A. (15 pts) Before implementing anything, take a look at the code and answer the following questions. 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. To find out which is the value of `bearing_std` you should look at the default parameters passed to `run.py` lines 44 - 121. 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. Then derive the equations for the Jacobians G_t, V_t and H_t, and evaluate them at the initial mean state $\mu_1 = [x, y, \theta]^\top = [180, 50, 0]^\top$ as it is considered in `run.py`. (It is not requested to evaluate observation Jacobians.)

1. $Q = \begin{pmatrix} bearing\_std^2 & 0 \\ 0 & 0 \end{pmatrix}$ = - toolS.py (102-Line)

$= \begin{pmatrix} 0.35^2 & 0 \\ 0 & 0 \end{pmatrix}$ - bearing_std = 0.35 - run.py (80-85 Lines)

2. $\sqrt{A} = [0.05, 0.001, 0.05, 0.01]$ - run.py (79 Line)

$A = [0.05^2, 0.001^2, 0.05^2, 0.01^2]$ - vector of alphas - run.py (152 line)

Formula for M_t from get_motion_noise_covariance() task.py (Line 136)

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

For initial step $u = [\delta_{rot_1}, \delta_{trans}, \delta_{rot_2}]^T = [0, 10, 0]^T$

$$M_t = \begin{pmatrix} \alpha_2 \delta_{trans}^2 & 0 & 0 \\ 0 & \alpha_3 \delta_{trans}^2 & 0 \\ 0 & 0 & \alpha_3 \delta_{trans} \end{pmatrix} = \begin{pmatrix} 0.001^2 \cdot 100 & 0 & 0 \\ 0 & 0.05^2 \cdot 100 & 0 \\ 0 & 0 & 0.001^2 \cdot 100 \end{pmatrix} =$$

$$= \begin{pmatrix} 0.0001 & 0 & 0 \\ 0 & 0.25 & 0 \\ 0 & 0 & 0.0001 \end{pmatrix}$$

For R_t calculation we need V_t matrix

$$V_t = \frac{\partial g(x_{t-1}, u_t, \xi_t)}{\partial u_t} \Big|_{\mu_{t-1}, \xi_t = 0} =$$

$$= \begin{pmatrix} -\delta_{trans} \sin(\theta + \delta_{rot_1}) & \cos(\theta + \delta_{rot_1}) & 0 \\ \delta_{trans} \cos(\theta + \delta_{rot_1}) & \sin(\theta + \delta_{rot_1}) & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \mu_0 = [x, y, \theta]^T = \\ = [180, 50, 0] \end{pmatrix}$$

$$= \begin{pmatrix} -10 \cdot \sin(0+0) & \cos(0+0) & 0 \\ 10 \cdot \cos(0+0) & \sin(0+0) & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 10 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

$$R_t = V_t \cdot M_t \cdot V_t^T = \begin{pmatrix} 0 & 1 & 0 \\ 10 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0.0001 & 0 & 0 \\ 0 & 0.25 & 0 \\ 0 & 0 & 0.0001 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 10 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}^T$$

In [1]:
```
import numpy as np
```

In [2]:
```
Vt = np.array([[0,  1, 0],
               [10, 0, 0],
               [1,  0, 1]])

Mt = np.array([[0.0001, 0,    0],
               [0,      0.25, 0],
               [0,      0,    0.0001]])
```

In [4]:
```
Rt = Vt @ Mt @ Vt.T
Rt
```

Out[4]:
```
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]])
```

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

$$3.\ G_t = \frac{\partial g(x_{t-1}, u_t, \xi)}{\partial x_{t-1}}\Big|_{\mu_{t-1}, \xi_t = 0} = \begin{pmatrix} 1 & 0 & -\delta_{trans}\sin(\theta+\delta_{rot1}) \\ 0 & 1 & \delta_{trans}\cos(\theta+\delta_{rot1}) \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & -10\cdot\sin(0+0) \\ 0 & 1 & 10\cdot\cos(0+0) \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{pmatrix}$$

$$V_t = \frac{\partial g(x_{t-1}, u_t, \xi_t)}{\partial u_t}\Big|_{\mu_{t-1}, \xi_t = 0} =$$

$$= \begin{pmatrix} -\delta_{trans}\sin(\theta+\delta_{rot1}) & \cos(\theta+\delta_{rot1}) & 0 \\ \delta_{trans}\cos(\theta+\delta_{rot1}) & \sin(\theta+\delta_{rot1}) & 0 \\ 1 & 0 & 1 \end{pmatrix} = \left(\mu_0 = [x, y, \theta]^T = [150, 50, 0]\right) =$$

$$= \begin{pmatrix} -10\cdot\sin(0+0) & \cos(0+0) & 0 \\ 10\cdot\cos(0+0) & \sin(0+0) & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 10 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

$$H_t = \frac{\partial h(x_t)}{\partial x_t}\Big|_{\bar{\mu}_t} =$$

$$= \left(\frac{m_{i,y} - \bar{\mu}_{t,y}}{(m_{i,x} - \bar{\mu}_{t,x})^2 + (m_{i,y} - \bar{\mu}_{t,y})^2} \quad \frac{-m_{i,x} + \bar{\mu}_{t,x}}{(m_{i,x} - \bar{\mu}_{t,x})^2 + (m_{i,y} - \bar{\mu}_{t,y})^2} \quad -1 \right)$$

B. (50 pts) Implement EKF and PF-based robot localization using odometry and bearing-only observations to features in a landmark map. Remember to run the evaluation command to properly use the common created data file `evaluation-input.npy`.

1. For the EKF plot the filter estimated mean position and 3-sigma covariance ellipsoid overlaid on top of the simulation figure at every time step. If your filter is working correctly, the robot should lie within the 3-sigma ellipse 98.89% of the time.
2. For the PF, plot the sample distribution every time step, it should be centered on the robot's true position.

Include videos in your submission for the EKF and PF under evaluation conditions and using the corresponding input parameter -m.

1. Video and data for EKF is in ekf_out folder
2. Video and data for PF is in pf_out folder

C. (20 pts) Create plots of pose error versus time i.e., a plot of $\hat{x} - x$ vs. t, $\hat{y} - y$ vs t, and $\hat{\theta} - \theta$ vs. t where $(\hat{x}, \hat{y}, \hat{\theta})$ 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.

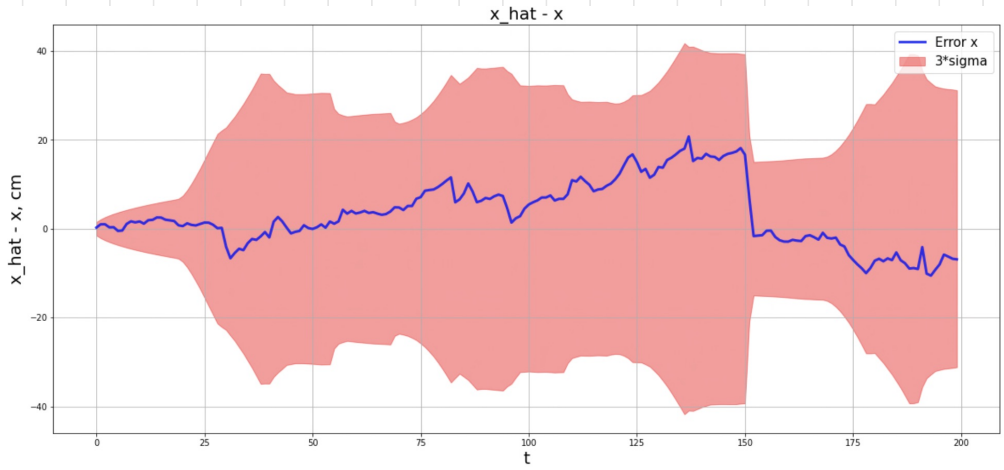
Function for plotting results of experiments

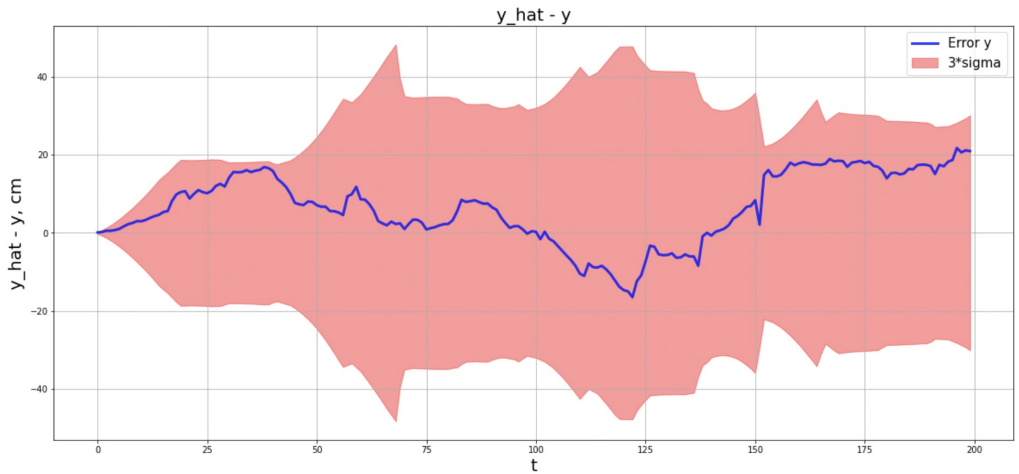
```
import numpy as np
import matplotlib.pyplot as plt

def wrap_angle(angle):
    pi2 = 2 * np.pi

    while angle < -np.pi:
        angle += pi2

    while angle >= np.pi:
        angle -= pi2

    return angle

def plot(var, inp, outp):
    dic = {'x':1, 'y':2, 'th':3}
    v_hat = outp.f.mean_trajectory[:, dic[var] - 1]
    v = inp.f.real_robot_path[:, dic[var] - 1]
    v_std = np.sqrt(outp.f.covariance_trajectory[dic[var] - 1, dic[var] - 1, :])
    if var == 'th':
        v = np.array([wrap_angle(v_hat[i]-v[i]) for i in range(len(v))])
        v_hat = v + v

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

    plt.subplot(3, 1, dic[var])
    plt.plot(t, v_hat - v, linewidth=3, color='blue', label='Error ' + var, alpha=0.8)
    plt.fill_between(t, 3 * v_std, -3 * v_std, color='red', alpha=0.4, label='3*sigma')
    plt.title(var + '_hat - ' + var, fontsize = 20)
    plt.legend(fontsize = 15)
    plt.grid(True)
    plt.xlabel('t', fontsize = 20)
    plt.ylabel(var + '_hat - ' + var + ', cm', fontsize = 20)
```

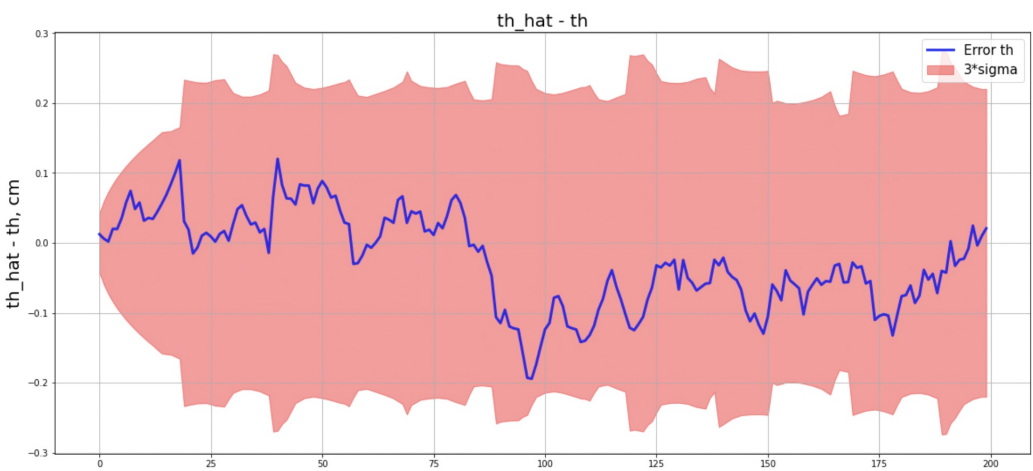
```
ekf_input = np.load('/Users/vladimirberman/Documents/Perception-in-Robotics-2023/PS2/ps2_code/results/ekf/input_data
ekf_output = np.load('/Users/vladimirberman/Documents/Perception-in-Robotics-2023/PS2/ps2_code/results/ekf/output_da
```

```
plt.figure(figsize=[20,30])
plot('x', ekf_input, ekf_output)
plot('y', ekf_input, ekf_output)
plot('th', ekf_input, ekf_output)
```

EKF


x_hat - x
Error x
3*sigma
x_hat - x, cm
t

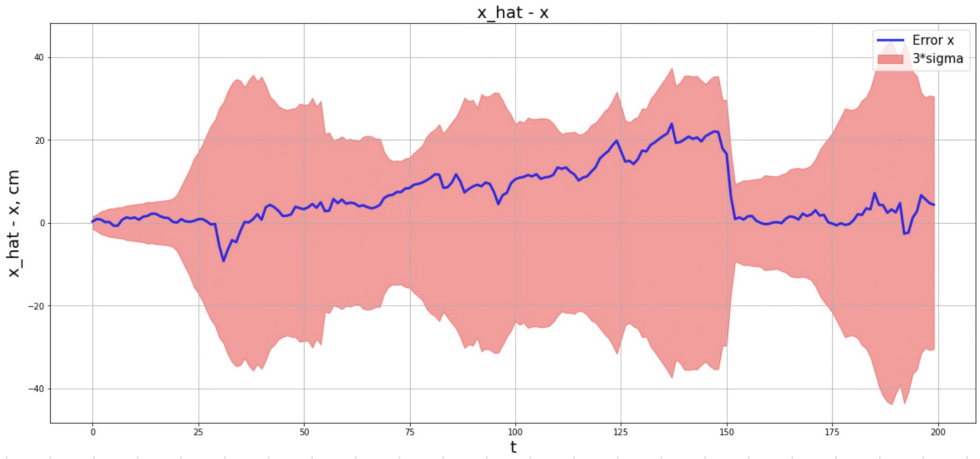

y_hat - y
Error y
3*sigma
y_hat - y, cm
t

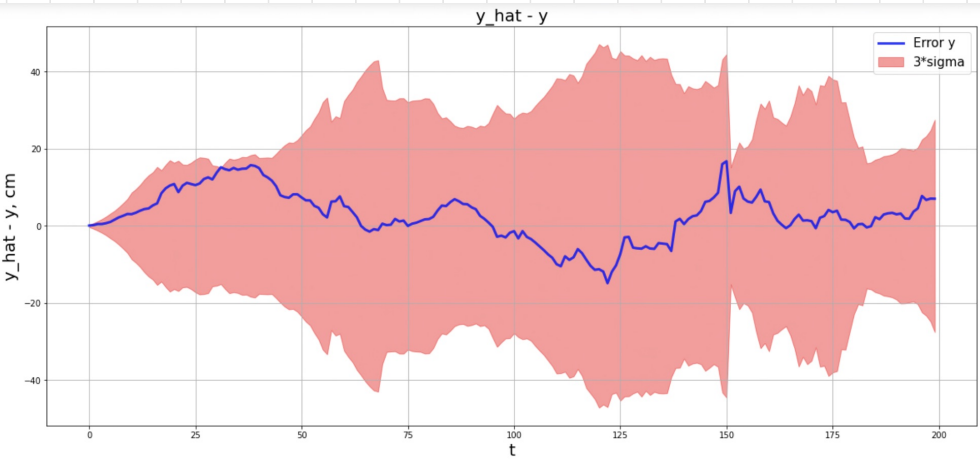

th_hat - th
Error th
3*sigma
th_hat - th, cm
t

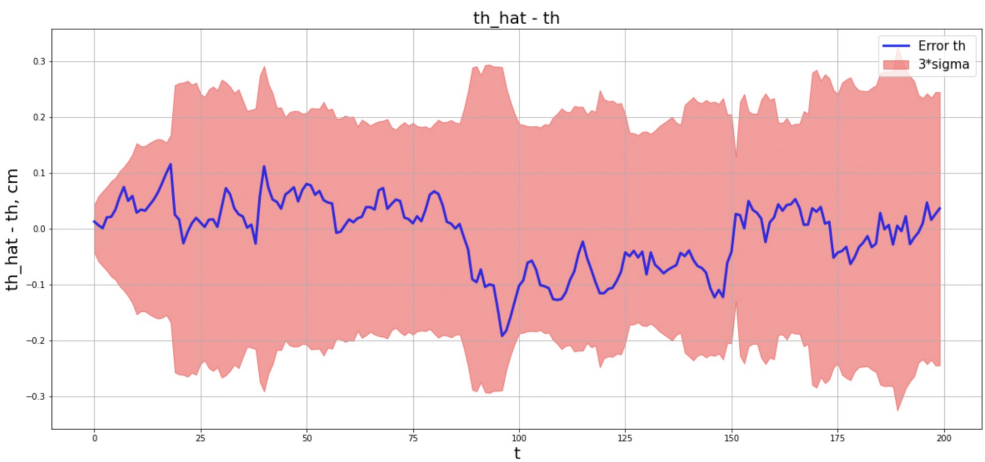
In [71]:
```
pf_input = np.load('/Users/vladimirberman/Documents/Perception-in-Robotics-2023/PS2/ps2_code/results/pf/input_data.n
pf_output = np.load('/Users/vladimirberman/Documents/Perception-in-Robotics-2023/PS2/ps2_code/results/pf/output_data
```

In [72]:
```
plt.figure(figsize=[20,30])
plot('x', pf_input, pf_output)
plot('y', pf_input, pf_output)
plot('th', pf_input, pf_output)
```

PF







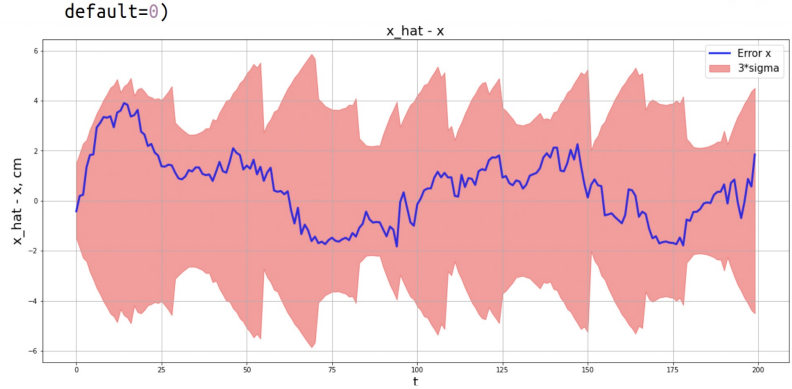
D. (15 pts) Once your filters are implemented, please investigate some properties of them. How do they behave

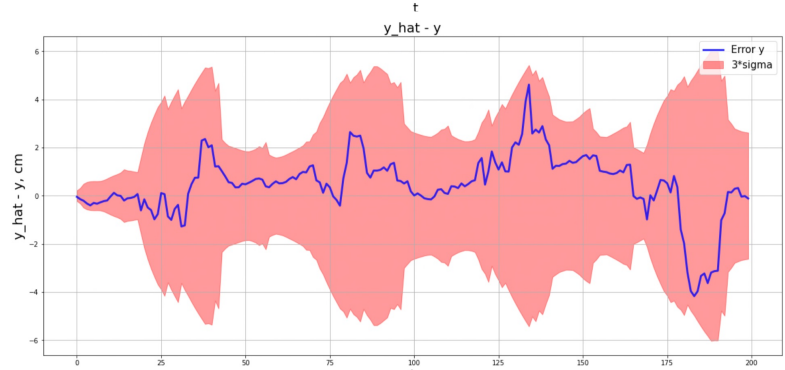
- as the sensor or motion noise go toward zero? (Please provide plots and explanation)
- as the number of particles decrease?
- if the filter noise parameters underestimate or overestimate the true noise parameters? Please clarify what underestimation and overestimation of noise is?

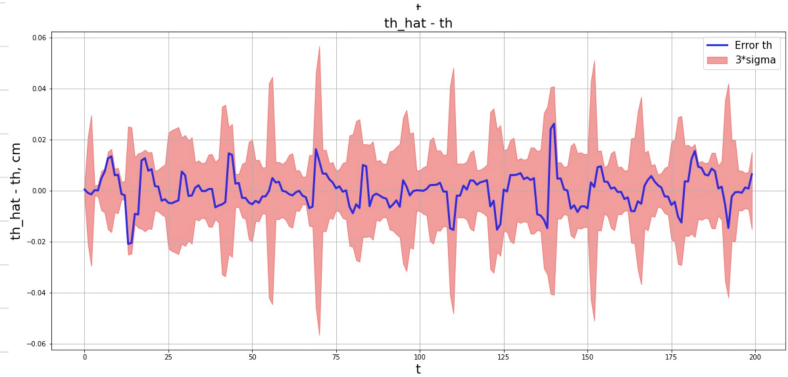
(Items 1 and 3 are for EKF and "the noise" refers to both observation and control noises)

```
parser.add_argument('-b',
                    '--bearing_std',
                    type=float,
                    action='store',
                    help='Diagonal of Standard deviations of the Observation noise Q. (format: rad).',
                    default=0)
```

Q = 0 -> bearing_std = 0



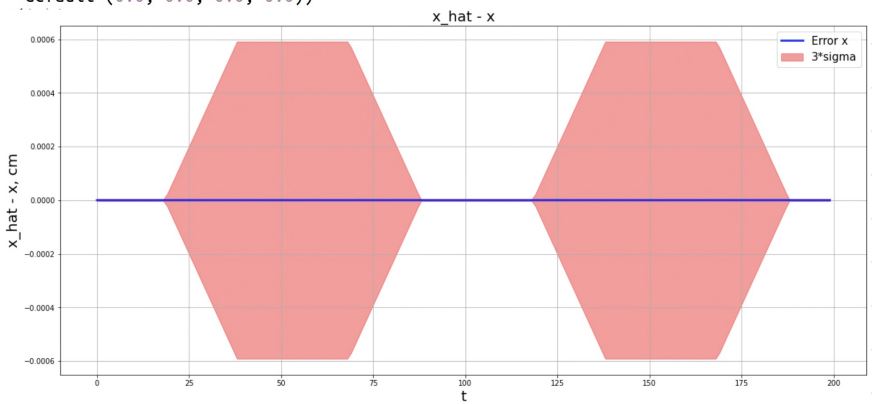


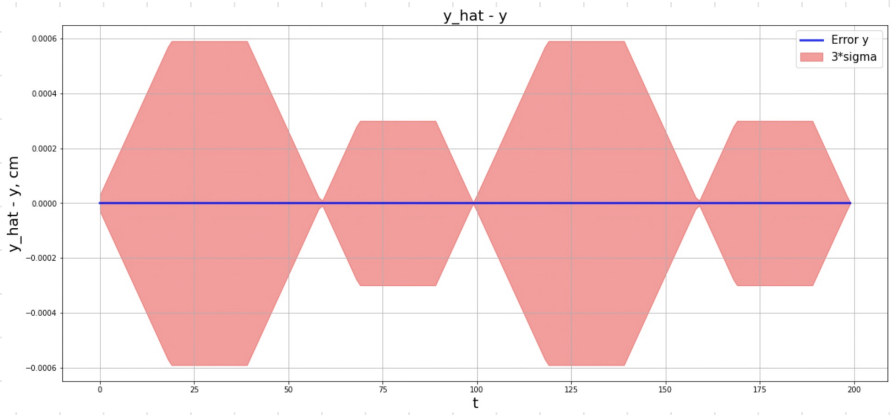


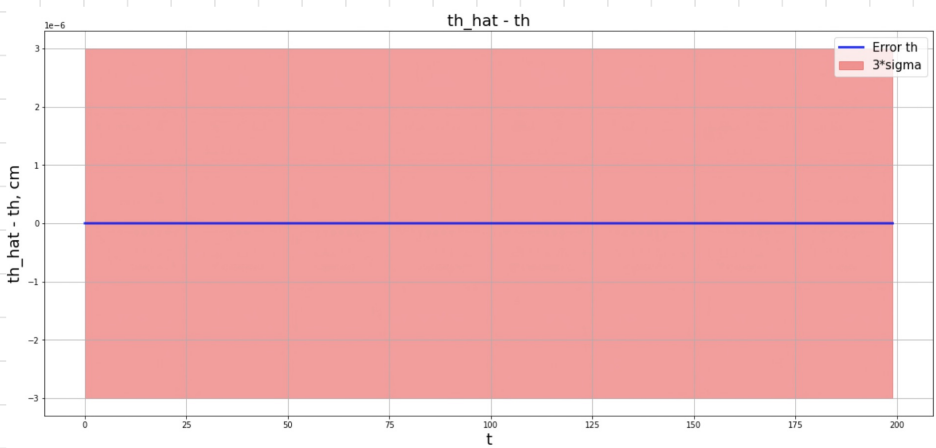
If we decrease sensor noise, error decreases too, but we also have motion noise, so trajectory estimation is not finally correct.

```
parser.add_argument('-a',
                    '--alphas',
                    nargs=4,
                    metavar=('A1', 'A2', 'A3', 'A4'),
                    action='store',
                    help='Squared root of alphas, used for transition noise in action space (M_t). (format: a1 a2 a3 a4).'
                    default=(0.0, 0.0, 0.0, 0.0))
```

R = 0 -> alphas = [0, 0, 0, 0]



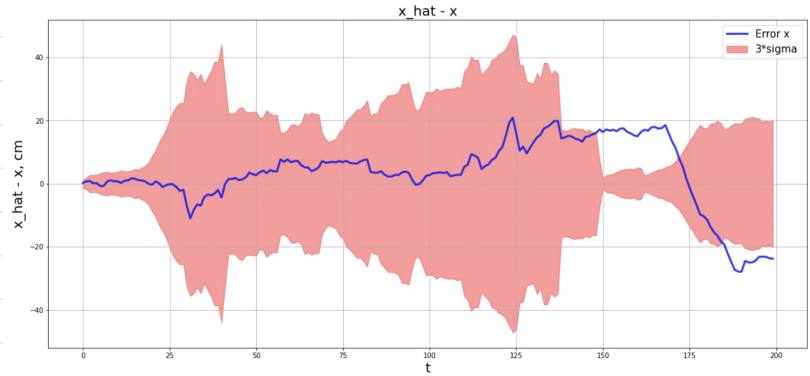


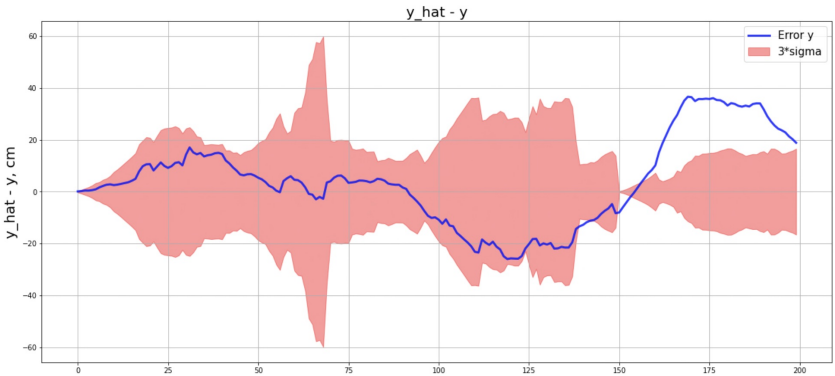


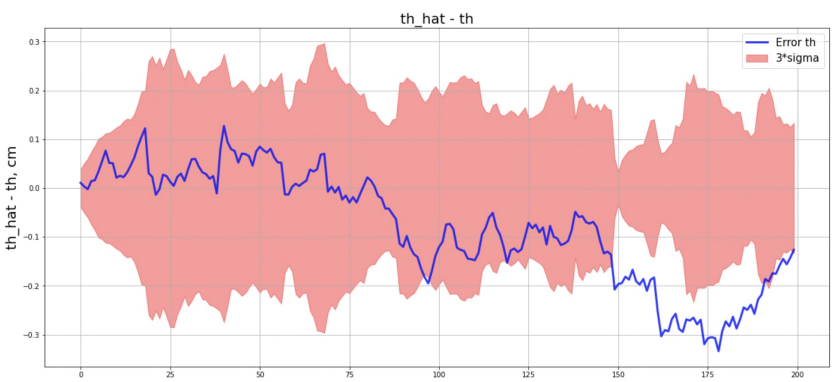
If we define motion noise R equals zero, error in trajectory estimation goes to zero too due to effective work of Extended Kalman Filter. Value of sensor noise almost does not affect on estimation.

Number of particles = 30

```
class PF(LocalizationFilter):
    def __init__(self, initial_state, alphas, bearing_std, num_particles, global_localization):
        super(PF, self).__init__(initial_state, alphas, bearing_std)

        # TODO add here specific class variables for the PF
        self.num_particles = 30
        self.X = np.random.multivariate_normal(self.mu, self.Sigma, self.num_particles)
        self.weights = np.zeros(self.num_particles)
```

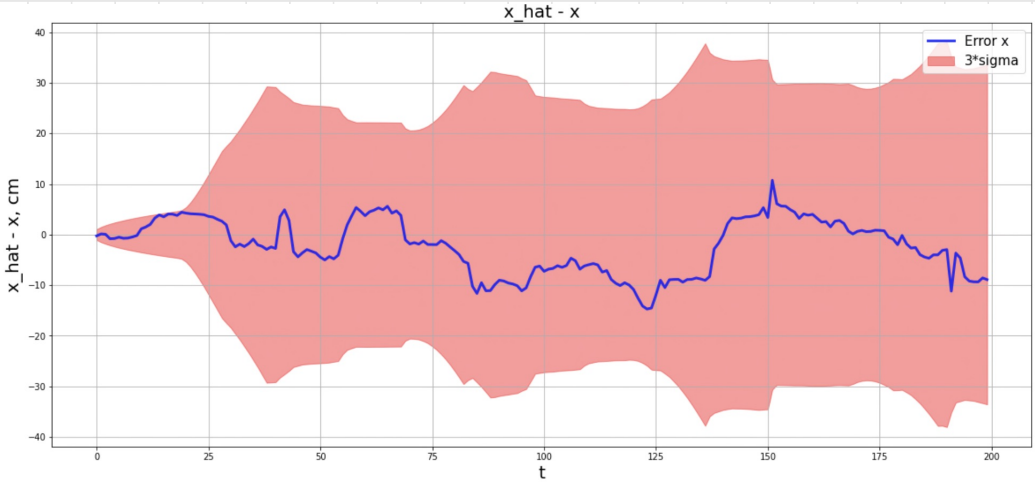


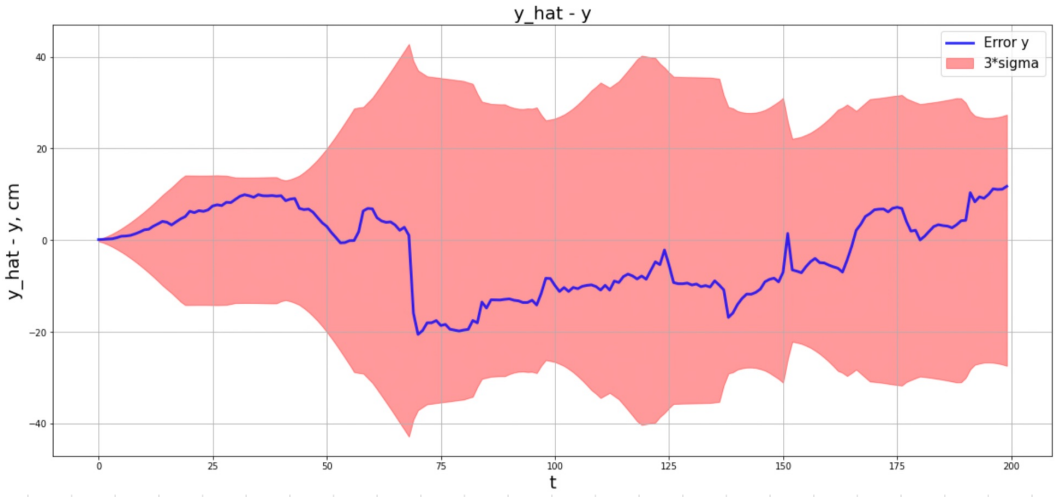


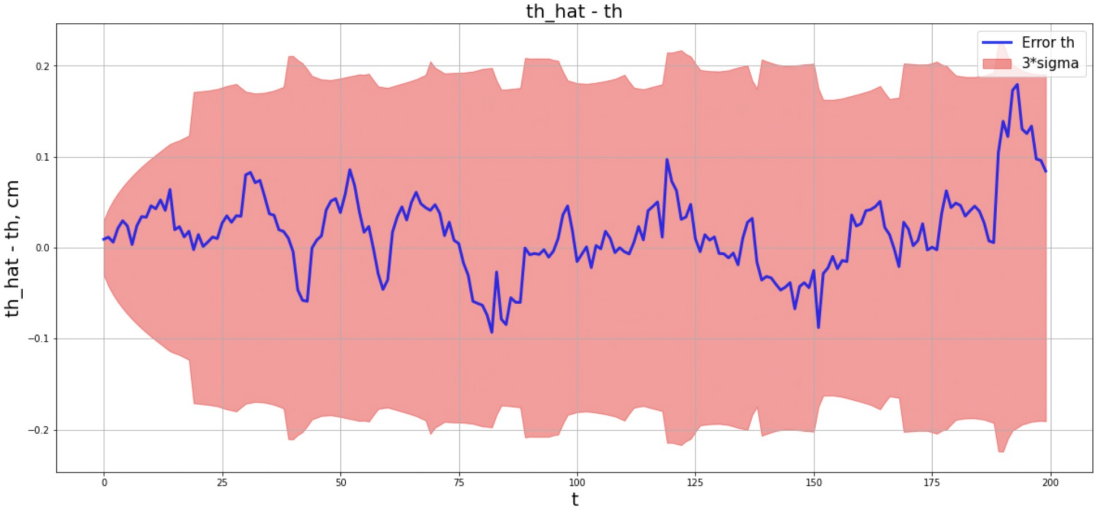


If we decrease number of particles from 100 to 30, estimation error goes out from 3*sigma interval. This case cannot be used for trajectory estimation due to incorrect localisation information.

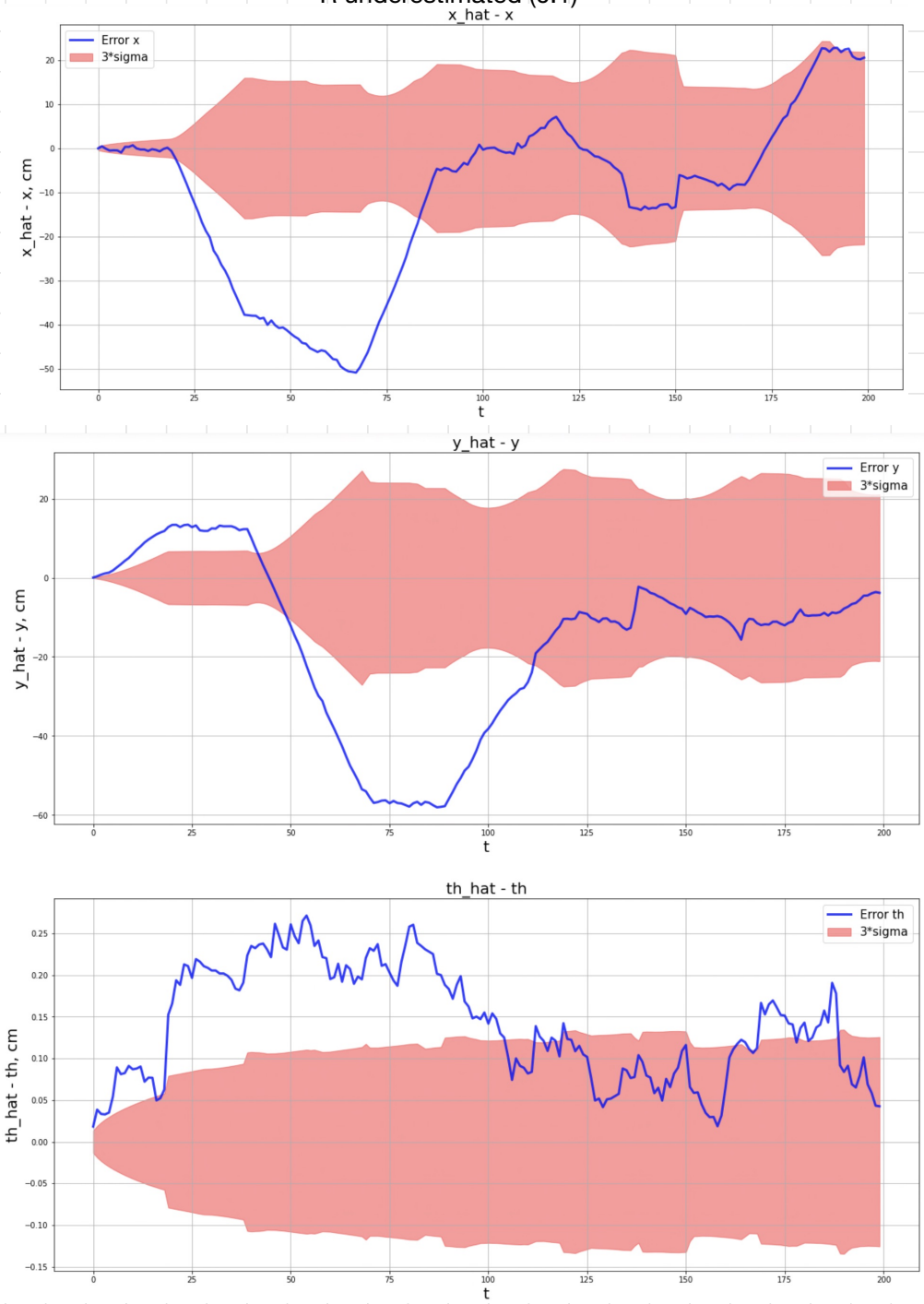
R underestimated (0.5)





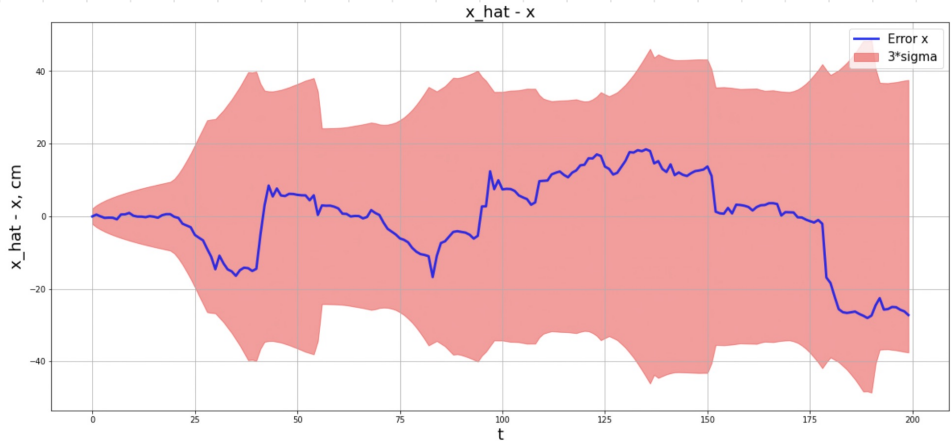


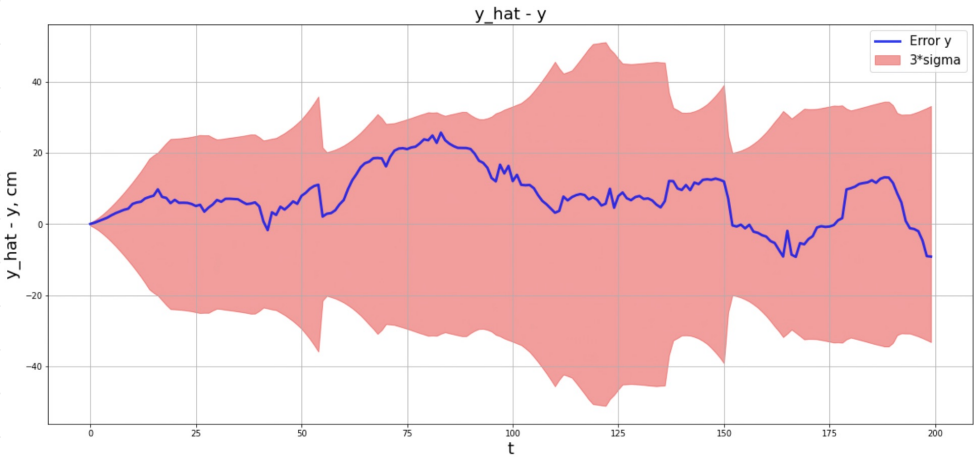
R underestimated (0.1)

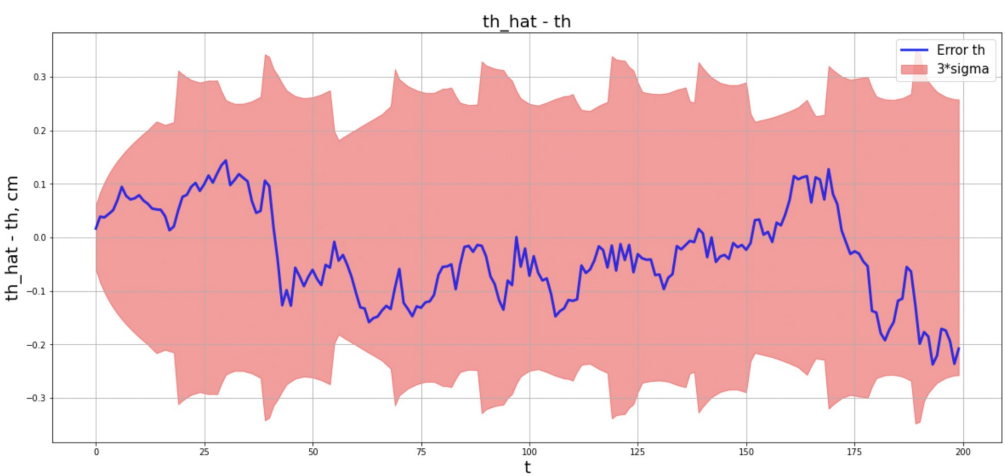


Underestimating of motion noise significantly affect on trajectory estimation only if we decrease matrix R by more than 10 times. When we reduce matrix by 2 times, changes are insignificant.

R overestimated (2)

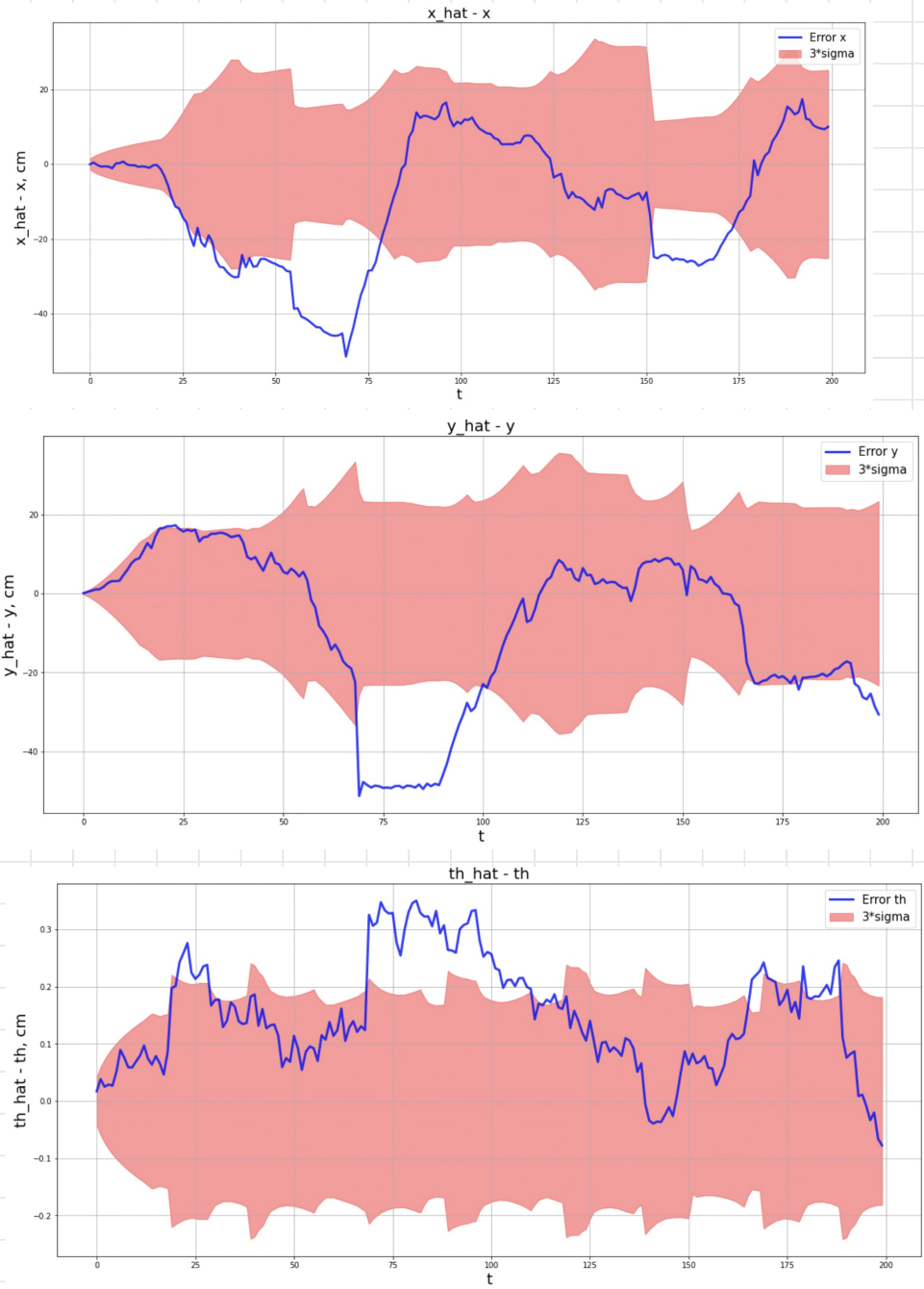






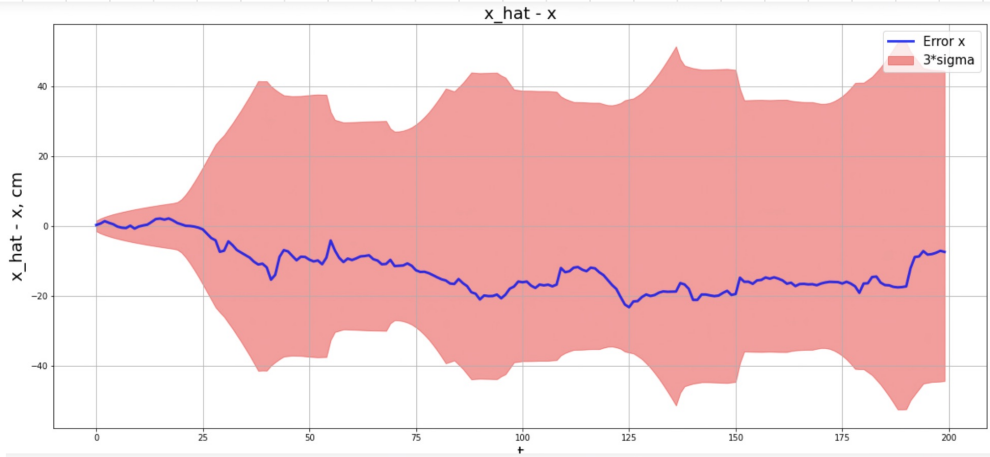
Overestimating of motion noise only decrease 3*sigma interval and almost does not affect on trajectory estimation.

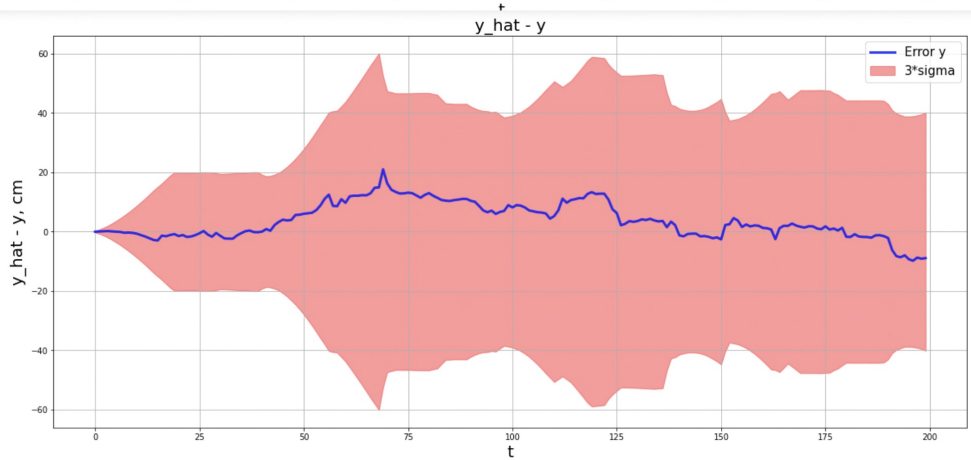
Q underestimated (0.5)

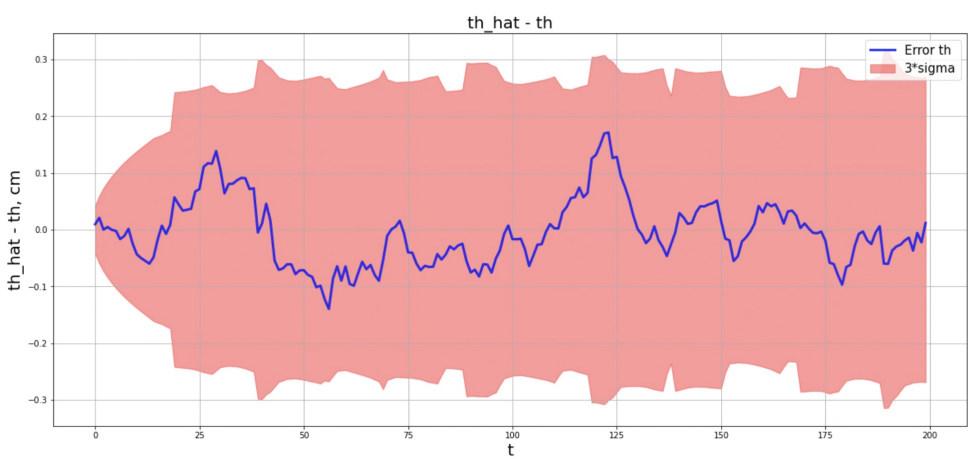


Underestimating of sensor noise more affect on trajectory estimation compare to motion noise. Error become significant with matrix Q reduced by 2 times.

Q overestimated (2)







Overestimating of sensor noise just increase 3*sigma interval.
Conclusion: In all cases it is better to overestimate the noise, than underestimate.