \mathrm{F}_k \mathbf{x}_{k-1} + \mathrm{B}_k \mathbf{u}_k$	$\hat{\mathrm{P}}_k = \mathrm{F}_k \mathrm{P}_{k-1} \mathrm{F}_k^\mathrm{T} + \mathrm{Q}_k$
	$\hat{x}_k = x_{k-1}$	$\hat{\sigma}_k^2 = \sigma_{k-1}^2 + \sigma_Q^2$
Correction	$\mathbf{x}_k = \hat{\mathbf{x}}_k + \mathrm{K}_k(\mathbf{z}_k - \mathrm{H}_k \hat{\mathbf{x}}_k)$	$\mathrm{P}_k = (1 - \mathrm{K}_k \mathrm{H}_k)\hat{\mathrm{P}}_k$
	$x_k = \hat{x}_k + \alpha(z_k - \hat{x}_k)$	$\sigma_k^2 = (1-\alpha)\hat{\sigma}_k^2$

- Kalman gain: $\mathrm{K}_k = \hat{\mathrm{P}}_k \mathrm{H}_k^\mathrm{T} \left(\mathrm{H}_k \hat{\mathrm{P}}_k \mathrm{H}_k^\mathrm{T} + \mathrm{R}_k\right)^{-1}$
  - Note) $\alpha = \frac{\hat{\sigma}_k^2}{\hat{\sigma}_k^2 + \sigma_R^2}$

Kalman Filter

- **Example) 1-D noisy signal filtering with [FilterPy]** (kf_1d_signal.py)

```python
import numpy as np
import matplotlib.pyplot as plt
from filterpy.kalman import KalmanFilter

if __name__ == '__main__':
    # Prepare a noisy signal
    ...

    # Instantiate Kalman filter for noise filtering
    kf = KalmanFilter(dim_x=1, dim_z=1)
    kf.F = np.eye(1)
    kf.H = np.eye(1)
    kf.P = 10 * np.eye(1)
    kf.Q = 0.02 * np.eye(1)
    kf.R = 1 * np.eye(1)

    xs = []
    for z in obs_signal:
        # Predict and update the Kalman filter
        kf.predict()
        kf.update(z)
        xs.append(kf.x.flatten())

    # Visualize the results
    ...
```



Kalman Filter

- **Why Bayesian filters?**
  - Kalman filter represents its belief $p(\mathbf{x})$ as Gaussian distribution (mean $\mathbf{x}$ and covariance P).
  - Prediction: $p(\mathbf{x}_{k-1}) \rightarrow p(\mathbf{x}_k|\mathbf{x}_{k-1})$ Belief propagation under Markov assumption
  - Correction: $p(\mathbf{x}_k|\mathbf{z}_k) = \frac{p(\mathbf{z}_k|\mathbf{x}_k)\,p(\mathbf{x}_k|\mathbf{x}_{k-1})}{P(\mathbf{z}_k)}$ Bayesian theorem

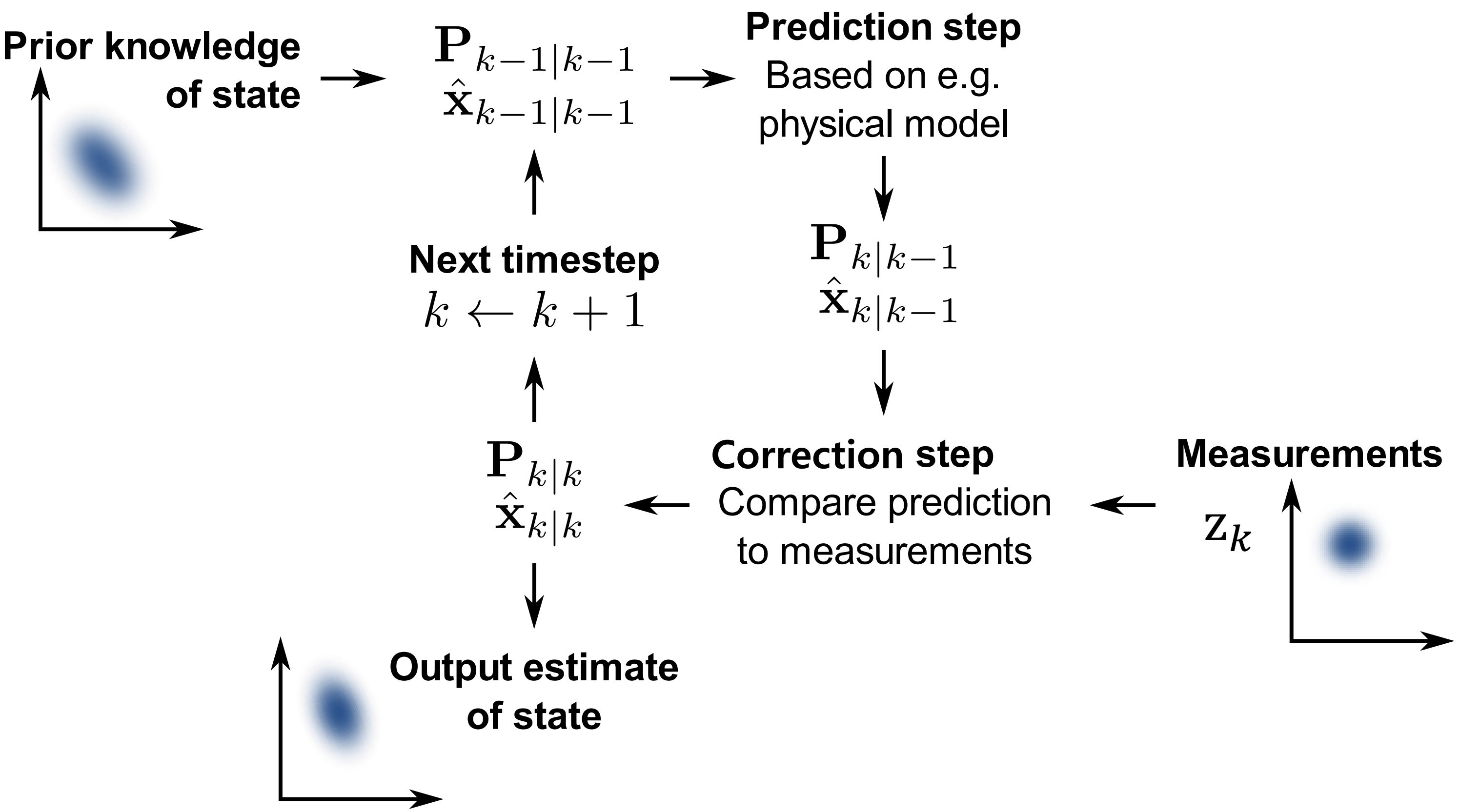


Image: Wikipedia

Kalman Filter

- **Example) 2-D position tracking** `(kf_2d_position.py)`
  - State variable: $\mathbf{x} = [x, y]^\top$
  - State transition function: $\mathbf{x}_{k+1} = f(\mathbf{x}_k; \mathbf{u}_{k+1}) = \mathbf{x}_k \quad (\mathrm{F}_k = \mathrm{I}_{2\times2},\ \mathrm{B}_k = 0)$
    - Control input: $\mathbf{u}_k = [\,]$
  - State transition noise: $\mathrm{Q} = \mathrm{diag}\left(\sigma_x^2, \sigma_y^2\right)$
  - Observation function: $\mathbf{z} = h(\mathbf{x}) = [x, y]^\top \quad (\mathrm{H}_k = \mathrm{I}_{2\times2})$
    - Observation: $\mathbf{z} = [x_{GPS}, y_{GPS}]^\top$
  - Observation noise: $\mathrm{R} = \mathrm{diag}(\sigma_{GPS}^2, \sigma_{GPS}^2)$
  - Note) The above definition is a simple 2-D extension of the previous 1-D signal filter.

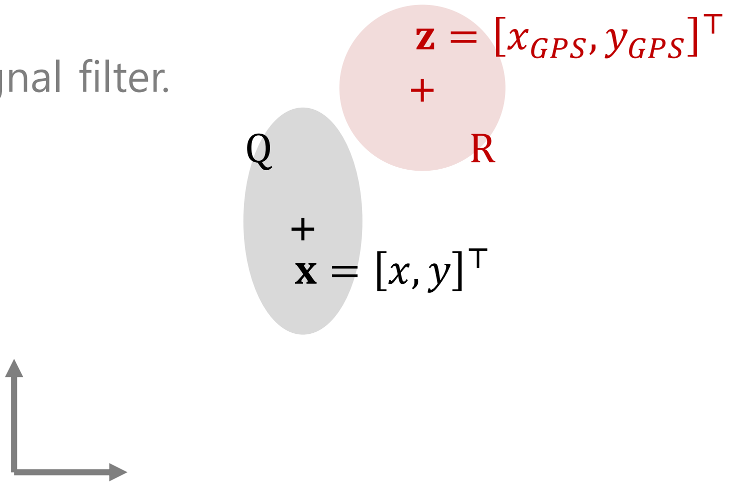
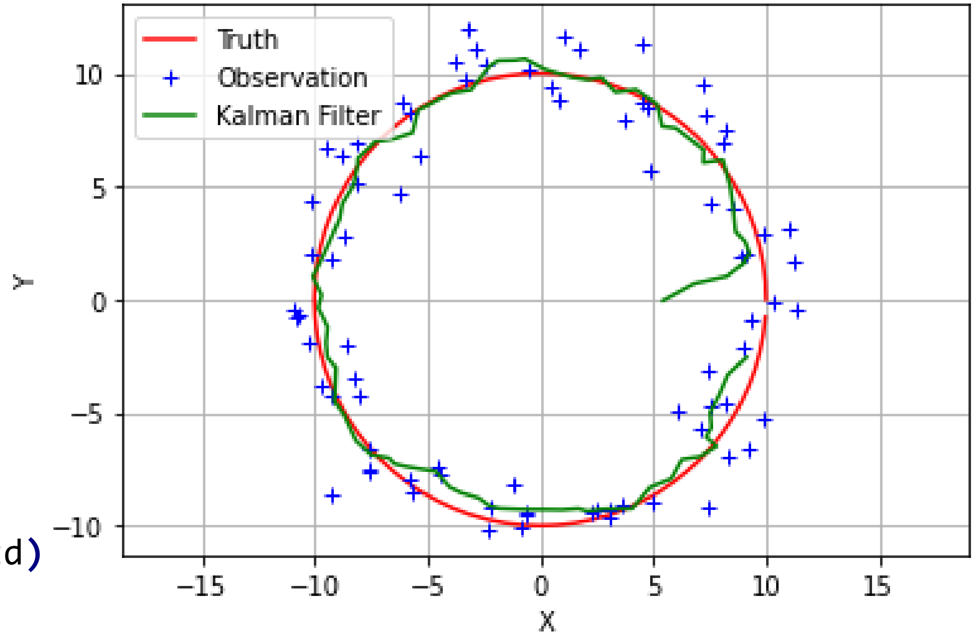
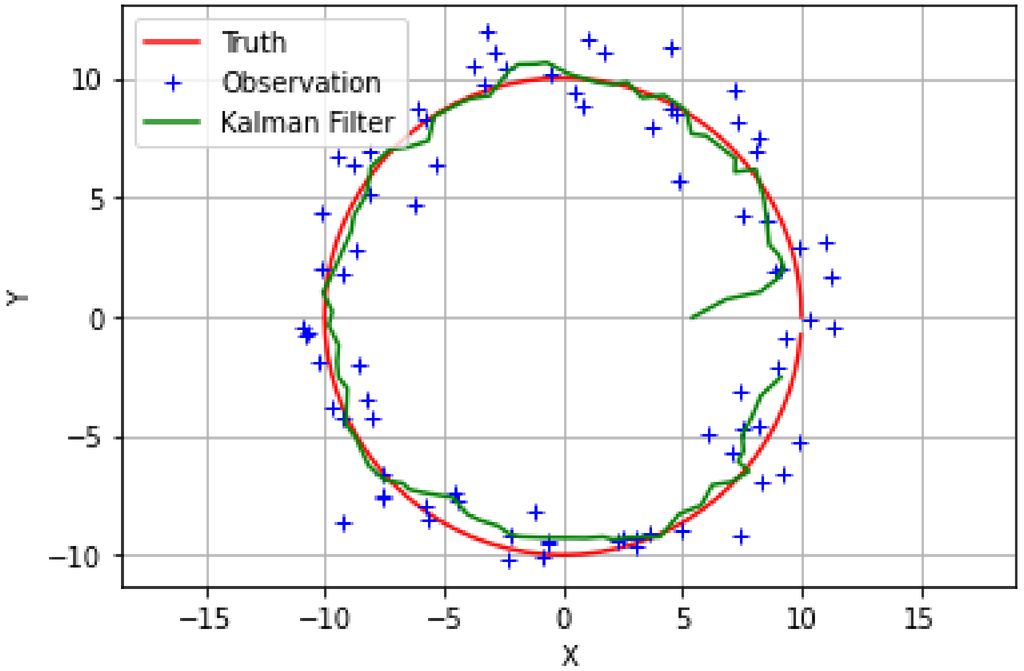


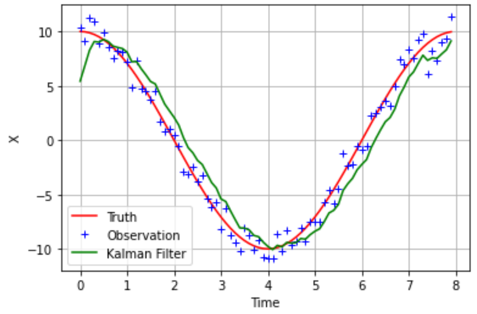
Image: TURBOSQUID

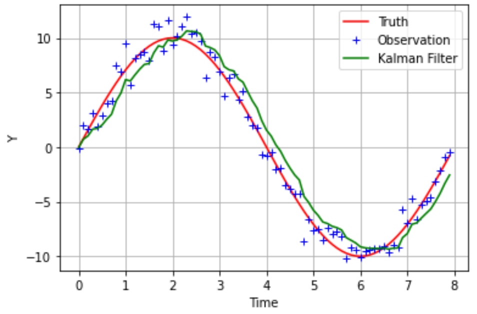
```python
import numpy as np
import matplotlib.pyplot as plt
from filterpy.kalman import KalmanFilter

if __name__ == '__main__':
    # Define experimental configuration
    dt, t_end = 0.1, 8
    r, w = 10., np.pi / 4
    get_true_position = lambda t: r * np.array([[np.cos(w * t)], [np.sin(w * t)]]) # Circular motion
    gps_noise_std = 1

    # Instantiate Kalman filter for position tracking
    localizer_name = 'Kalman Filter'
    localizer = KalmanFilter(dim_x=2, dim_z=2)
    localizer.F = np.eye(2)
    localizer.H = np.eye(2)
    localizer.Q = 0.1 * np.eye(2)
    localizer.R = gps_noise_std * gps_noise_std * np.eye(2)

    times, truth, zs, xs, = [], [], [], []
    for t in np.arange(0, t_end, dt):
        # Simulate position observation with additive Gaussian noise
        true = get_true_position(t)
        z = true + np.random.normal(size=true.shape, scale=gps_noise_std)

        # Predict and update the Kalman filter
        localizer.predict()
        localizer.update(z)

        times.append(t)
        truth.append(true.flatten())
        zs.append(z.flatten())
        xs.append(localizer.x.flatten())
    times, truth, zs, xs = np.array(times), np.array(truth), np.array(zs), np.array(xs)
```



```python
# Visualize the results
plt.figure()
plt.plot(truth[:,0], truth[:,1], 'r-', label='Truth')
plt.plot(zs[:,0],    zs[:,1],    'b+', label='Observation')
plt.plot(xs[:,0],    xs[:,1],    'g-', label=localizer_name)
plt.axis('equal')
plt.xlabel('X')
plt.ylabel('Y')
plt.grid()
plt.legend()

plt.figure()
plt.plot(times, truth[:,0], 'r-', label='Truth')
plt.plot(times, zs[:,0],    'b+', label='Observation')
plt.plot(times, xs[:,0],    'g-', label=localizer_name)
plt.xlabel('Time')
plt.ylabel('X')
plt.grid()
plt.legend()

plt.figure()
plt.plot(times, truth[:,1], 'r-', label='Truth')
plt.plot(times, zs[:,1],    'b+', label='Observation')
plt.plot(times, xs[:,1],    'g-', label=localizer_name)
plt.xlabel('Time')
plt.ylabel('Y')
plt.grid()
plt.legend()

plt.show()
```







Extended Kalman Filter

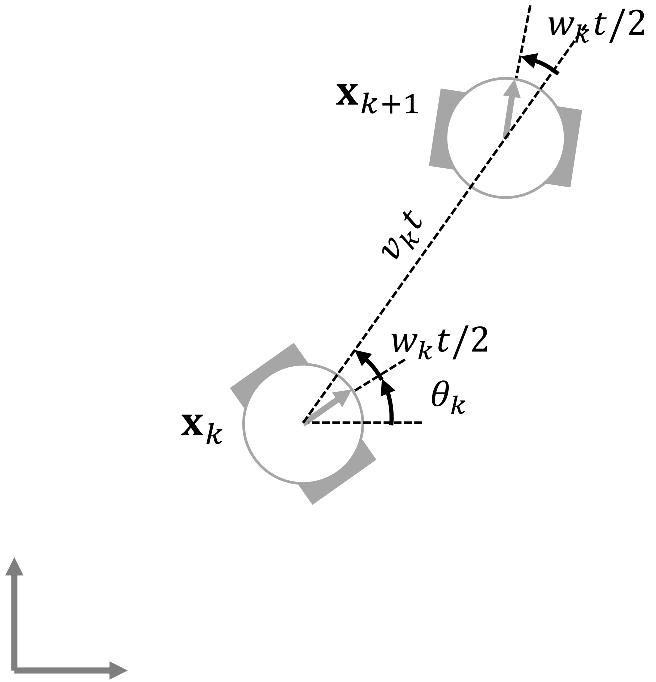
- Extended Kalman filter (shortly EKF) is a **nonlinear** **version of Kalman filter** using **linearization**.
  - ~~Linear~~ dynamic system
    - State variable: $\mathbf{x}$
    - State transition function: $\mathbf{x}_k = f(\mathbf{x}_{k-1}; \mathbf{u}_k) \cancel{= \mathrm{F}_k \mathbf{x}_{k-1} + \mathrm{B}_k \mathbf{u}_k + \mathbf{w}_k}$ where transition noise $\mathbf{w}_k \sim N(0, \mathrm{Q}_k)$
      - Control input: $\mathbf{u}$
    - Observation function: $\mathbf{z}_k = h(\mathbf{x}_k) \cancel{= \mathrm{H}_k \mathbf{x}_k + \mathbf{v}_k}$ where observation noise $\mathbf{v}_k \sim N(0, \mathrm{R}_k)$
      - Observation: $\mathbf{z}$
  - Recursive estimator (P: state covariance)
    - **Prediction**: $\hat{\mathbf{x}}_k = f(\mathbf{x}_{k-1}; \mathbf{u}_k)$ $\quad \widehat{\mathrm{P}}_k = \mathrm{F}_k \mathrm{P}_{k-1} \mathrm{F}_k^\top + \mathrm{Q}_k$ where $\mathrm{F}_k = \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}; \mathbf{u})\Big|_{\mathbf{x}=\mathbf{x}_{k-1}, \mathbf{u}=\mathbf{u}_{k-1}}$
    - **Correction**: $\mathbf{x}_k = \hat{\mathbf{x}}_k + \mathrm{K}_k(\mathbf{z}_k - h(\hat{\mathbf{x}}_k))$ $\quad \mathrm{P}_k = (1 - \mathrm{K}_k \mathrm{H}_k)\widehat{\mathrm{P}}_k$ where $\mathrm{H}_k = \frac{\partial}{\partial \mathbf{x}} h(\mathbf{x})\Big|_{\mathbf{x}=\hat{\mathbf{x}}_k}$
      - **Kalman gain**: $\mathrm{K}_k = \widehat{\mathrm{P}}_k \mathrm{H}_k^\top \left(\mathrm{H}_k \widehat{\mathrm{P}}_k \mathrm{H}_k^\top + \mathrm{R}_k\right)^{-1}$
  - Note) Optimality: EKF is not an optimal nonlinear estimator but widely used.

Extended Kalman Filter

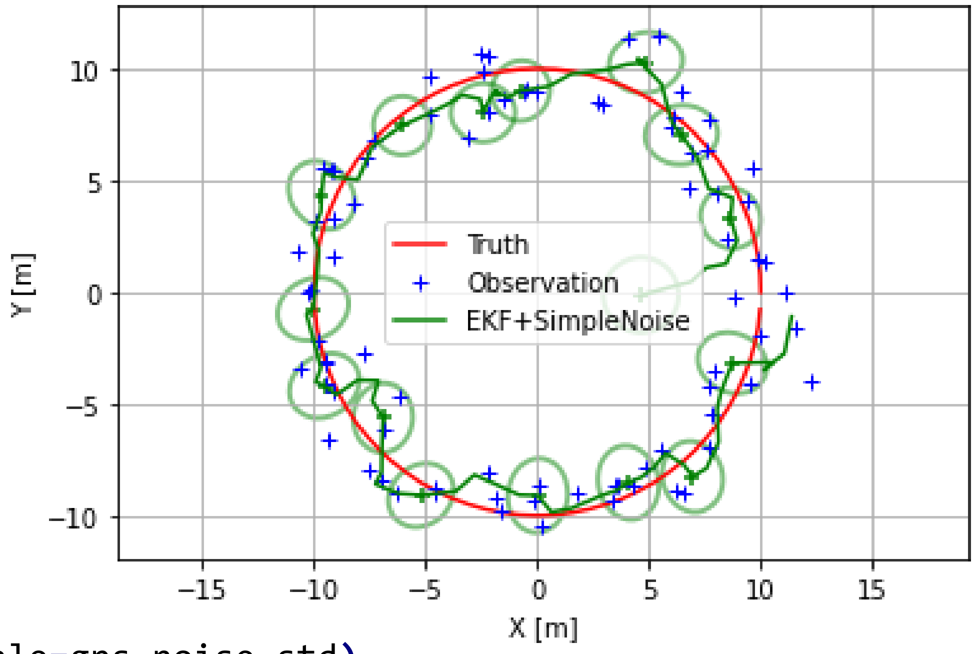
- **Example) 2-D pose tracking with simple transition noise** (ekf_2d_pose_simple_noise.py)
  - State variable: $\mathbf{x} = [x, y, \theta, v, w]^\top$
  - State transition function: Constant velocity model (time interval: t)

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k; \mathbf{u}_{k+1}) = \begin{bmatrix} x_k + v_k t \cos(\theta_k + w_k t/2) \\ y_k + v_k t \sin(\theta_k + w_k t/2) \\ \theta_k + w_k t \\ v_k \\ w_k \end{bmatrix}$$

    - Control input: $\mathbf{u}_k = [\,]$
  - State transition noise: $\mathrm{Q} = \mathrm{diag}\left(\sigma_x^2, \sigma_y^2, \sigma_\theta^2, \sigma_v^2, \sigma_w^2\right)$
  - Observation function: $\mathbf{z} = h(\mathbf{x}) = [x, y]^\top$
    - Observation: $\mathbf{z} = [x_{GPS}, y_{GPS}]^\top$
  - Observation noise: $\mathrm{R} = \mathrm{diag}(\sigma_{GPS}^2, \sigma_{GPS}^2)$



```python
if __name__ == '__main__':
    # Define experimental configuration
    ...

    # Instantiate EKF for pose (and velocity) tracking
    localizer_name = 'EKF+SimpleNoise'
    localizer = ExtendedKalmanFilter(dim_x=5, dim_z=2)
    localizer.Q = 0.1 * np.eye(5)
    localizer.R = gps_noise_std * gps_noise_std * np.eye(2)

    truth, state, obser, covar = [], [], [], []
    for t in np.arange(0, t_end, dt):
        # Simulate position observation with additive Gaussian noise
        true_pos = get_true_position(t)
        true_ori = get_true_heading(t)
        gps_data = true_pos + np.random.normal(size=true_pos.shape, scale=gps_noise_std)

        # Predict and update the EKF
        localizer.F = Fx(localizer.x, dt)
        localizer.x = fx(localizer.x, dt)
        localizer.predict()
        localizer.update(gps_data, Hx, hx)

        if localizer.x[2] >= np.pi:
            localizer.x[2] -= 2 * np.pi
        elif localizer.x[2] < -np.pi:
            localizer.x[2] += 2 * np.pi

        # Record true state, observation, estimated state, and its covariance
        ...

    # Visualize the results
    ...
```



```python
import numpy as np
import matplotlib.pyplot as plt
from filterpy.kalman import ExtendedKalmanFilter
from ekf_2d_pose import plot_results

def fx(state, dt):
    x, y, theta, v, w = state.flatten()
    vt, wt = v * dt, w * dt
    s, c = np.sin(theta + wt / 2), np.cos(theta + wt / 2)
    return np.array([
        [x + vt * c],
        [y + vt * s],
        [theta + wt],
        [v],
        [w]])

def Fx(state, dt):
    x, y, theta, v, w = state.flatten()
    vt, wt = v * dt, w * dt
    s, c = np.sin(theta + wt / 2), np.cos(theta + wt / 2)
    return np.array([
        [1, 0, -vt * s, dt * c, -vt * dt * s / 2],
        [0, 1,  vt * c, dt * s,  vt * dt * c / 2],
        [0, 0,       1,      0,               dt],
        [0, 0,       0,      1,                0],
        [0, 0,       0,      0,                1]])

def hx(state):
    x, y, *_ = state.flatten()
    return np.array([[x], [y]])

def Hx(state):
    return np.eye(2, 5)
```

$$\mathbf{x} = [x, y, \theta, v, w]^\top$$

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k; \mathbf{u}_{k+1}) = \begin{bmatrix} x_k + v_k t\cos(\theta_k + w_k t/2) \\ y_k + v_k t\sin(\theta_k + w_k t/2) \\ \theta_k + w_k t \\ v_k \\ w_k \end{bmatrix}$$

$$\mathrm{F}_k = \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}; \mathbf{u})\bigg|_{\mathbf{x}=\mathbf{x}_{k-1}, \mathbf{u}=\mathbf{u}_{k-1}}$$

$$\mathbf{z} = h(\mathbf{x}) = \begin{bmatrix} x \\ y \end{bmatrix}$$

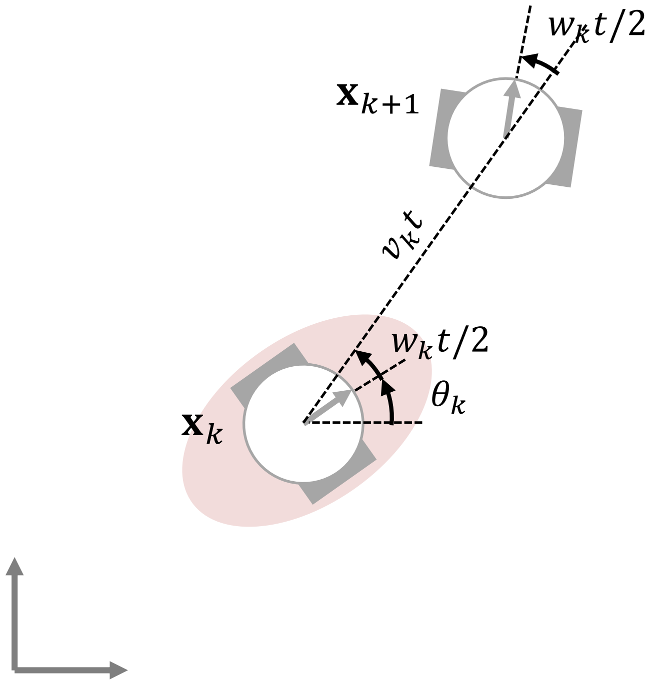
$$\mathrm{H}_k = \frac{\partial}{\partial \mathbf{x}} h(\mathbf{x})\bigg|_{\mathbf{x}=\hat{\mathbf{x}}_k}$$

Extended Kalman Filter

- **Example) 2-D pose tracking** `(ekf_2d_pose.py)`
  - State variable: $\mathbf{x} = [x, y, \theta, v, w]^\top$
  - State transition function: Constant velocity model (time interval: t)

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k; \mathbf{u}_{k+1}) = \begin{bmatrix} x_k + v_k t \cos(\theta_k + w_k t/2) \\ y_k + v_k t \sin(\theta_k + w_k t/2) \\ \theta_k + w_k t \\ v_k \\ w_k \end{bmatrix}$$

    - Control input: $\mathbf{u}_k = [\,]$
  - State transition noise: $\mathrm{Q} = \mathrm{W}\mathrm{M}\mathrm{W}^\top$ where $\mathrm{W} = \begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial f}{\partial w} \end{bmatrix}$ and $\mathrm{M} = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_w^2 \end{bmatrix}$
  - Observation function: $\mathbf{z} = h(\mathbf{x}) = [x, y]^\top$
    - Observation: $\mathbf{z} = [x_{GPS}, y_{GPS}]^\top$
  - Observation noise: $\mathrm{R} = \mathrm{diag}(\sigma_{GPS}^2, \sigma_{GPS}^2)$

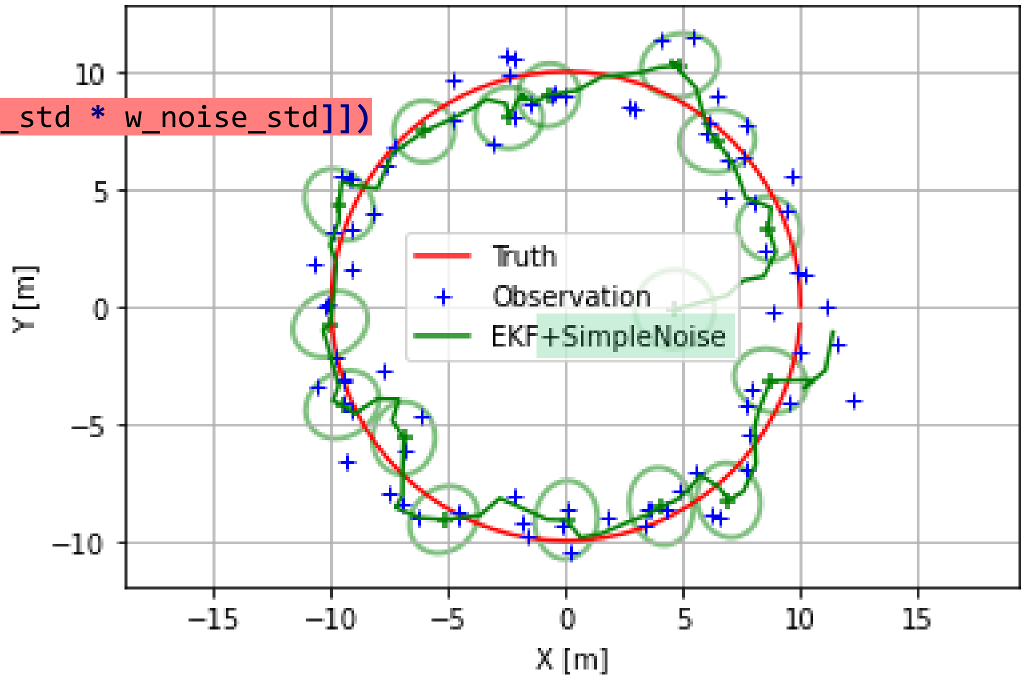


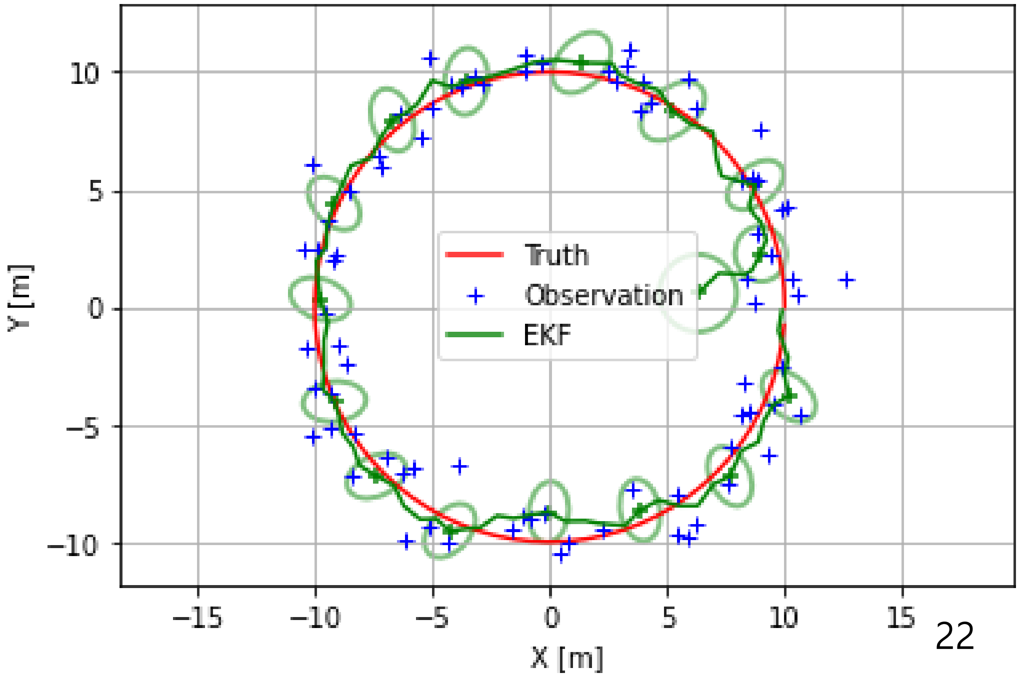
```python
class EKFLocalizer(ExtendedKalmanFilter):
    def __init__(self, v_noise_std=1, w_noise_std=1, gps_noise_std=1, dt=1):
        super().__init__(dim_x=5, dim_z=2)
        self.motion_noise = np.array([[v_noise_std * v_noise_std, 0], [0, w_noise_std * w_noise_std]])
        self.h = lambda x: x[0:2]
        self.H = lambda x: np.eye(2, 5)
        self.R = gps_noise_std * gps_noise_std * np.eye(2)
        self.dt = dt

    def predict(self):
        x, y, theta, v, w = self.x.flatten()
        vt, wt = v * self.dt, w * self.dt
        s, c = np.sin(theta + wt / 2), np.cos(theta + wt / 2)

        # Predict the state
        self.x[0] = x + vt * c
        self.x[1] = y + vt * s
        self.x[2] = theta + wt
        #self.x[3] = v # Not necessary
        #self.x[4] = w # Not necessary

        # Predict the covariance
        self.F = np.array([
            [1, 0, -vt * s, self.dt * c, -vt * self.dt * s / 2],
            [0, 1,  vt * c, self.dt * s,  vt * self.dt * c / 2],
            [0, 0,       1,           0,               self.dt],
            [0, 0,       0,           1,                     0],
            [0, 0,       0,           0,                     1]])
        W = np.array([
            [self.dt * c, -vt * self.dt * s / 2],
            [self.dt * s,  vt * self.dt * c / 2],
            [0, self.dt],
            [1, 0],
            [0, 1]])
        self.Q = W @ self.motion_noise @ W.T
        self.P = self.F @ self.P @ self.F.T + self.Q

        # Save prior
        self.x_prior = np.copy(self.x)
        self.P_prior = np.copy(self.P)

    def update(self, z):
        super().update(z, HJacobian=self.H, Hx=self.h, R=self.R)
```

$$M = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_w^2 \end{bmatrix}$$

$$W = \begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial f}{\partial w} \end{bmatrix}$$

$$Q = WMW^T$$



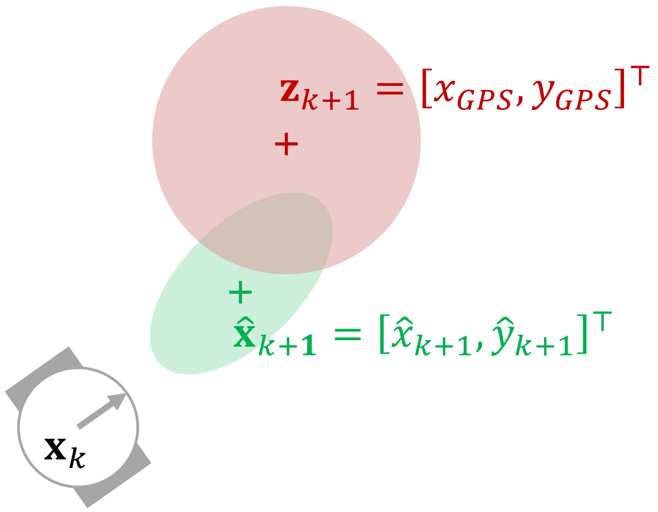


Extended Kalman Filter

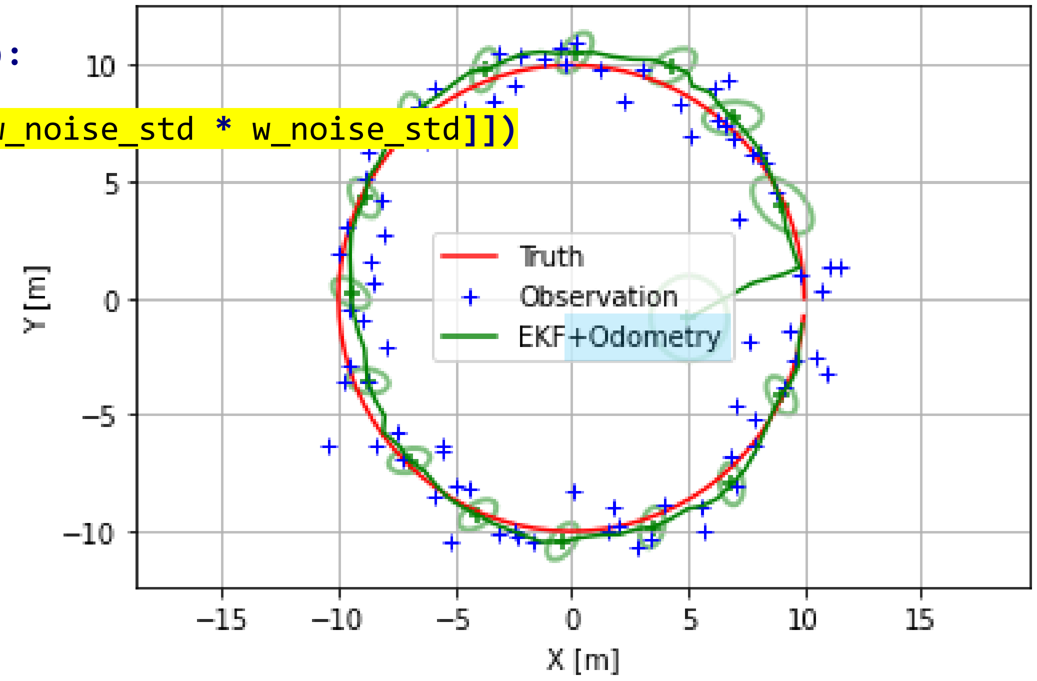
- **Example) 2-D pose tracking with odometry** (ekf_2d_pose_odometry.py)
  - State variable: $\mathbf{x} = [x, y, \theta]^\top$
  - State transition function: Constant velocity model (time interval: t)

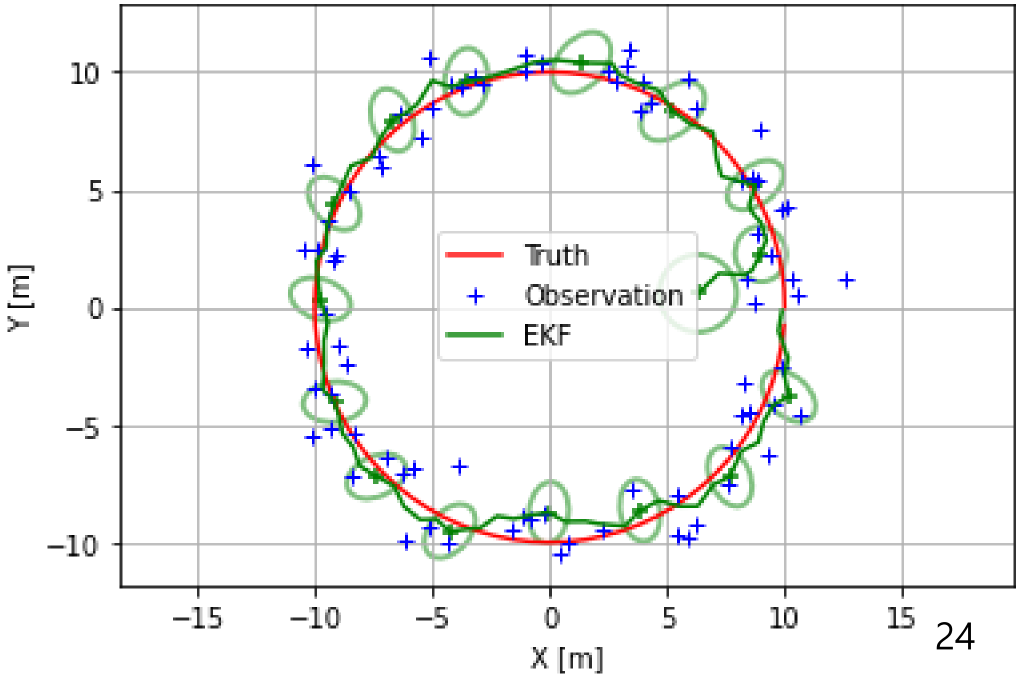
$$\mathbf{x}_{k+1} = f(\mathbf{x}_k; \mathbf{u}_{k+1}) = \begin{bmatrix} x_k + v_{k+1} t \cos(\theta_k + w_{k+1} t/2) \\ y_k + v_{k+1} t \sin(\theta_k + w_{k+1} t/2) \\ \theta_k + w_{k+1} t \end{bmatrix}$$

  - Control input: $\mathbf{u}_k = [v_k, w_k]$
    - e.g. *Wheel odometry* is more precise than GPS, but has drift error due to wheel slippage.
      - Note) It is possible to use $\mathbf{u}_k = [\rho_k, \Delta\theta_k]$ instead of $v_k t$ and $w_k t$.
  - State transition noise: $\mathrm{Q} = \mathrm{W}\mathrm{M}\mathrm{W}^\top$ where $\mathrm{W} = \begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial f}{\partial w} \end{bmatrix}$ and $\mathrm{M} = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_w^2 \end{bmatrix}$
  - Observation function: $\mathbf{z} = h(\mathbf{x}) = [x, y]^\top$
    - Observation: $\mathbf{z} = [x_{GPS}, y_{GPS}]^\top$
      - e.g. *GPS* is less accurate, but does not have drift error.
  - Observation noise: $\mathrm{R} = \mathrm{diag}(\sigma_{GPS}^2, \sigma_{GPS}^2)$



```
class EKFLocalizerOD(ExtendedKalmanFilter):
    def __init__(self, v_noise_std=1, w_noise_std=1, gps_noise_std=1, dt=1):
        super().__init__(dim_x=3, dim_z=2)
        self.motion_noise = np.array([[v_noise_std * v_noise_std, 0], [0, w_noise_std * w_noise_std]])
        self.h = lambda x: x[0:2]
        self.H = lambda x: np.eye(2, 3)
        self.R = gps_noise_std * gps_noise_std * np.eye(2)
        self.dt = dt

    def predict(self, u):
        x, y, theta = self.x.flatten()
        v, w = u.flatten()
        vt, wt = v * self.dt, w * self.dt
        s, c = np.sin(theta + wt / 2), np.cos(theta + wt / 2)

        # Predict the state
        self.x[0] = x + vt * c
        self.x[1] = y + vt * s
        self.x[2] = theta + wt

        # Predict the covariance
        self.F = np.array([
            [1, 0, -vt * s],
            [0, 1,  vt * c],
            [0, 0,       1]])
        W = np.array([
            [self.dt * c, -vt * self.dt * s / 2],
            [self.dt * s,  vt * self.dt * c / 2],
            [0, self.dt]])
        self.Q = W @ self.motion_noise @ W.T
        self.P = self.F @ self.P @ self.F.T + self.Q

        # Save prior
        self.x_prior = np.copy(self.x)
        self.P_prior = np.copy(self.P)

    def update(self, z):
        super().update(z, HJacobian=self.H, Hx=self.h, R=self.R)
```



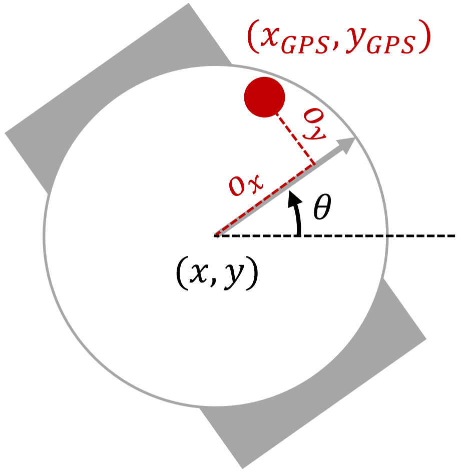


Extended Kalman Filter

- **Example) 2-D pose tracking with off-centered GPS [1]** `(ekf_2d_pose_off_centered.py)`
  - State variable: $\mathbf{x} = [x, y, \theta, v, w]^\top$
  - State transition function: Constant velocity model (time interval: t)

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k; \mathbf{u}_{k+1}) = \begin{bmatrix} x_k + v_k t \cos(\theta_k + w_k t/2) \\ y_k + v_k t \sin(\theta_k + w_k t/2) \\ \theta_k + w_k t \\ v_k \\ w_k \end{bmatrix}$$

    - Control input: $\mathbf{u}_k = [\,]$
  - State transition noise: $\mathrm{Q} = \mathrm{W}\mathrm{M}\mathrm{W}^\top$where $\mathrm{W} = \begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial f}{\partial w} \end{bmatrix}$ and $\mathrm{M} = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_w^2 \end{bmatrix}$
  - Observation function: $\mathbf{z} = h(\mathbf{x}) = \begin{bmatrix} x + o_x \cos\theta - o_y \sin\theta \\ y + o_x \sin\theta + o_y \cos\theta \end{bmatrix}$
    - Note) o_x and o_y are frontal and lateral offset of the GPS.
    - Observation: $\mathbf{z} = [x_{GPS}, y_{GPS}]^\top$
  - Observation noise: $\mathrm{R} = \mathrm{diag}(\sigma_{GPS}^2, \sigma_{GPS}^2)$



[1] Choi and Kim, "Leveraging Localization Accuracy With Off-Centered GPS", T-ITS, 2020

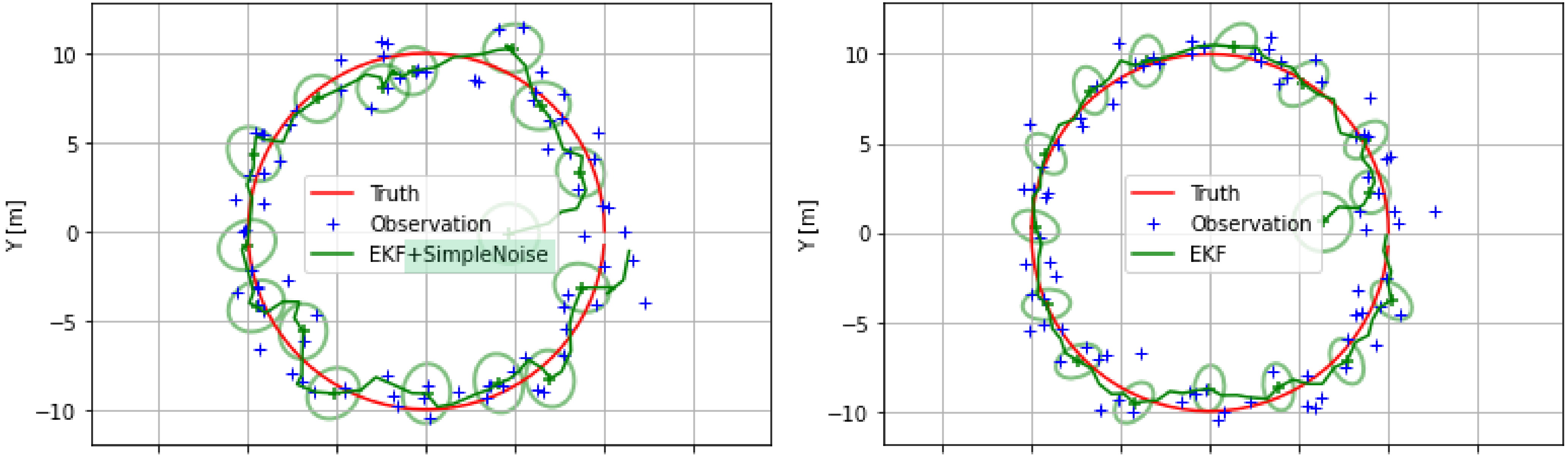
Extended Kalman Filter

- **Example) 2-D pose tracking with off-centered GPS [1]** (ekf_2d_pose_off_centered.py)

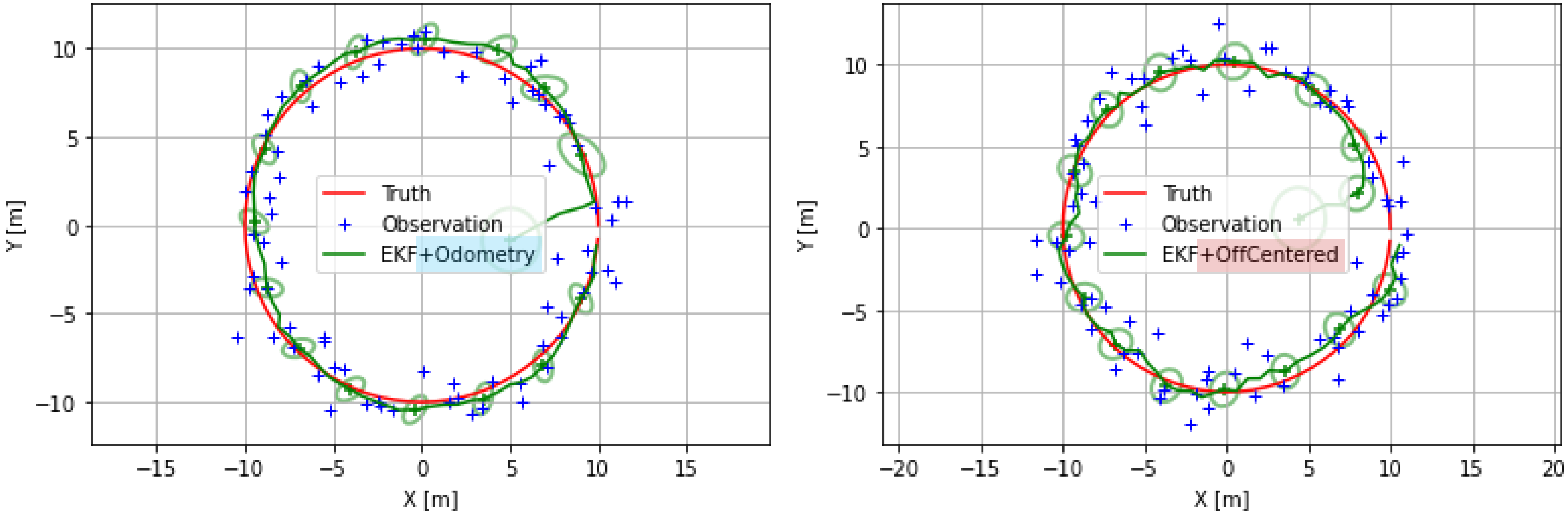
```
class EKFLocalizerOC(EKFLocalizer):
    def __init__(self, v_noise_std=1, w_noise_std=1, gps_noise_std=1, gps_offset=(0,0), dt=1):
        super().__init__(v_noise_std, w_noise_std, gps_noise_std, dt)
        self.h = self.hx
        self.H = self.Hx
        self.gps_offset_x, self.gps_offset_y = gps_offset.flatten()

    def hx(self, state):
        x, y, theta, *_ = state.flatten()
        s, c = np.sin(theta), np.cos(theta)
        return np.array([
            [x + self.gps_offset_x * c - self.gps_offset_y * s],
            [y + self.gps_offset_x * s + self.gps_offset_y * c]])

    def Hx(self, state):
        _, _, theta, *_ = state.flatten()
        s, c = np.sin(theta), np.cos(theta)
        return np.array([
            [1, 0, -self.gps_offset_x * s - self.gps_offset_y * c, 0, 0],
            [0, 1,  self.gps_offset_x * c - self.gps_offset_y * s, 0, 0]])
```

$$\mathbf{z} = h(\mathbf{x}) = \begin{bmatrix} x + o_x \cos\theta - o_y \sin\theta \\ y + o_x \sin\theta + o_y \cos\theta \end{bmatrix}$$

$$\mathrm{H}_k = \frac{\partial}{\partial \mathbf{x}} h(\mathbf{x})\bigg|_{\mathbf{x}=\hat{\mathbf{x}}_k}$$

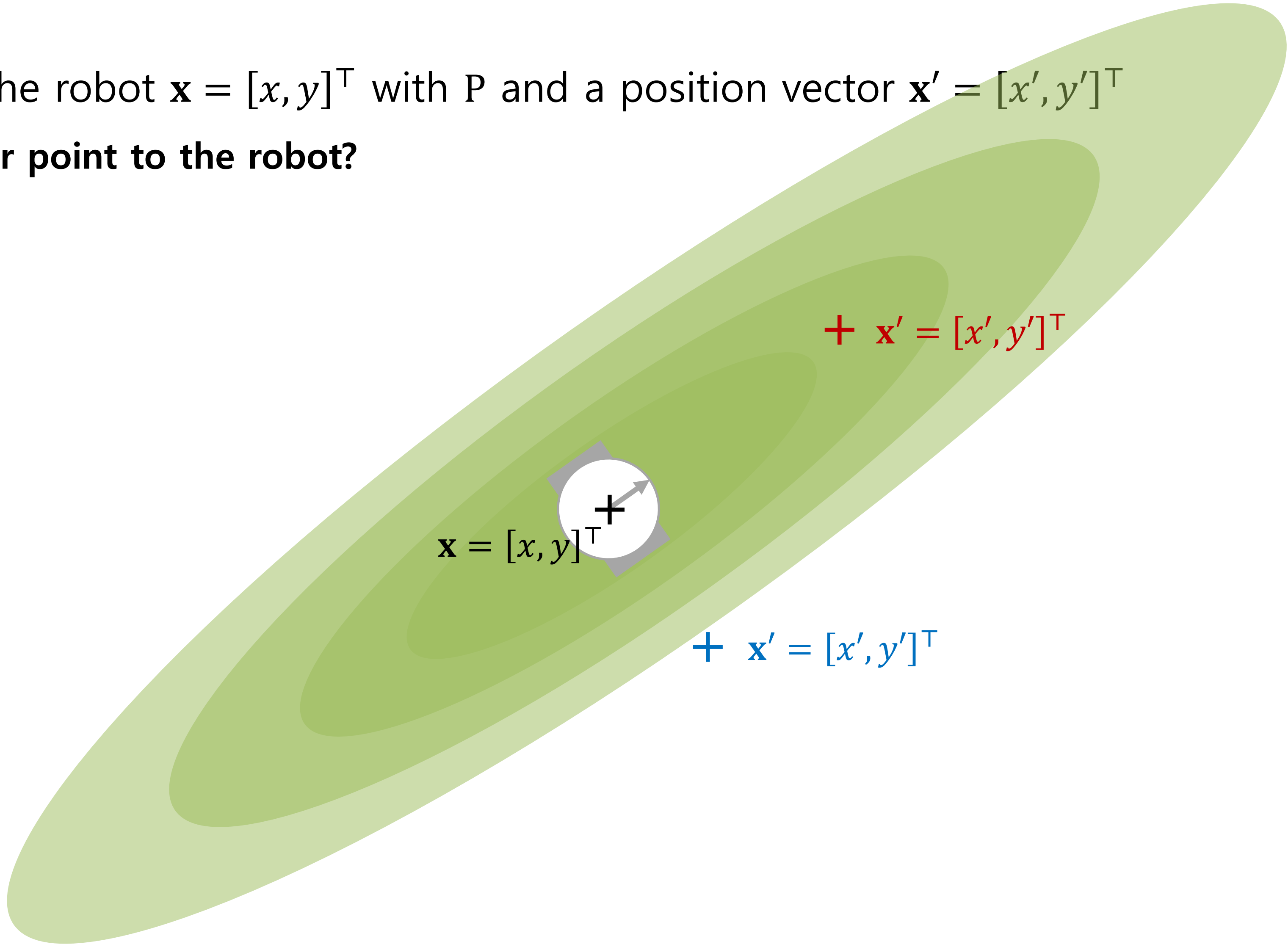


Note) The five definitions (x, f, h, Q, and R) are important to design and analyze Bayesian filtering.



Note) Mahalanobis Distance

- Distance between the robot $\mathbf{x} = [x, y]^\top$ with P and a position vector $\mathbf{x}' = [x', y']^\top$
  - **Q) What is closer point to the robot?**



Note) Mahalanobis Distance

- Distance between the robot $\mathbf{x} = [x, y]^\top$ with P and a position vector $\mathbf{x}' = [x', y']^\top$
  - Euclidean distance (L2-norm)
    - $d_e(\mathbf{x}', \mathbf{x}) = \sqrt{(x'-x)^2 + (y'-y)^2} = \|\mathbf{x}' - \mathbf{x}\|_2 = \left((\mathbf{x}' - \mathbf{x})^\top (\mathbf{x}' - \mathbf{x})\right)^{\frac{1}{2}}$
    - Note) Euclidean distance does not consider the robot covariance P.
  - Mahalanobis distance
    - $d_m(\mathbf{x}', \mathbf{x}) = \left((\mathbf{x}' - \mathbf{x})^\top \, \mathrm{P}^{-1} \, (\mathbf{x}' - \mathbf{x})\right)^{\frac{1}{2}}$

- Distance between the robot $\mathbf{x} = [x, y, \theta]^\top$ with P and a pose vector $\mathbf{x}' = [x', y', \theta']^\top$
  - ~~Euclidean distance (L2-norm)~~
    - $\cancel{d_e(\mathbf{x}', \mathbf{x}) = \sqrt{(x'-x)^2 + (y'-y)^2 + (\theta'-\theta)^2} = \|\mathbf{x}' - \mathbf{x}\|_2 = \left((\mathbf{x}' - \mathbf{x})^\top (\mathbf{x}' - \mathbf{x})\right)^{\frac{1}{2}}}$
    - Q) What is a good scaling for position (x, y) and orientation θ?
  - Mahalanobis distance
    - $d_m(\mathbf{x}', \mathbf{x}) = \left((\mathbf{x}' - \mathbf{x})^\top \, \mathrm{P}^{-1} \, (\mathbf{x}' - \mathbf{x})\right)^{\frac{1}{2}}$
    - Note) The covariance P can describe the scale of position and orientation.

Unscented Kalman Filter

- Unscented Kalman filter (shortly UKF) is a **nonlinear version of Kalman filter** using a deterministic **sampling technique** known as unscented transformation.
  - Why? Linearized covariance in EKF has large error when f and h are highly nonlinear.
  - Unscented transformation (shortly UT) approximates mean $\bar{\mathbf{y}}$ and covariance $\mathbf{P}_y$ using sigma points χ with an exact nonlinear function $\mathbf{y} = \mathbf{f}(\mathbf{x})$.
    - Transformation $\mathbf{f}$: From source $\mathbf{x}$ to target $\mathbf{y}$
    - Step #1) Extract sigma points χ from $\bar{\mathbf{x}}$ and $\mathbf{P}_x$
      (The weights of χ are also derived.)
    - Step #2) Transform sigma points χ using $\mathbf{y} = \mathbf{f}(\mathbf{x})$
    - Step #3) Calculate $\bar{\mathbf{y}}$ and $\mathbf{P}_y$ from *transformed* sigma points χ and their weights
  - Note) EKF and UKF usually have similar performance.
    However, UKF does not require Jacobian matrices.

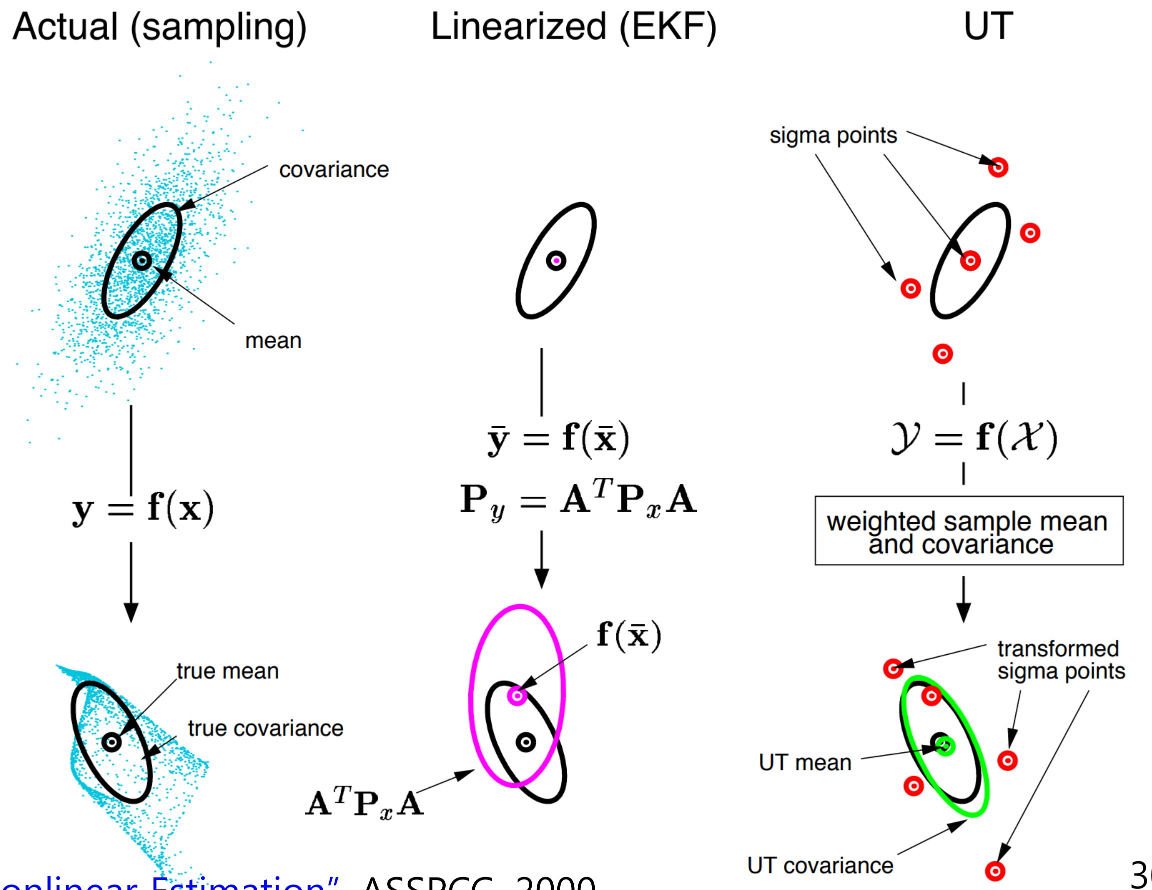


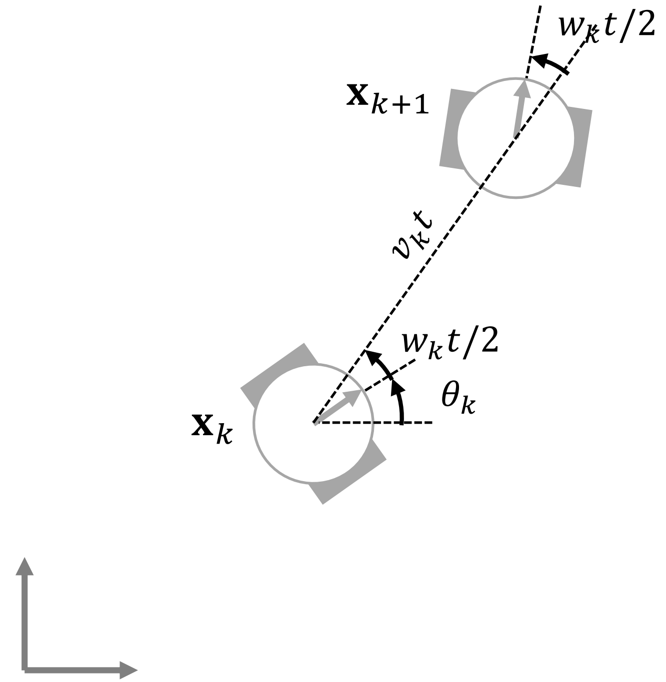
Image: E. A. Wan and R. Van Der Merwe, "The Unscented Kalman Filter for Nonlinear Estimation", ASSPCC, 2000

Unscented Kalman Filter

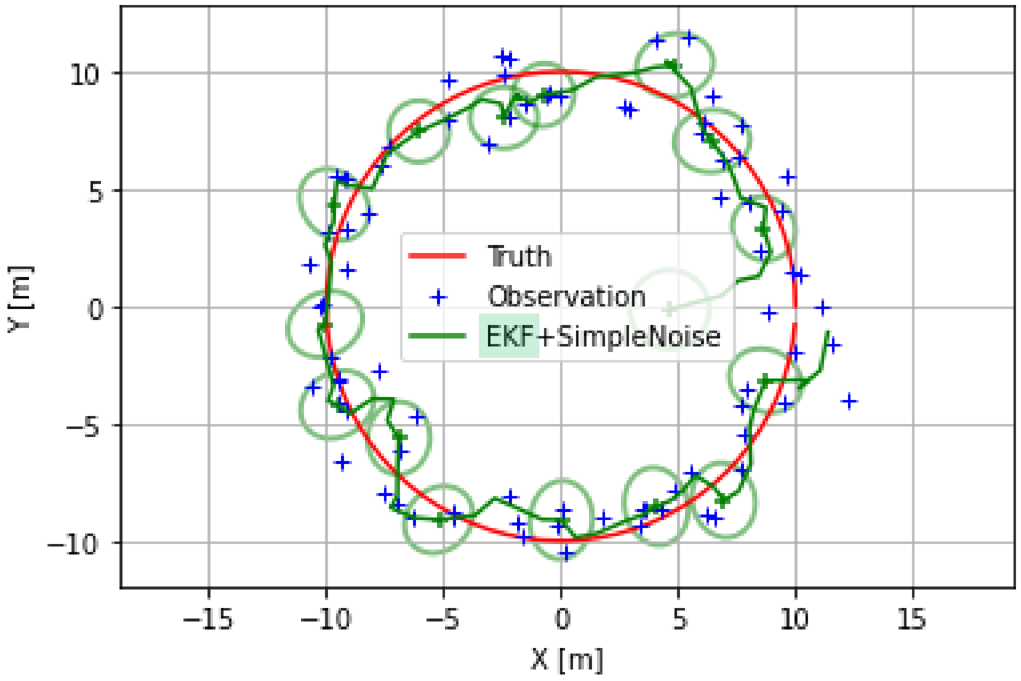
- **Example) 2-D pose tracking with simple transition noise** (`ukf_2d_pose_simple_noise.py`)
  - State variable: $\mathbf{x} = [x, y, \theta, v, w]^\top$
  - State transition function: Constant velocity model (time interval: t)

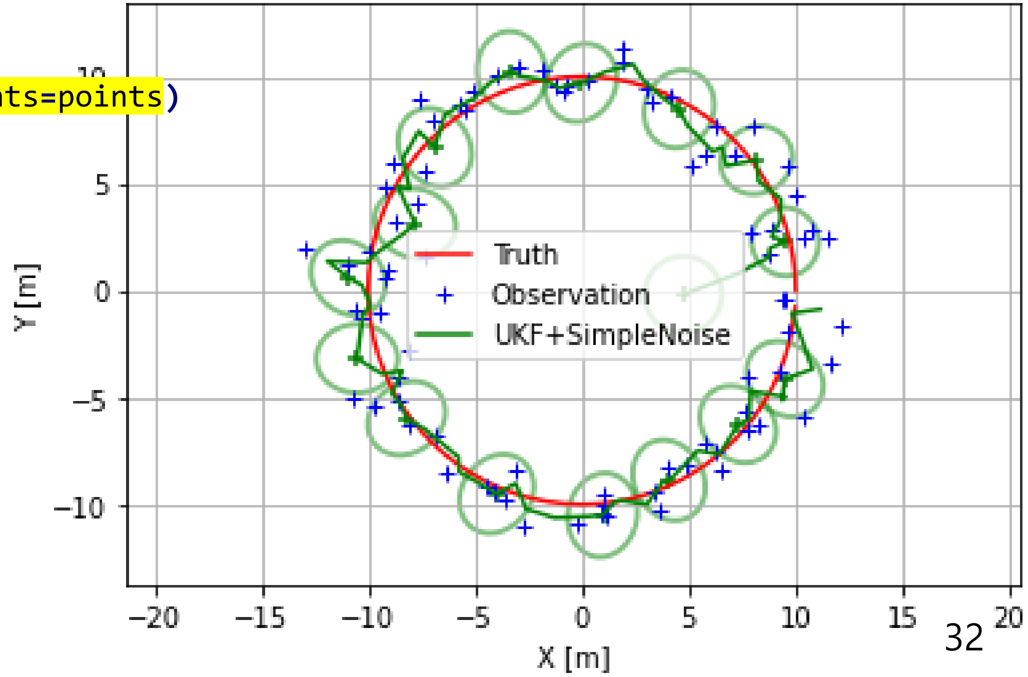
$$\mathbf{x}_{k+1} = f(\mathbf{x}_k; \mathbf{u}_{k+1}) = \begin{bmatrix} x_k + v_k t \cos(\theta_k + w_k t/2) \\ y_k + v_k t \sin(\theta_k + w_k t/2) \\ \theta_k + w_k t \\ v_k \\ w_k \end{bmatrix}$$

    - Control input: $\mathbf{u}_k = [\,]$
  - State transition noise: $\mathrm{Q} = \mathrm{diag}(\sigma_x^2, \sigma_y^2, \sigma_\theta^2, \sigma_v^2, \sigma_w^2)$
  - Observation function: $\mathbf{z} = h(\mathbf{x}) = [x, y]^\top$
    - Observation: $\mathbf{z} = [x_{GPS}, y_{GPS}]^\top$
  - Observation noise: $\mathrm{R} = \mathrm{diag}(\sigma_{GPS}^2, \sigma_{GPS}^2)$



```python
...

def fx(state, dt):
    x, y, theta, v, w = state.flatten()
    vt, wt = v * dt, w * dt
    s, c = np.sin(theta + wt / 2), np.cos(theta + wt / 2)
    return np.array([
        x + vt * c,
        y + vt * s,
        theta + wt,
        v,
        w]) # Note) UKF prefers to use horizontal vectors.

def hx(state):
    x, y, *_ = state.flatten()
    return np.array([x, y]) # Note) UKF prefers to use horizontal vectors.

if __name__ == '__main__':
    # Define experimental configuration
    ...

    # Instantiate UKF for pose (and velocity) tracking
    localizer_name = 'UKF+SimpleNoise'
    points = MerweScaledSigmaPoints(5, alpha=.1, beta=2., kappa=-1)
    localizer = UnscentedKalmanFilter(dim_x=5, dim_z=2, dt=dt, fx=fx, hx=hx, points=points)
    localizer.Q = 0.1 * np.eye(5)
    localizer.R = gps_noise_std * gps_noise_std * np.eye(2)

    truth, state, obser, covar = [], [], [], []
    for t in np.arange(0, t_end, dt):
        # Simulate position observation with additive Gaussian noise
        ...

        # Predict and update the UKF
        ...

        # Record true state, observation, estimated state, and its covariance
        ...

    # Visualize the results
    ...
```



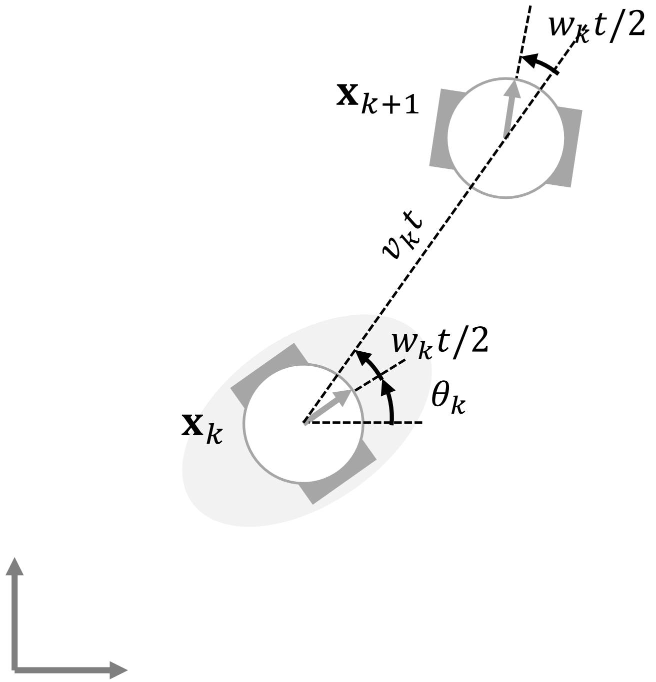


Unscented Kalman Filter

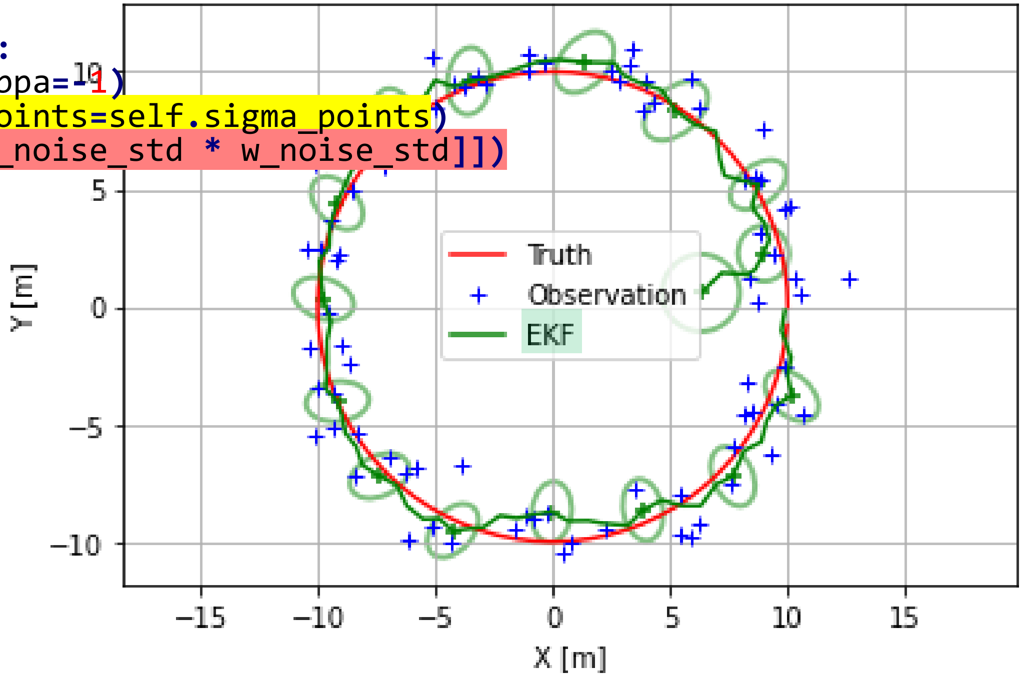
- **Example) 2-D pose tracking** `(ukf_2d_pose.py)`
  - State variable: $\mathbf{x} = [x, y, \theta, v, w]^\top$
  - State transition function: Constant velocity model (time interval: t)

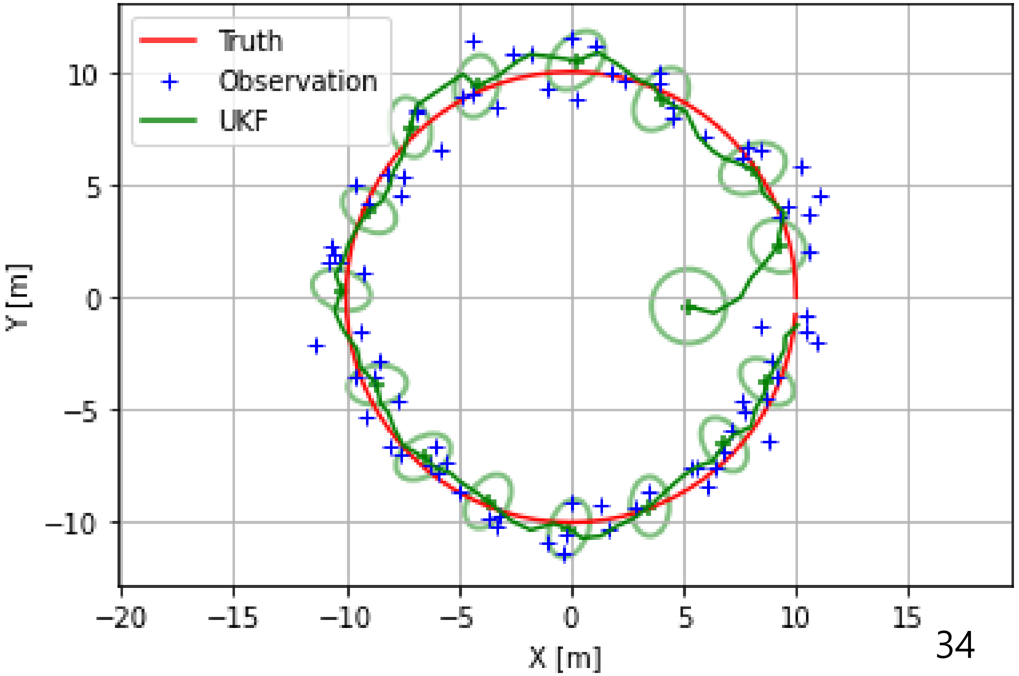
$$\mathbf{x}_{k+1} = f(\mathbf{x}_k; \mathbf{u}_{k+1}) = \begin{bmatrix} x_k + v_k t \cos(\theta_k + w_k t/2) \\ y_k + v_k t \sin(\theta_k + w_k t/2) \\ \theta_k + w_k t \\ v_k \\ w_k \end{bmatrix}$$

    - Control input: $\mathbf{u}_k = [\,]$
  - State transition noise: $\mathrm{Q} = \mathrm{W}\mathrm{M}\mathrm{W}^\top$where $\mathrm{W} = \begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial f}{\partial w} \end{bmatrix}$ and $\mathrm{M} = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_w^2 \end{bmatrix}$
  - Observation function: $\mathbf{z} = h(\mathbf{x}) = [x, y]^\top$
    - Observation: $\mathbf{z} = [x_{GPS}, y_{GPS}]^\top$
  - Observation noise: $\mathrm{R} = \mathrm{diag}(\sigma_{GPS}^2, \sigma_{GPS}^2)$



```python
class UKFLocalizer(UnscentedKalmanFilter):
    def __init__(self, v_noise_std=1, w_noise_std=1, gps_noise_std=1, dt=1):
        self.sigma_points = MerweScaledSigmaPoints(5, alpha=.1, beta=2., kappa=-1)
        super().__init__(dim_x=5, dim_z=2, dt=dt, fx=self.fx, hx=self.hx, points=self.sigma_points)
        self.motion_noise = np.array([[v_noise_std * v_noise_std, 0], [0, w_noise_std * w_noise_std]])
        self.R = gps_noise_std * gps_noise_std * np.eye(2)
        self.dt = dt

    def fx(self, state, dt):
        x, y, theta, v, w = state.flatten()
        vt, wt = v * dt, w * dt
        s, c = np.sin(theta + wt / 2), np.cos(theta + wt / 2)
        return np.array([
            x + vt * c,
            y + vt * s,
            theta + wt,
            v,
            w])

    def hx(self, state):
        x, y, *_ = state.flatten()
        return np.array([x, y])

    def predict(self):
        x, y, theta, v, w = self.x.flatten()
        vt, wt = v * self.dt, w * self.dt
        s, c = np.sin(theta + wt / 2), np.cos(theta + wt / 2)

        # Set the covariance of transition noise
        W = np.array([
            [self.dt * c, -vt * self.dt * s / 2],
            [self.dt * s,  vt * self.dt * c / 2],
            [0, self.dt],
            [1, 0],
            [0, 1]])
        self.Q = W @ self.motion_noise @ W.T

        super().predict()

    def update(self, z):
        super().update(z.flatten())
```





Particle Filter

- Particle filter (a.k.a. sequential Monte Carlo method) is a **nonparametric** filters which **represents its belief $p(\mathbf{x})$ as a set of points** (a.k.a. particles).
  - Why?
    - Particle filter can deal with nonlinear systems.
    - Particle filter can represent its belief $p(\mathbf{x})$ as a multi-modal distribution.
      - Kalman filter and its variants represent their belief $p(\mathbf{x})$ only in an *unimodal* distribution.
        - They are *parametric* filters whose parameters are a *single* mean and covariance.
      - Example) 2-D pose tracking with particle filter and only two sonar sensors

Image: Particle Filters in Action at the UW RSE-lab

Image: Kuptametee et al., "A review of resampling techniques in particle filtering framework", Measurement, 2022

Particle Filter

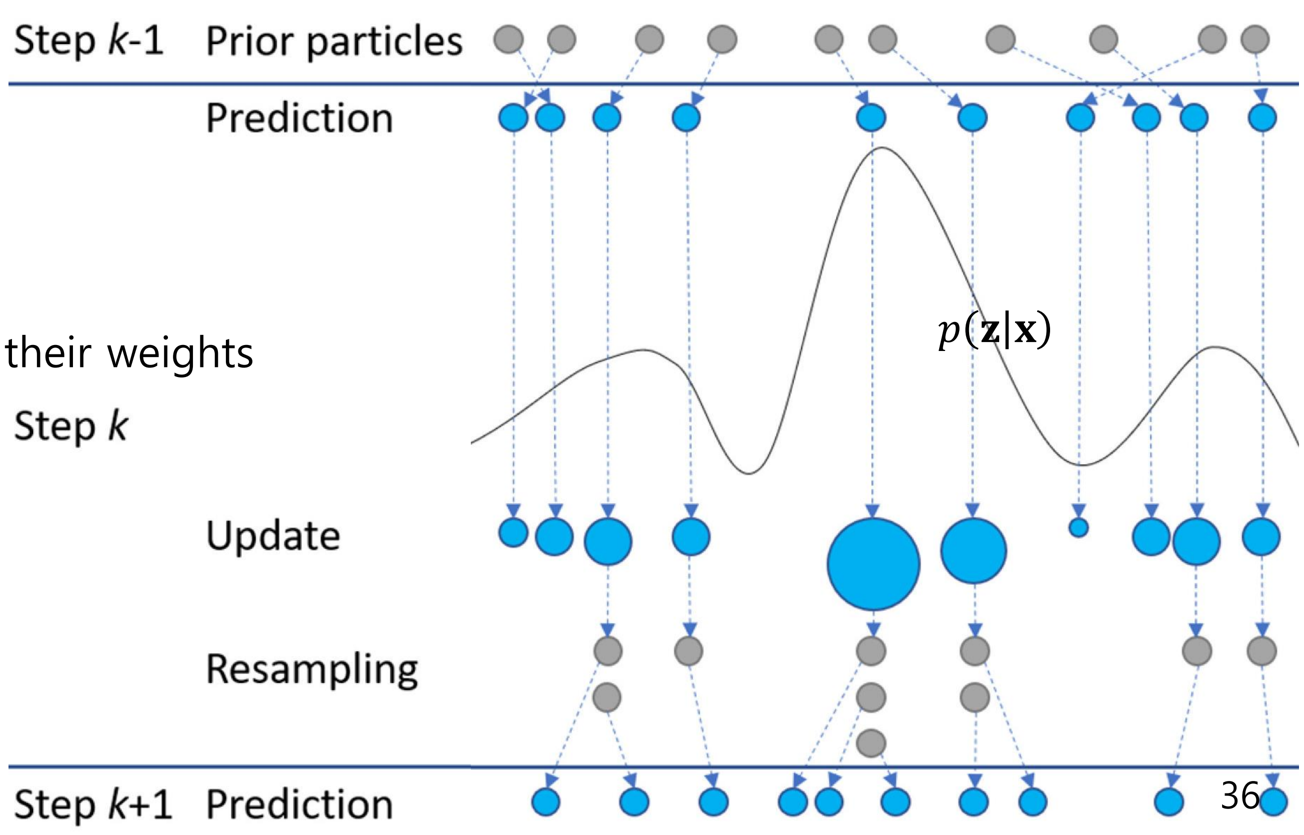
- Particle filter (a.k.a. sequential Monte Carlo method) is a **nonparametric** filters which **represents its belief $p(\mathbf{x})$ as a set of points** (a.k.a. particles).
  - Why? For nonlinear systems and multi-modal belief representation
  - Procedure
    - Preparation) Generate a set of particles randomly
    - Prediction) Predict next state of the particles

      $$\mathbf{x}_{k+1}^{i} = f\left(\mathbf{x}_{k}^{i}; \mathbf{u}_{k+1}\right) \quad \text{for } \; i\text{-th particle}$$

    - Correction #1) Update the weight of particles

      $$w^{i} = p\left(\mathbf{x}^{i} \middle| \mathbf{z}\right) = p\left(\mathbf{z} \middle| \mathbf{x}^{i}\right) p\left(\mathbf{x}^{i}\right) / \, p(\mathbf{z})$$

    - Correction #2) Resample the particles based on their weights

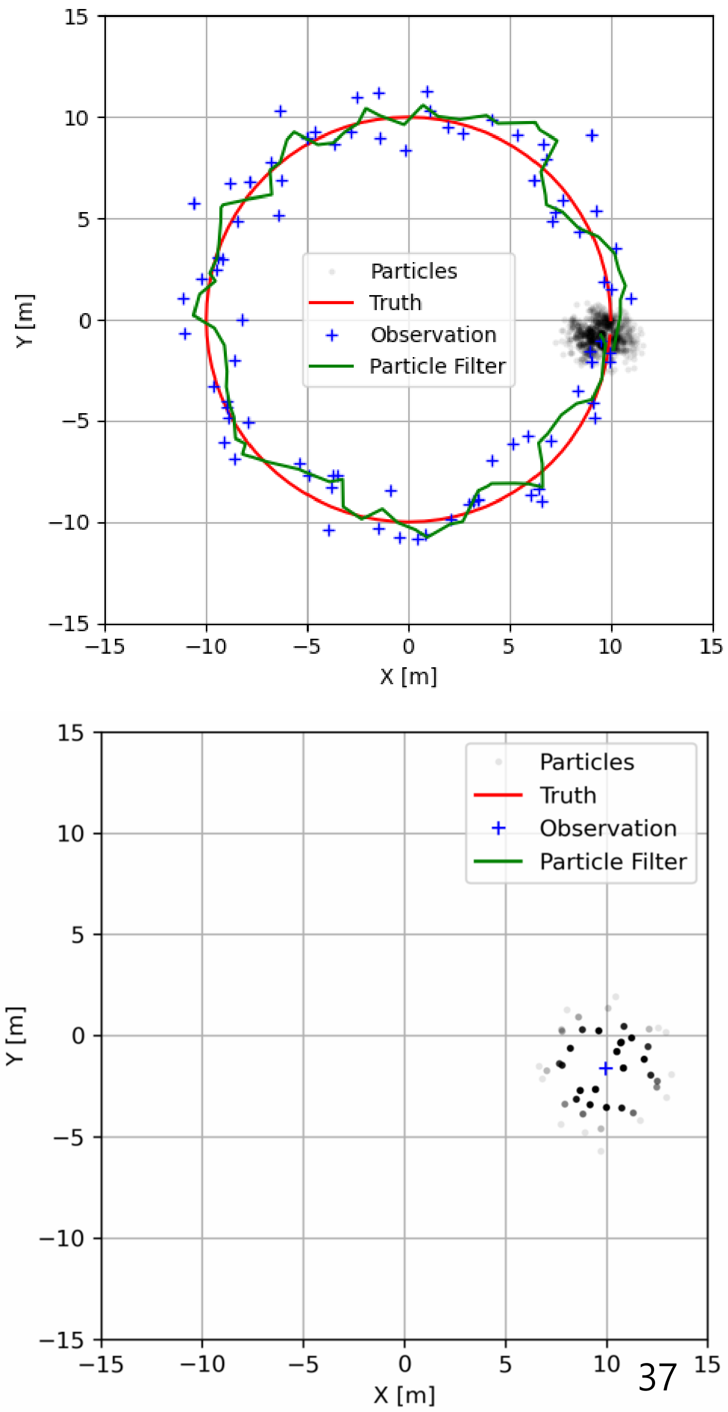


Unscented Kalman Filter

- **Example) 2-D pose tracking** `(pf_2d_pose.py)`
  - State variable: $\mathbf{x} = [x, y, \theta, v, w]^\top$
  - State transition function: Constant velocity model (time interval: t)

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k; \mathbf{u}_k) = \begin{bmatrix} x_k + v_k t \cos(\theta_k + w_k t/2) \\ y_k + v_k t \sin(\theta_k + w_k t/2) \\ \theta_k + w_k t \\ v_k \\ w_k \end{bmatrix}$$

    - Control input: $\mathbf{u}_k = [\,]$
  - State transition noise: $\mathrm{Q} = \mathrm{W}\mathrm{M}\mathrm{W}^\top$where $\mathrm{W} = \begin{bmatrix} \frac{\partial f}{\partial v} & \frac{\partial f}{\partial w} \end{bmatrix}$ and $\mathrm{M} = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_w^2 \end{bmatrix}$
  - Observation function: $\mathbf{z} = h(\mathbf{x}) = [x, y]^\top$
    - Observation: $\mathbf{z} = [x_{GPS}, y_{GPS}]^\top$
  - Observation noise: $\mathrm{R} = \mathrm{diag}(\sigma_{GPS}^2, \sigma_{GPS}^2)$



```python
def neff(weight):
    return 1. / np.sum(np.square(weight))

class PFLocalizer:
    def __init__(self, v_noise_std=1, w_noise_std=1, gps_noise_std=1, ...):
        self.v_noise_std = v_noise_std
        self.w_noise_std = w_noise_std
        self.gps_noise_std = gps_noise_std
        self.dt = dt

        # Spread the initial particles uniformly
        self.pts = np.zeros((N, 5))
        self.pts[:,0] = np.random.uniform(*x_range, size=N)
        self.pts[:,1] = np.random.uniform(*y_range, size=N)
        self.pts[:,2] = np.random.uniform(*theta_range, size=N)
        self.pts[:,3] = np.random.uniform(*v_range, size=N)
        self.pts[:,4] = np.random.uniform(*w_range, size=N)
        self.weight = np.ones(N) / N

    def predict(self):
        # Move the particles
        v_noise = self.v_noise_std * np.random.randn(len(self.pts))
        w_noise = self.w_noise_std * np.random.randn(len(self.pts))
        v_delta = (self.pts[:,3] + v_noise) * self.dt
        w_delta = (self.pts[:,4] + w_noise) * self.dt
        self.pts[:,0] += v_delta * np.cos(self.pts[:,2] + w_delta / 2)
        self.pts[:,1] += v_delta * np.sin(self.pts[:,2] + w_delta / 2)
        self.pts[:,2] += w_delta
        self.pts[:,3] += v_noise
        self.pts[:,4] += w_noise
```

```python
def update(self, z):
        # Update weights of the particles
        d = np.linalg.norm(self.pts[:,0:2] - z.flatten(), axis=1)
        self.weight *= scipy.stats.norm(scale=gps_noise_std).pdf(d)
        self.weight += 1e-10
        self.weight /= sum(self.weight)

        # Resample the particles
        N = len(self.pts)
        if neff(self.weight) < N / 2:
            indices = systematic_resample(self.weight)
            self.pts[:] = self.pts[indices]
            self.weight = np.ones(N) / N

    def get_state(self):
        xy = np.average(self.pts[:,0:2], weights=self.weight, axis=0)
        c = np.average(np.cos(self.pts[:,2]), weights=self.weight)
        s = np.average(np.sin(self.pts[:,2]), weights=self.weight)
        theta = np.arctan2(s, c)
        vw = np.average(self.pts[:,3:5], weights=self.weight, axis=0)
        return np.hstack((xy, theta, vw))
```

Summary

- **Introduction**
  - Simple 1-D Kalman filter = Exponential moving average + inverse-variance weight
- **Kalman Filter**
  - The optimal estimator for linear dynamic systems whose noise is unbiased Gaussian noise.
  - Two steps: Prediction → correction (a.k.a. update)
- **Extended Kalman Filter**
  - A nonlinear version of Kalman filter with linearization (in calculating a covariance matrix)
  - **The five definitions ($\mathbf{x}$, f, h, Q, and R) are important to design and analyze Bayesian filtering.**
  - Note) Euclidean distance vs. **Mahalanobis distance**
- **Unscented Kalman Filter**
  - A nonlinear version of Kalman filter with sigma points (named as unscented transformation)
- **Particle Filter**
  - A nonparametric estimator which represent its belief as a set of particles
  - Why? For nonlinear systems and multi-modal belief representation