Kashirin_Aleksandr_PS2

February 23, 2022

Perception in Robotics

PS2: Localization

Kashirin Aleksandr

Landmark localization

Task A: 1. Write the value for the covariance Q of the noise added to the observation function, knowing that the parameter *bearing_std* is its standard deviation. 2. Write the equation for the covariance R_t of the noise added to the transition function, as explained in class and their corresponding numeric values for the initial robot command $u = [\delta_{rot1}, \delta_{trans}, \delta_{rot2}]^\top = [0, 10, 0]^\top$. Find out the default values of α in *run.py* line 152. 3. Derive the equations for the Jacobians G_t , V_t and H_t , and evaluate them at the initial mean state $_1 = [x, y, \theta]^\top = [180, 50, 0]^\top$ as it is considered in *run.py*.

Answers: 1. Q - is the covariance matrix of the noise added to the observation function. According to the line 81 in *run.py* default value for the *bearing_std* is 0.35. Also, according to the *tools/data.py* line 102,

$$Q = \begin{pmatrix} \sigma^2_{bearing} & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0.35^2 & 0 \\ 0 & 0 \end{pmatrix}$$

2. According to the line 79 in *run.py* : $\sqrt{\text{A}} = [0.05, 0.001, 0.05, 0.01]$, where A is a vector of parameters. Then, according to the line 152 in *run.py* : $\text{A} = [0.05^2, 0.001^2, 0.05^2, 0.01^2]$. Then, in file *tools/task.py* we can find function *get_motion_noise_covariance* that calculates covariance matrix M_t of the noise added to the action space with respect to the formula:

$$M_t = \begin{pmatrix} \alpha_1\delta^2_{rot_1} + \alpha_2\delta^2_{trans} & 0 & 0 \\ 0 & \alpha_3\delta^2_{trans} + \alpha_4(\delta^2_{rot_1} + \delta^2_{rot_2}) & 0 \\ 0 & 0 & \alpha_1\delta^2_{rot_2} + \alpha_2\delta^2_{trans} \end{pmatrix}$$

At the initial step $u = [\delta_{rot_1}, \delta_{trans}, \delta_{rot_2}]^\top = [0, 10, 0]^\top$, covariance noise added tot the transition function would be:

$$M_t = \begin{pmatrix} \alpha_2\delta^2_{trans} & 0 & 0 \\ 0 & \alpha_3\delta^2_{trans} & 0 \\ 0 & 0 & \alpha_2\delta^2_{trans} \end{pmatrix} = \begin{pmatrix} 0.001^2 * 100 & 0 & 0 \\ 0 & 0.05^2 * 100 & 0 \\ 0 & 0 & 0.001^2 * 100 \end{pmatrix} = \begin{pmatrix} 0.0001 & 0 & 0 \\ 0 & 0.25 & 0 \\ 0 & 0 & 0.0001 \end{pmatrix}$$

To calculate R_t we firstly need to derive V_t from task A.3: 3. G_t - is the jacobian that linearizes input gain matrix B when B is nonlinear with respect to x_{t-1}. In our case:

$$G_t = \frac{\partial g(x_{t-1}, u_t, \varepsilon_t)}{\partial x_{t-1}}|_{\mu_{t-1}, \varepsilon_t = 0} = \begin{bmatrix} 1 & 0 & -\delta_{trans}\sin(\theta + \delta_{rot1}) \\ 0 & 1 & \delta_{trans}\cos(\theta + \delta_{rot1}) \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{bmatrix}$$

V_t - is the jacobian that linearizes input gain matrix B when B is nonlinear and noise was accounted in the action space with respect to u_t. In our case:

$$V_t = \frac{\partial g(x_{t-1}, u_t, \varepsilon_t)}{\partial u_t}|_{\mu_{t-1}, \varepsilon_t = 0} = \begin{bmatrix} -\delta_{trans}\sin(\theta + \delta_{rot1}) & \cos(\theta + \delta_{rot1}) & 0 \\ \delta_{trans}\cos(\theta + \delta_{rot1}) & \sin(\theta + \delta_{rot1}) & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 10 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

H_t - is the jacobian that linearizes output gain matrix C when C is nonlinear with respect to x_t. In our case Output measurement (observation) vector is $y_t = [bearing, ID]^T$ but $\sigma_{ID} = 0$ and then state vector is $x_t = [x, y, \theta]^\top$, so the corresponding equation is: $y = Cx$, hence:

$$H_t = \frac{\partial h(x_t)}{\partial x_t}|_{\bar{\mu}_t} = \begin{bmatrix} \frac{m_{i,y} - \bar{\mu}_{t,y}}{(m_{i,x} - \bar{\mu}_{t,x})^2 + (m_{i,y} - \bar{\mu_{t,y}})^2} & \frac{-(m_{i,x} - \bar{\mu}_{t,x})}{(m_{i,x} - \bar{\mu}_{t,x})^2 + (m_{i,y} - \bar{\mu_{t,y}})^2} & -1 \end{bmatrix}$$

For the sake of simplicity, mathematical derivation of R_t is presented down below:

[1]:

```
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt
import sympy
from tools.task import wrap_angle

# Control variables
np.random.seed(42)  # Predefined random seed
labelsize = 15      # Label font size
titlesize = 20      # Title font size
suptitlesize = 25   # Suptitle font size
```

[2]:

```
# Define symbols
t, x_t, y_t, th_t, x_t1, y_t1, th_t1, drot1, drot2, dtrans = sympy.symbols('t␣
  ↪x_t y_t theta_t x_{t-1} y_{t-1} theta_t-1 delta_rot1 delta_rot2 delta_trans')

# Define State Vector
X_t = sympy.Matrix([x_t, y_t, th_t])
X_t1 = sympy.Matrix([x_t1, y_t1, th_t1])

# Define Control Vector
U = sympy.Matrix([drot1, dtrans, drot2])

# Define state transition function
g = sympy.Matrix([[X_t1[0] + U[1]*sympy.cos(X_t1[2] + U[0])],
                  [X_t1[1] + U[1]*sympy.sin(X_t1[2] + U[0])],
                  [X_t1[2] + U[0] + U[2]]])

# Define M_t
a1, a2, a3, a4 = sympy.symbols('alpha_1 alpha_2 alpha_3 alpha_4')
M_t = sympy.Matrix([[a1*U[0]**2 + a2*U[1]**2, 0, 0],
```

```
                     [0, a3*U[1]**2 + a4*(U[0]**2+U[2]**2), 0],
                     [0, 0, a1*U[2]**2 + a2*U[1]**2]])


def jacobian(a, b):
    """ Function returns jacobian between two given vectors
    Args:
        a - Vector a
        b - Vector b

    Returns:
        J - Jacobian of two vectors
    """
    # Get length of vectors
    n = len(a)
    m = len(b)
    # Initialize Jacobian
    J = sympy.zeros(n, m)
    # For each cell in Jacobian
    for i in range(n):
        for j in range(m):
            # Get derivative with related respect
            J[i, j] = a[i].diff(b[j])
    return sympy.simplify(J)

# Print nonlinear transition function
print('Nonlinear transition function g(x_{t-1}, u(t)):')
g
```

```
Nonlinear transition function g(x_{t-1}, u(t)):
```

[2]:
$$\begin{bmatrix}\delta_{trans}\cos\left(\delta_{rot1}+\theta_{t-1}\right)+x_{t-1}\\ \delta_{trans}\sin\left(\delta_{rot1}+\theta_{t-1}\right)+y_{t-1}\\ \delta_{rot1}+\delta_{rot2}+\theta_{t-1}\end{bmatrix}$$

[3]:
```
print('Jacobian with respect to previous state G_t:')
G_t = jacobian(g, X_t1)
G_t
```

```
Jacobian with respect to previous state G_t:
```

[3]:
$$\begin{bmatrix}1 & 0 & -\delta_{trans}\sin\left(\delta_{rot1}+\theta_{t-1}\right)\\ 0 & 1 & \delta_{trans}\cos\left(\delta_{rot1}+\theta_{t-1}\right)\\ 0 & 0 & 1\end{bmatrix}$$

[4]:
```
print('Jacobian with respect to control action V_t:')
V_t = jacobian(g, U)
V_t
```

Jacobian with respect to control action V_t:

[4]:

$$\begin{bmatrix} -\delta_{trans}\sin\left(\delta_{rot1}+\theta_{t-1}\right) & \cos\left(\delta_{rot1}+\theta_{t-1}\right) & 0 \\ \delta_{trans}\cos\left(\delta_{rot1}+\theta_{t-1}\right) & \sin\left(\delta_{rot1}+\theta_{t-1}\right) & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

[5]:
```python
print('Covariance matrix M_t of the noise added to the action space:')
M_t
```

Covariance matrix M_t of the noise added to the action space:

[5]:

$$\begin{bmatrix} \alpha_1\delta_{rot1}^2+\alpha_2\delta_{trans}^2 & 0 & 0 \\ 0 & \alpha_3\delta_{trans}^2+\alpha_4\left(\delta_{rot1}^2+\delta_{rot2}^2\right) & 0 \\ 0 & 0 & \alpha_1\delta_{rot2}^2+\alpha_2\delta_{trans}^2 \end{bmatrix}$$

[6]:
```python
# Calculate R_t = V_t * M_t * V_t^T
print('Covariance matrix R_t of the transformed noise added to the transition␣
 ↪function:')
R_t = V_t * M_t * V_t.T
sympy.simplify(R_t)
```

Covariance matrix R_t of the transformed noise added to the transition function:

[6]:

$$\begin{bmatrix} \delta_{trans}^2\left(\alpha_1\delta_{rot1}^2+\alpha_2\delta_{trans}^2\right)\sin^2\left(\delta_{rot1}+\theta_{t-1}\right)+\left(\alpha_3\delta_{trans}^2+\alpha_4\left(\delta_{rot1}^2+\delta_{rot2}^2\right)\right)\cos^2\left(\delta_{rot1}+\theta_{t-1}\right) & \\ \frac{\left(\alpha_3\delta_{trans}^2+\alpha_4\left(\delta_{rot1}^2+\delta_{rot2}^2\right)-\delta_{trans}^2\left(\alpha_1\delta_{rot1}^2+\alpha_2\delta_{trans}^2\right)\right)\sin\left(2\delta_{rot1}+2\theta_{t-1}\right)}{2} & \delta_{trans}^2\left(\alpha_1\delta_{rot1}^2+\right. \\ -\delta_{trans}\left(\alpha_1\delta_{rot1}^2+\alpha_2\delta_{trans}^2\right)\sin\left(\delta_{rot1}+\theta_{t-1}\right) & \end{bmatrix}$$

Compute numerically R_t at the initial time:

[7]:
```python
#Compute numerically R_t
def compute_num(func, params, num_params):
    """ Function perform numerical computation

    Args:
        func - Given function or Symbolic matrix to calculate
        params - tuple or list of parameters used in the function
        num_params - Corresponded numerical values of the parameters
    Returns:
        func_num - Numerical computed values of the function
    """
    # Lambdify the given function
    func_num = sympy.lambdify(params, func)
    # Calculate numerical values
    func_num = func_num(*num_params)
    return func_num

# Create symbolic list of parameters
params = (a1, a2, a3, a4, drot1, dtrans, drot2, x_t1, y_t1, th_t1)

# Create corresponded numerical list of parameters
```

```
num_params = [0.05**2, 0.001**2, 0.05**2, 0.001**2, 0, 10, 0, 180, 50, 0]

# R_t
R_t_num = compute_num(R_t, params, num_params)
print('Numerical covariance matrix R_t of the transformed noise added to the␣
 ↪transition function:')
R_t_num
```

```
Numerical covariance matrix R_t of the transformed noise added to the transition
function:
```

[7]:
```
array([[ 2.5e-01,  0.0e+00, -0.0e+00],
       [ 0.0e+00,  1.0e-02,  1.0e-03],
       [-0.0e+00,  1.0e-03,  2.0e-04]])
```

Hence, at the initial time:

$$R_t = \begin{bmatrix} 0.25 & 0 & 0 \\ 0 & 0.01 & 0.001 \\ 0 & 0.01 & 0.0002 \end{bmatrix}$$

Task B: Implement EKF and PF-based robot localization using odometry and bearing-only observations to features in a landmark map.

1. EKF is presented in file `filters/ekf.py`
2. PF is presented in file `filters/pf.py`

Videos are included in folder `video` as well!

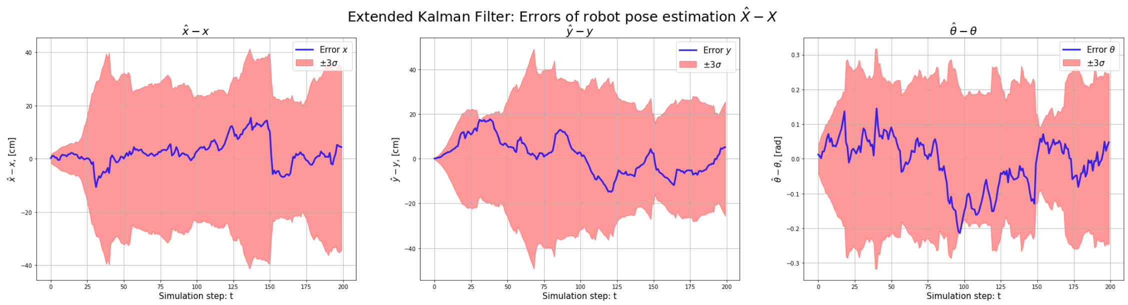
[15]:
```
#![](Task_B/ekf.gif)
#![](Task_B/pf_show_particles.gif)
```

Task C: Create plots of pose error versus time i.e., a plot of $x - x$ vs. t, $-y$ vs t, and $\theta - \theta$ vs. t where $(x, , \theta)$ is the filter estimated pose and (x, y, θ) is the ground-truth actual pose known only to the simulator. Plot the error in blue and in red plot the $\pm 3\sigma$ uncertainty bounds. Your state error should lie within these bounds approximately 99.73% of the time (assuming Gaussian statistics). For the PF, use the sample mean and variance.

[8]:
```
def plot_results(input, output, title):
    ''' Function plots results for tasks C and D
    '''
    # Assign data
    # X mean
    X_hat = output.f.mean_trajectory[:, 0] # Estimated pose X
    X = input.f.real_robot_path[:, 0]       # Real robot position X
    # X std
    X_std = np.sqrt(output.f.covariance_trajectory[0, 0, :])

    # Y mean
    Y_hat = output.f.mean_trajectory[:, 1] # Estimated pose Y
```

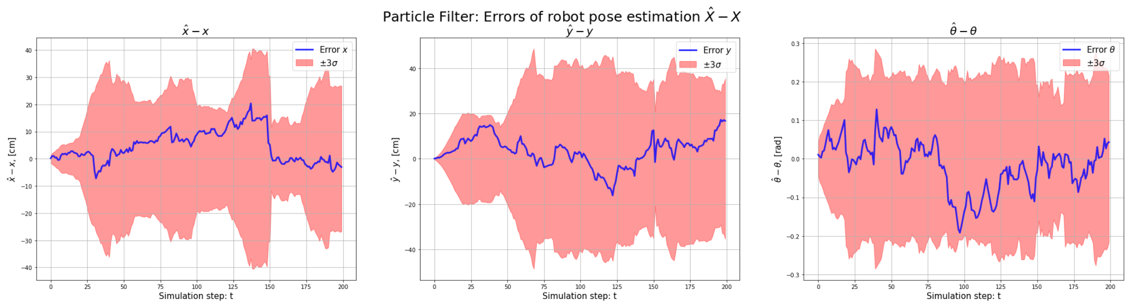
```
    Y = input.f.real_robot_path[:, 1]      # Real robot position Y
    # Y std
    Y_std = np.sqrt(output.f.covariance_trajectory[1, 1, :])

    # Theta
    Theta_hat = output.f.mean_trajectory[:, 2] # Estimated pose Theta
    Theta = input.f.real_robot_path[:, 2]      # Real robot position Theta
    diff_Theta = np.array([wrap_angle(Theta_hat[i]-Theta[i]) for i in␣
↪range(len(Theta))])
    # Theta std
    Theta_std = np.sqrt(output.f.covariance_trajectory[2, 2, :])

    t = np.linspace(0, len(X)-1, len(X))

    plt.figure(figsize=[35,8], facecolor='white')
    plt.suptitle(title, fontsize=suptitlesize)

    #X
    plt.subplot(1, 3, 1)
    plt.plot(t, X_hat-X, linewidth=3, color='blue', label='Error $x$', alpha=0.
↪8) # X Mean
    plt.fill_between(t, 3*X_std, -3*X_std, color='red', alpha=0.4, label=r'$\pm␣
↪3\sigma$')
    plt.title(r'$\hat{x}-x$', fontsize=titlesize)
    plt.legend(loc='upper right', fontsize=labelsize)
    plt.grid(True)
    plt.xlabel('Simulation step: t', fontsize=labelsize)
    plt.ylabel(r'$\hat{x}-x$, [cm]', fontsize=labelsize)

    #Y
    plt.subplot(1, 3, 2)
    plt.plot(t, Y_hat-Y, linewidth=3, color='blue', label='Error $y$', alpha=0.
↪8) # Y Mean
    plt.fill_between(t, 3*Y_std, -3*Y_std, color='red', alpha=0.4, label=r'$\pm␣
↪3\sigma$')
    plt.title(r'$\hat{y}-y$', fontsize=titlesize)
    plt.legend(loc='upper right', fontsize=labelsize)
    plt.grid(True)
    plt.xlabel('Simulation step: t', fontsize=labelsize)
    plt.ylabel(r'$\hat{y}-y$, [cm]', fontsize=labelsize)

    #Theta
    plt.subplot(1, 3, 3)
    plt.plot(t, diff_Theta, linewidth=3, color='blue', label=r'Error $\theta$',␣
↪alpha=0.8) # Theta Mean
```

```
    plt.fill_between(t, 3*Theta_std, -3*Theta_std, color='red', alpha=0.4,␣
 ↪label=r'$\pm 3\sigma$')
    plt.title(r'$\hat{\theta}-\theta$', fontsize=titlesize)
    plt.legend(loc='upper right', fontsize=labelsize)
    plt.grid(True)
    plt.xlabel('Simulation step: t', fontsize=labelsize)
    plt.ylabel(r'$\hat{\theta}-\theta$, [rad]', fontsize=labelsize)
    plt.show()

# Load data
ekf_input = np.load('ekf_out/input_data.npy')
ekf_output = np.load('ekf_out/output_data.npy')
# Define title
ekf_title = r'Extended Kalman Filter: Errors of robot pose estimation $\hat{X}␣
 ↪- X$'

plot_results(ekf_input, ekf_output, ekf_title)
```



Conclusion: As was expected, an estimation error lays within $\pm 3\sigma$ interval.

[9]:
```
# Load data
pf_input = np.load('pf_out/input_data.npy')
pf_output = np.load('pf_out/output_data.npy')
# Define title
pf_title = r'Particle Filter: Errors of robot pose estimation $\hat{X} - X$'

plot_results(pf_input, pf_output, pf_title)
```



Conclusion: As was expected, an estimation error lays within $\pm 3\sigma$ interval.

Task D: Once your filters are implemented, please investigate some properties of them. How do they behave * as the sensor or motion noise go toward zero? (Please provide plots and explanation) * as the number of particles decrease? * if the filter noise parameters underestimate or overestimate the true noise parameters? Please clarify what underestimation and overestimation of noise is?

(Items 1 and 3 are for EKF and "the noise" refers to both observation and control noises)

1. How does EKF behaves when motion noise R_t goes towards zero?

Let us assume following sets of motion noise constants in order to investigate properties of EKF:

Set 1: $ $= 0.8$ [0.05, 0.001, 0.05, 0.01] $

Set 2: $ $= 0.2$ [0.05, 0.001, 0.05, 0.01] $

Set 3: $ $= 0$ [0.05, 0.001, 0.05, 0.01] $

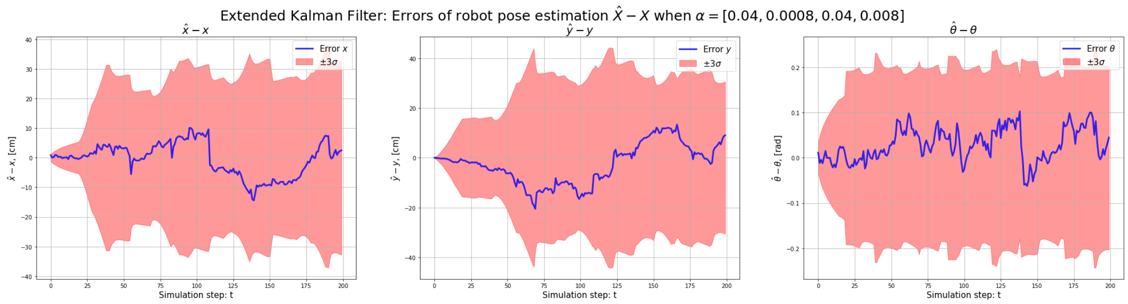
We can obtain data with following commands:

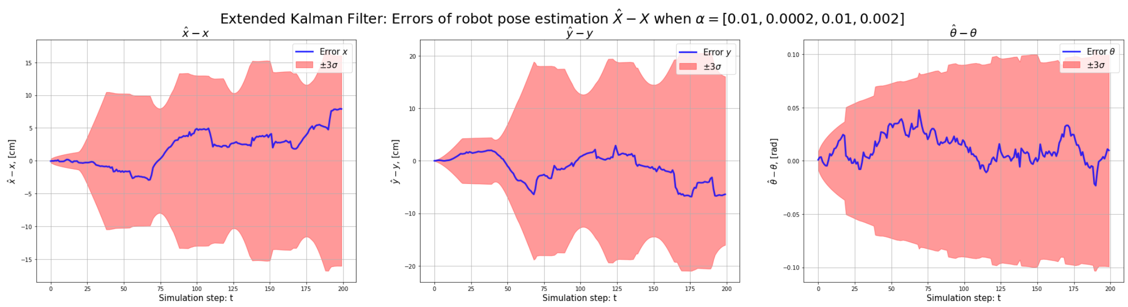
```
python run.py --animate -s -f ekf -n 200 -a 0.04 0.0008 0.04 0.008  -o Task_D/Experiment_1/Moti
python run.py --animate -s -f ekf -n 200 -a 0.01 0.0002 0.01 0.002  -o Task_D/Experiment_1/Moti
python run.py --animate -s -f ekf -n 200 -a 0.0 0.0 0.0 0.0  -o Task_D/Experiment_1/Motion_Nois
```

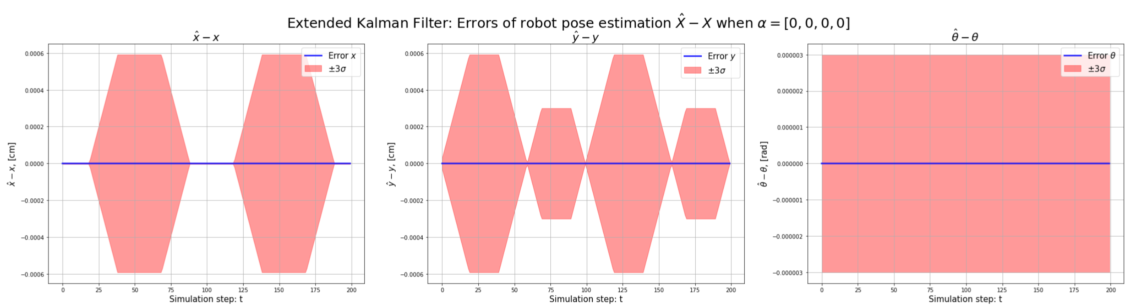
Let us plot graphs in order to investigate properties of the filter.

```
[10]: # Set_1 Load data
ex_1_mn_set_1_input = np.load('Task_D/Experiment_1/Motion_Noise/set_1/
 ↪input_data.npy')
ex_1_mn_set_1_output = np.load('Task_D/Experiment_1/Motion_Noise/set_1/
 ↪output_data.npy')
# Define title
ex_1_mn_set_1_title = r'Extended Kalman Filter: Errors of robot pose estimation␣
 ↪$\hat{X} - X$ when $\alpha=[0.04, 0.0008, 0.04, 0.008]$'

plot_results(ex_1_mn_set_1_input, ex_1_mn_set_1_output, ex_1_mn_set_1_title)

# Set_2 Load data
ex_1_mn_set_2_input = np.load('Task_D/Experiment_1/Motion_Noise/set_2/
 ↪input_data.npy')
ex_1_mn_set_2_output = np.load('Task_D/Experiment_1/Motion_Noise/set_2/
 ↪output_data.npy')
# Define title
ex_1_mn_set_2_title = r'Extended Kalman Filter: Errors of robot pose estimation␣
 ↪$\hat{X} - X$ when $\alpha=[0.01, 0.0002, 0.01, 0.002]$'

plot_results(ex_1_mn_set_2_input, ex_1_mn_set_2_output, ex_1_mn_set_2_title)
```

```python
# Set_3 Load data
ex_1_mn_set_3_input = np.load('Task_D/Experiment_1/Motion_Noise/set_3/
↪input_data.npy')
ex_1_mn_set_3_output = np.load('Task_D/Experiment_1/Motion_Noise/set_3/
↪output_data.npy')
# Define title
ex_1_mn_set_3_title = r'Extended Kalman Filter: Errors of robot pose estimation␣
↪$\hat{X} - X$ when $\alpha=[0, 0, 0, 0]$'

plot_results(ex_1_mn_set_3_input, ex_1_mn_set_3_output, ex_1_mn_set_3_title)
```







Conclusion: As we can see, the $\pm 3\sigma$ interval becomes less and hence the uncertainty in position estimation decreases.

2. How does EKF behaves when motion noise Q_t goes towards zero?

Let us assume following sets of measurement noise constants in order to investigate properties of EKF:

Set 1: $ Q_1 = 0.15^2 $

Set 2: $ Q_2 = 0.08^2 $

Set 3: $ Q_3 = 0 $

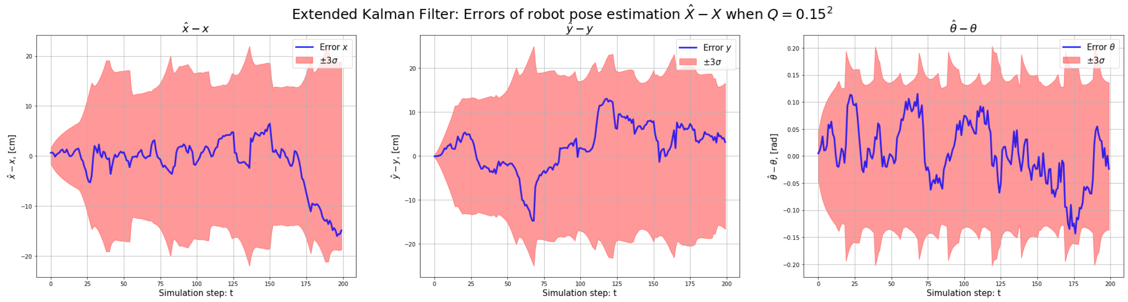
We can obtain data with following commands:

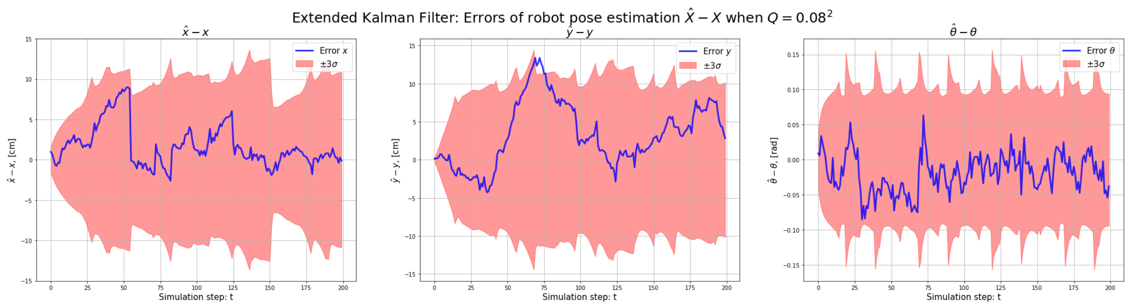
```
python run.py --animate -s -f ekf -n 200 -b 0.15 -o Task_D/Experiment_1/Obsv_Noise/set_1
python run.py --animate -s -f ekf -n 200 -b 0.08 -o Task_D/Experiment_1/Obsv_Noise/set_2
python run.py --animate -s -f ekf -n 200 -b 0.0 -o Task_D/Experiment_1/Obsv_Noise/set_3
```

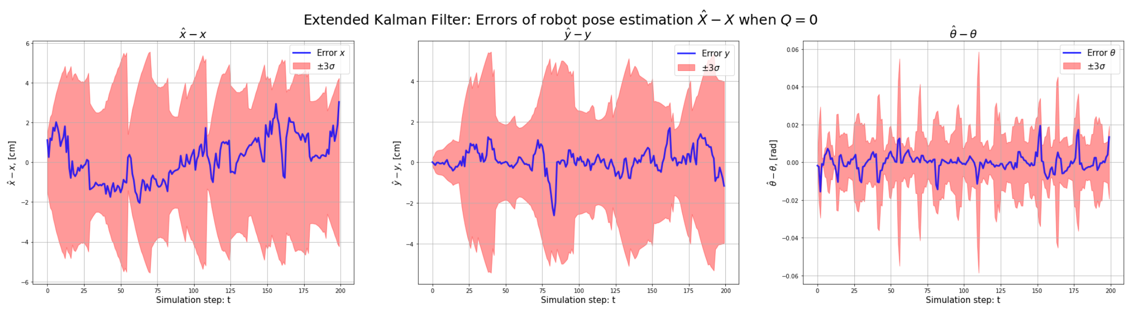
Let us plot graphs in order to investigate properties of the filter.

[11]:
```
# Set_1 Load data
ex_1_on_set_1_input = np.load('Task_D/Experiment_1/Obsv_Noise/set_1/input_data.
 ↪npy')
ex_1_on_set_1_output = np.load('Task_D/Experiment_1/Obsv_Noise/set_1/
 ↪output_data.npy')
# Define title
ex_1_on_set_1_title = r'Extended Kalman Filter: Errors of robot pose estimation␣
 ↪$\hat{X} - X$ when $Q=0.15^2$'

plot_results(ex_1_on_set_1_input, ex_1_on_set_1_output, ex_1_on_set_1_title)

# Set_2 Load data
ex_1_on_set_2_input = np.load('Task_D/Experiment_1/Obsv_Noise/set_2/input_data.
 ↪npy')
ex_1_on_set_2_output = np.load('Task_D/Experiment_1/Obsv_Noise/set_2/
 ↪output_data.npy')
# Define title
ex_1_on_set_2_title = r'Extended Kalman Filter: Errors of robot pose estimation␣
 ↪$\hat{X} - X$ when $Q=0.08^2$'

plot_results(ex_1_on_set_2_input, ex_1_on_set_2_output, ex_1_on_set_2_title)

# Set_3 Load data
ex_1_on_set_3_input = np.load('Task_D/Experiment_1/Obsv_Noise/set_3/input_data.
 ↪npy')
ex_1_on_set_3_output = np.load('Task_D/Experiment_1/Obsv_Noise/set_3/
 ↪output_data.npy')
# Define title
ex_1_on_set_3_title = r'Extended Kalman Filter: Errors of robot pose estimation␣
 ↪$\hat{X} - X$ when $Q=0$'
```

```
plot_results(ex_1_on_set_3_input, ex_1_on_set_3_output, ex_1_on_set_3_title)
```







Conclusion: Here we can also see that uncertainty decreases, but since we still have motion noise than can not absolutely rely on measurements and eleminate estimation errors.

3. How does PF behaves when amount of particles is decreased ?

Let us assume following sets of number of particles in order to investigate properties of PF:

Set 1: `num_particles$` = 50$

Set 2: `num_particles$` = 10$

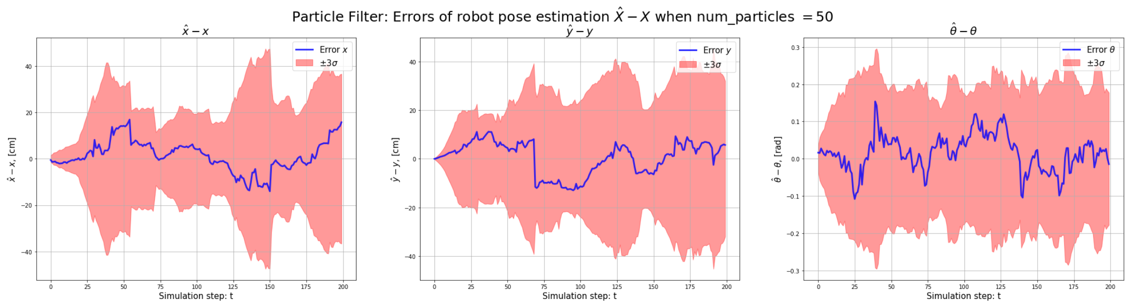
We can obtain data with following commands:

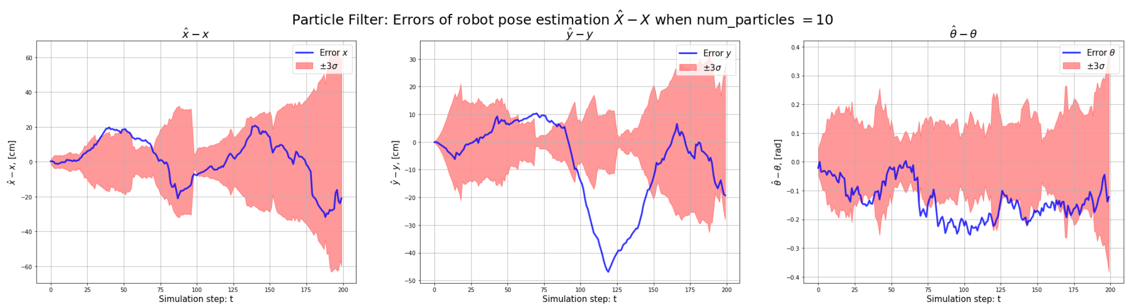
```
python run.py --animate -s -f pf -n 200 --num-particles 50 -o Task_D/Experiment_2/set_1
```

python run.py --animate -s -f pf -n 200 --num-particles 10 -o Task_D/Experiment_2/set_2

Let us plot graphs in order to investigate properties of the filter.

[12]:
```python
# Set_1 Load data
ex_2_set_1_input = np.load('Task_D/Experiment_2/set_1/input_data.npy')
ex_2_set_1_output = np.load('Task_D/Experiment_2/set_1/output_data.npy')
# Define title
ex_2_set_1_title = r'Particle Filter: Errors of robot pose estimation $\hat{X}␣
↪- X$ when num_particles $= 50$'

plot_results(ex_2_set_1_input, ex_2_set_1_output, ex_2_set_1_title)

# Set_2 Load data
ex_2_set_2_input = np.load('Task_D/Experiment_2/set_2/input_data.npy')
ex_2_set_2_output = np.load('Task_D/Experiment_2/set_2/output_data.npy')
# Define title
ex_2_set_2_title = r'Particle Filter: Errors of robot pose estimation $\hat{X}␣
↪- X$ when num_particles $= 10$'

plot_results(ex_2_set_2_input, ex_2_set_2_output, ex_2_set_2_title)
```





Conclusion: As we can see, when number of particles is extremely decreased, we can not provide a good estimation, since the distribution of particles tells us wrong localaziton information.

4. How does EKF behaves when we underestimate or overestimate filter parameters Q and R?

Let us assume following sets in order to investigate properties of EKF:

Motion noise:

Set 1: \$ = 2 [0.05, 0.001, 0.05, 0.01] \$ - Overestimated motion noise

Set 2: \$ = 0.5 [0.05, 0.001, 0.05, 0.01] \$ - Underestimated motion noise

Observation noise:

Set 1: \$ Q__1 = 0.6^2 \$ - Overestimated measurement noise

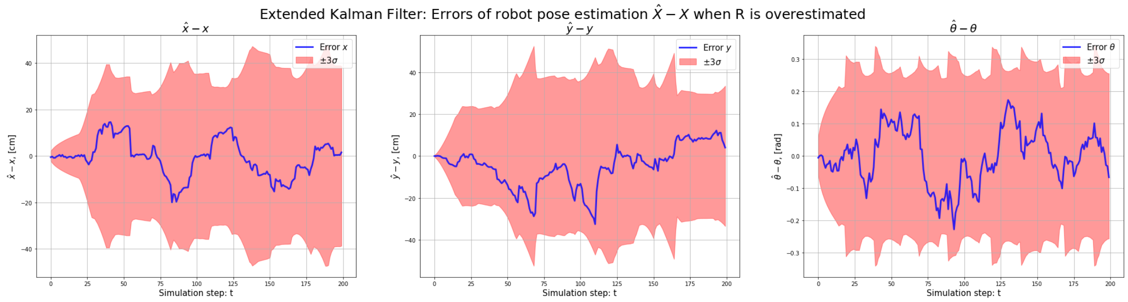
Set 2: \$ Q__2 = 0.1^2 \$ - Underestimated measurement noise

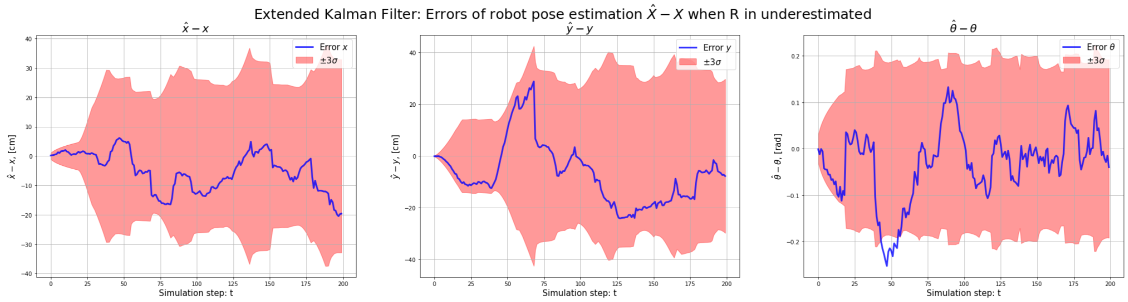
We can obtain data with following commands:

```
python run.py --animate -s -f ekf -n 200 -o Task_D/Experiment_3/Motion_Noise/set_1
python run.py --animate -s -f ekf -n 200 -o Task_D/Experiment_3/Motion_Noise/set_2
python run.py --animate -s -f ekf -n 200 -o Task_D/Experiment_3/Obsv_Noise/set_1
python run.py --animate -s -f ekf -n 200 -o Task_D/Experiment_3/Obsv_Noise/set_2
```

Let us plot graphs in order to investigate properties of the filter.

[13]:
```
# Set_1 Load data
ex_3_mn_set_1_input = np.load('Task_D/Experiment_3/Motion_Noise/set_1/
 ↪input_data.npy')
ex_3_mn_set_1_output = np.load('Task_D/Experiment_3/Motion_Noise/set_1/
 ↪output_data.npy')
# Define title
ex_3_mn_set_1_title = r'Extended Kalman Filter: Errors of robot pose estimation␣
 ↪$\hat{X} - X$ when R is overestimated'

plot_results(ex_3_mn_set_1_input, ex_3_mn_set_1_output, ex_3_mn_set_1_title)

# Set_2 Load data
ex_3_mn_set_2_input = np.load('Task_D/Experiment_3/Motion_Noise/set_2/
 ↪input_data.npy')
ex_3_mn_set_2_output = np.load('Task_D/Experiment_3/Motion_Noise/set_2/
 ↪output_data.npy')
# Define title
ex_3_mn_set_2_title = r'Extended Kalman Filter: Errors of robot pose estimation␣
 ↪$\hat{X} - X$ when R in underestimated'

plot_results(ex_3_mn_set_2_input, ex_3_mn_set_2_output, ex_3_mn_set_2_title)
```

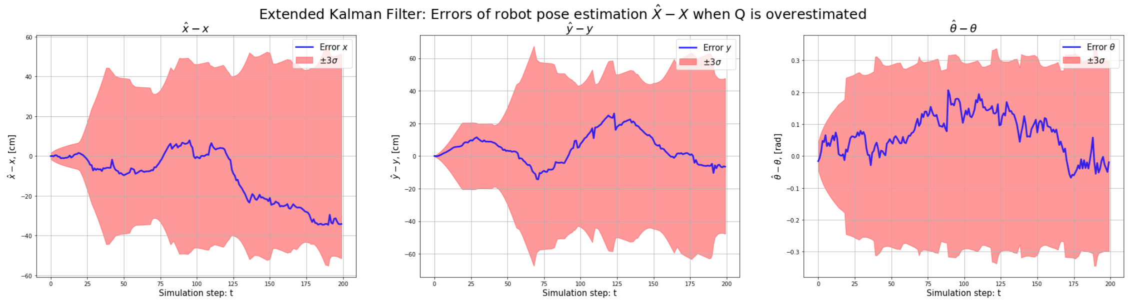


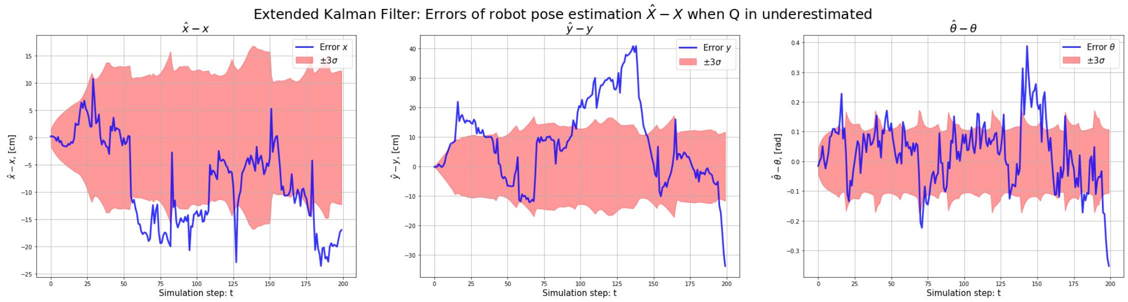


Conclusion: Underestimation or overestimation of the motion noise does not much affect the perfomance of EKF. In case of overestimation we have a bit higher convergence of Filter perfomance and higher sigma gap of possible estimation values.

[14]:
```python
# Set_1 Load data
ex_3_on_set_1_input = np.load('Task_D/Experiment_3/Obsv_Noise/set_1/input_data.npy')
ex_3_on_set_1_output = np.load('Task_D/Experiment_3/Obsv_Noise/set_1/output_data.npy')
# Define title
ex_3_on_set_1_title = r'Extended Kalman Filter: Errors of robot pose estimation $\hat{X} - X$ when Q is overestimated'

plot_results(ex_3_on_set_1_input, ex_3_on_set_1_output, ex_3_on_set_1_title)

# Set_2 Load data
ex_3_on_set_2_input = np.load('Task_D/Experiment_3/Obsv_Noise/set_2/input_data.npy')
ex_3_on_set_2_output = np.load('Task_D/Experiment_3/Obsv_Noise/set_2/output_data.npy')
# Define title
ex_3_on_set_2_title = r'Extended Kalman Filter: Errors of robot pose estimation $\hat{X} - X$ when Q in underestimated'
```

```
plot_results(ex_3_on_set_2_input, ex_3_on_set_2_output, ex_3_on_set_2_title)
```





Conclusion: Here, in the case of overestimation, we got the results of increased σ gap of possible estimation values. In case of underestimation, we have obtained incorrect estimation results. Hence, it is better to overestimate than underestimate the noise parameters.