Kinematic car sensor fusion example

RMM, 24 Feb 2022 (updated 23 Feb 2023)

In this example we work through estimation of the state of a car changing lanes with two different sensors available: one with good longitudinal accuracy and the other with good lateral accuracy.

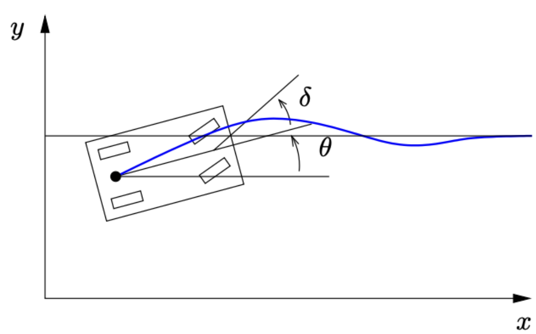
All calculations are done in discrete time, using both the form of the Kalman filter in Theorem 7.2 and the predictor corrector form.

In [1]:

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import control as ct
import control.optimal as opt
import control.flatsys as fs

# Define some line styles for later use
ebarstyle = {'elinewidth': 0.5, 'capsize': 2}
xdstyle = {'color': 'k', 'linestyle': '--', 'linewidth': 0.5,
           'marker': '+', 'markersize': 4}
```

System definition

We make use of a simple model for a vehicle navigating in the plane, known as the "bicycle model". The kinematics of this vehicle can be written in terms of the contact point (x, y) and the angle θ of the vehicle with respect to the horizontal axis:



$$
\begin{aligned}
\dot{x} &= \cos\theta\, v \\
\dot{y} &= \sin\theta\, v \\
\dot{\theta} &= \frac{v}{l}\tan\delta
\end{aligned}
$$

The input v represents the velocity of the vehicle and the input δ represents the turning rate. The parameter l is the wheelbase.

In [2]:

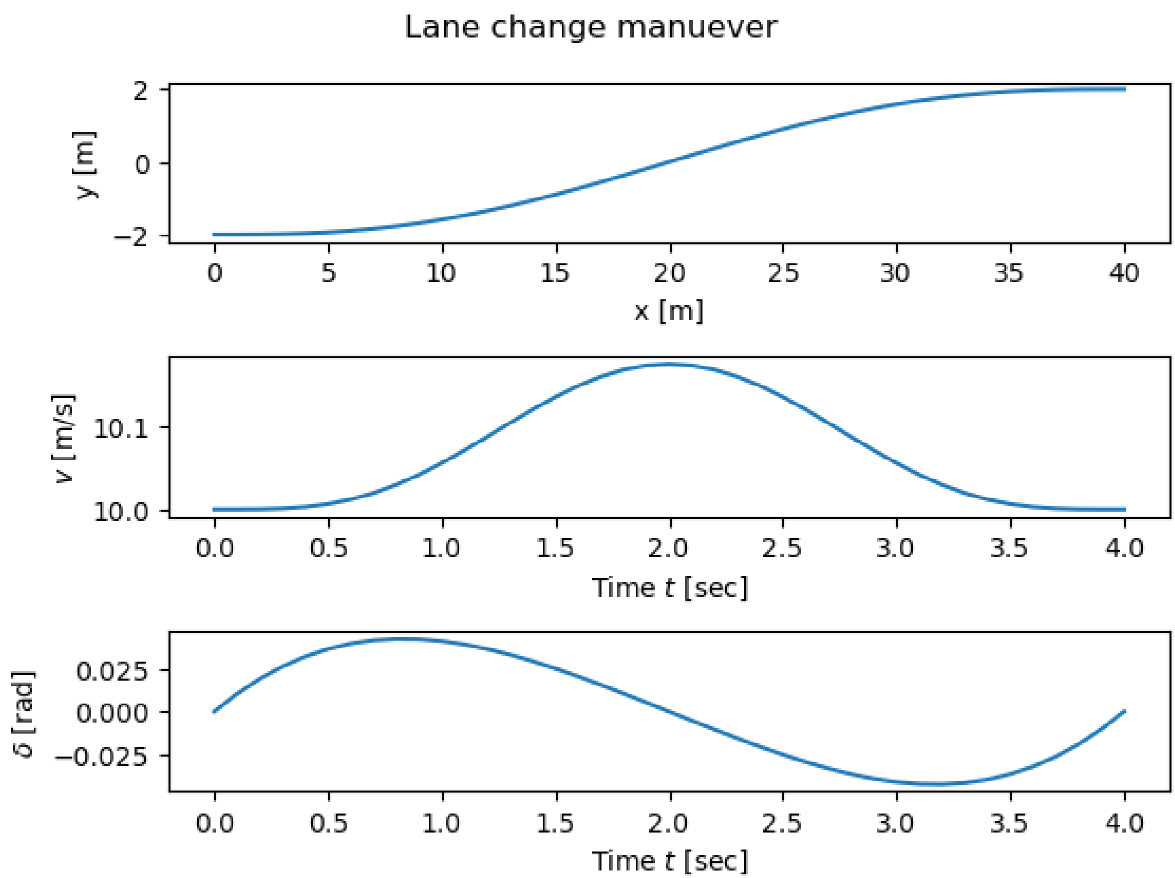
```
# Vehicle steering dynamics
#
# System state: x, y, theta
# System input: v, phi
```

```python
# System output: x, y
# System parameters: wheelbase, maxsteer
#
from kincar import kincar, plot_lanechange
print(kincar)
```

```
<FlatSystem>: kincar
Inputs (2): ['v', 'delta']
Outputs (3): ['x', 'y', 'theta']
States (3): ['x', 'y', 'theta']

Update: <function _kincar_update at 0x7fea526a4a60>
Output: <function _kincar_output at 0x7fea61832710>

Forward: <function _kincar_flat_forward at 0x7fea526a4820>
Reverse: <function _kincar_flat_reverse at 0x7fea526a4700>
```

In [3]:

```python
# Generate a trajectory for the vehicle
# Define the endpoints of the trajectory
x0 = [0., -2., 0.]; u0 = [10., 0.]
xf = [40., 2., 0.]; uf = [10., 0.]
Tf = 4

# Find a trajectory between the initial condition and the final condition
traj = fs.point_to_point(kincar, Tf, x0, u0, xf, uf, basis=fs.PolyFamily(6))

# Create the desired trajectory between the initial and final condition
Ts = 0.1
# Ts = 0.5
timepts = np.arange(0, Tf + Ts, Ts)
xd, ud = traj.eval(timepts)

plot_lanechange(timepts, xd, ud)
```



Discrete time system model

For the model that we use for the Kalman filter, we take a simple discretization using the approximation that $\dot{x} = (x[k+1] - x[k])/T_s$ where T_s is the sampling time.

In [4]:
```
#
# Create a discrete time, linear model
#

# Linearize about the starting point
linsys = ct.linearize(kincar, x0, u0)

# Create a discrete time model by hand
Ad = np.eye(linsys.nstates) + linsys.A * Ts
Bd = linsys.B * Ts
discsys = ct.ss(Ad, Bd, np.eye(linsys.nstates), 0, dt=Ts)
print(discsys);
```

```
<LinearIOSystem>: sys[3]
Inputs (2): ['u[0]', 'u[1]']
Outputs (3): ['y[0]', 'y[1]', 'y[2]']
States (3): ['x[0]', 'x[1]', 'x[2]']

A = [[ 1.0000000e+00  0.0000000e+00 -5.0004445e-07]
     [ 0.0000000e+00  1.0000000e+00  1.0000000e+00]
     [ 0.0000000e+00  0.0000000e+00  1.0000000e+00]]

B = [[0.1        0.        ]
     [0.         0.        ]
     [0.         0.33333333]]

C = [[1. 0. 0.]
     [0. 1. 0.]
     [0. 0. 1.]]

D = [[0. 0.]
     [0. 0.]
     [0. 0.]]

dt = 0.1
```

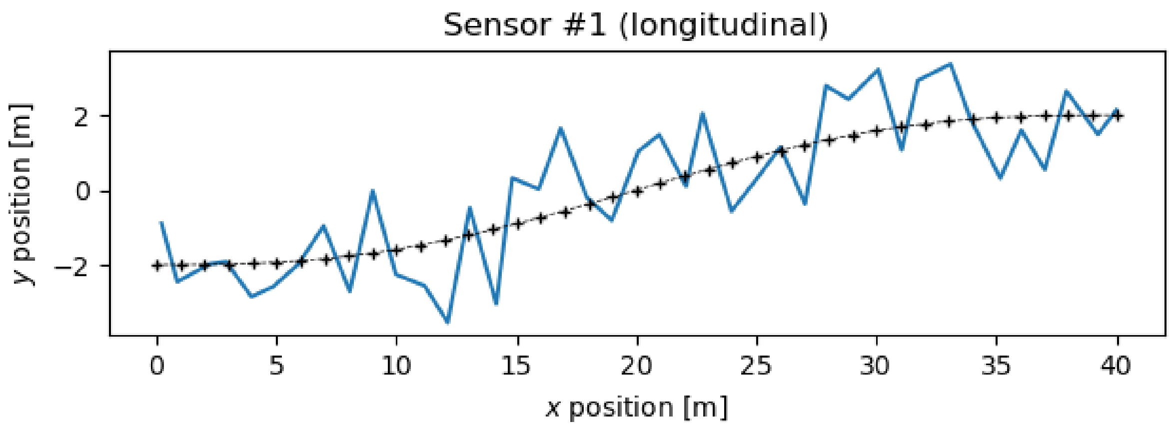
Sensor model

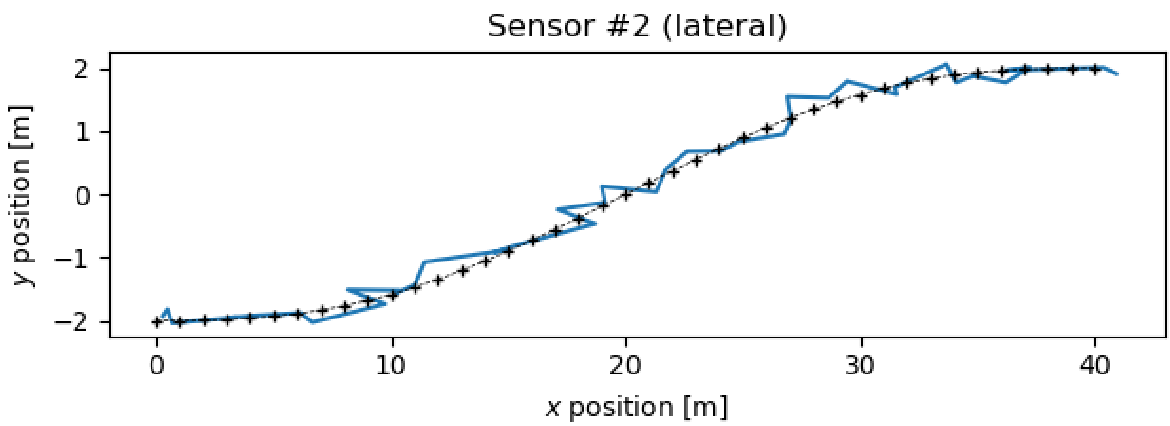
We assume that we have two sensors: one with good longitudinal accuracy and the other with good lateral accuracy. For each sensor we define the map from the state space to the sensor outputs, the covariance matrix for the measurements, and a white noise signal (now in discrete time).

Note: we pass the keyword `dt` to the `white_noise` function so that the white noise is consistent with a discrete time model (so the covariance is *not* rescaled by $\sqrt{dt}$).

In [5]:
```
# Sensor #1: longitudinal
C_lon = np.eye(2, discsys.nstates)
Rw_lon = np.diag([0.1 ** 2, 1 ** 2])
W_lon = ct.white_noise(timepts, Rw_lon, dt=Ts)

# Sensor #2: lateral
C_lat = np.eye(2, discsys.nstates)
Rw_lat = np.diag([1 ** 2, 0.1 ** 2])
W_lat = ct.white_noise(timepts, Rw_lat, dt=Ts)

# Plot the noisy signals
plt.subplot(2, 1, 1)
Y = xd[0:2] + W_lon
plt.plot(Y[0], Y[1])
plt.plot(xd[0], xd[1], **xdstyle)
plt.xlabel("$x$ position [m]")
plt.ylabel("$y$ position [m]")
plt.title("Sensor #1 (longitudinal)")

plt.subplot(2, 1, 2)
```

```
Y = xd[0:2] + W_lat
plt.plot(Y[0], Y[1])
plt.plot(xd[0], xd[1], **xdstyle)
plt.xlabel("$x$ position [m]")
plt.ylabel("$y$ position [m]")
plt.title("Sensor #2 (lateral)")
plt.tight_layout()
```





Linear Quadratic Estimator

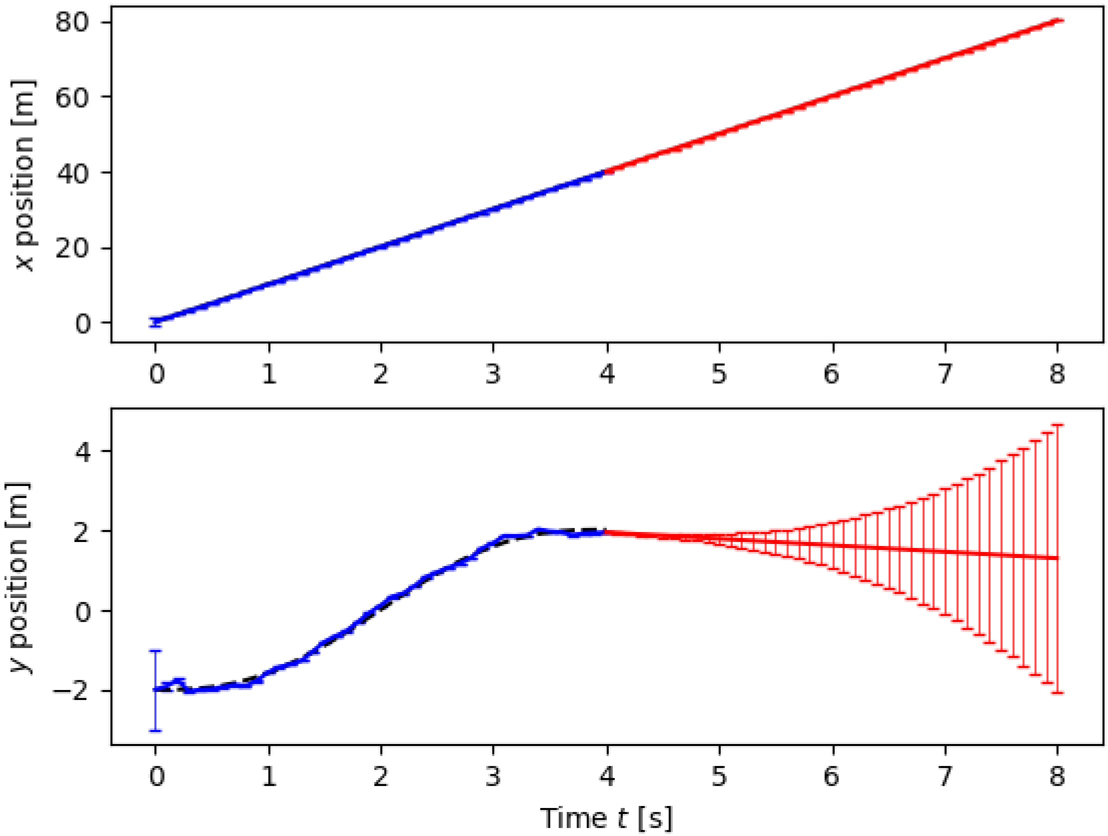
We now construct a linear quadratic estimator for the system usign the Kalman filter form. This is idone using the `create_estimator_iosystem` function in python-control.

In [6]:
```
# Disturbance and initial condition model
# Note: multiple by sampling time since we discretized the dynamics
Rv = np.diag([0.1, 0.01]) * Ts
# Rv = np.diag([10, 1]) * Ts        # Variant: no input information
P0 = np.diag([1, 1, 0.1])

# Combine the sensors
# Note: no sampling time here because we are doing discrete-time KF
C = np.vstack([C_lon, C_lat])
Rw = sp.linalg.block_diag(Rw_lon, Rw_lat)

estim = ct.create_estimator_iosystem(discsys, Rv, Rw, C=C, P0=P0)
print(estim)
```

```
<NonlinearIOSystem>: sys[4]
Inputs (6): ['y[0]', 'y[1]', 'y[2]', 'y[3]', 'u[0]', 'u[1]']
Outputs (3): ['xhat[0]', 'xhat[1]', 'xhat[2]']
States (12): ['xhat[0]', 'xhat[1]', 'xhat[2]', 'P[0,0]', 'P[0,1]', 'P[0,
2]', 'P[1,0]', 'P[1,1]', 'P[1,2]', 'P[2,0]', 'P[2,1]', 'P[2,2]']

Update: <function create_estimator_iosystem.<locals>._estim_update at 0x7fe
a529b7910>
Output: <function create_estimator_iosystem.<locals>._estim_output at 0x7fe
a529b79a0>
```

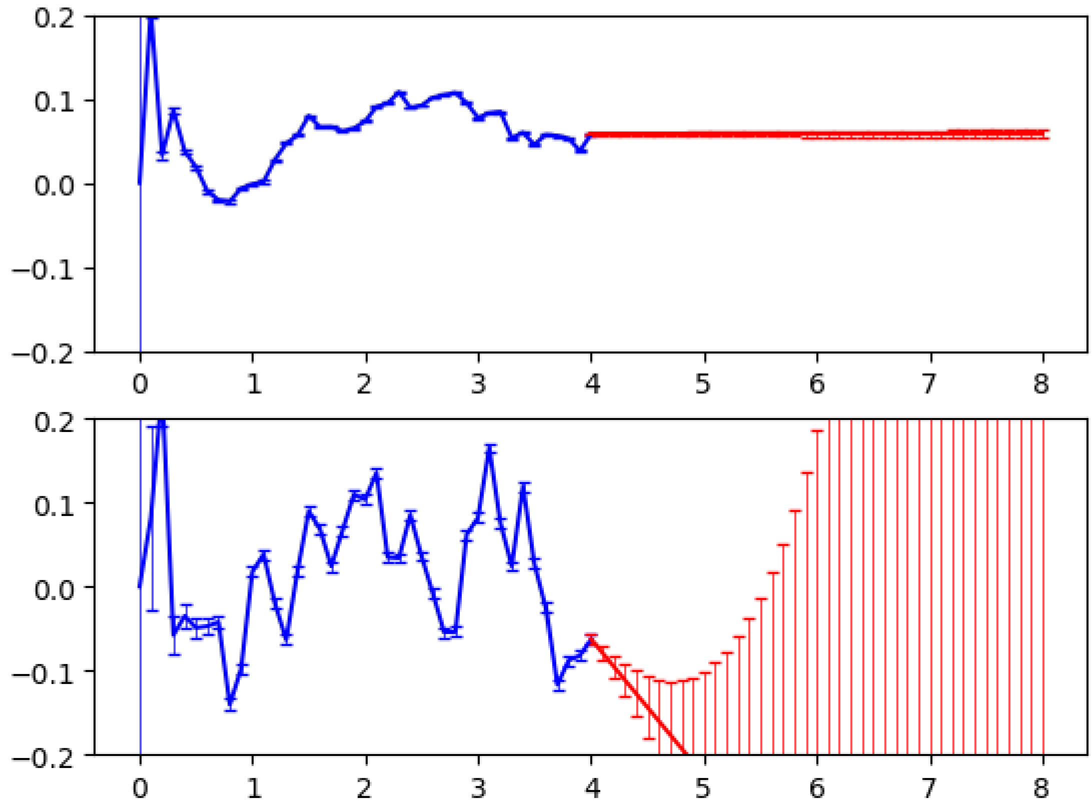
We can now run the estimator on the noisy signals to see how well it works.

In [7]:
```
# Compute the inputs to the estimator
Y = np.vstack([xd[0:2] + W_lon, xd[0:2] + W_lat])
U = np.vstack([Y, ud])        # add input to the Kalman filter
# U = np.vstack([Y, ud * 0])  # variant: no input information
X0 = np.hstack([xd[:, 0], P0.reshape(-1)])

# Run the estimator on the trajectory
estim_resp = ct.input_output_response(estim, timepts, U, X0)

# Run a prediction to see what happens next
T_predict = np.arange(timepts[-1], timepts[-1] + 4 + Ts, Ts)
U_predict = np.outer(U[:, -1], np.ones_like(T_predict))
predict_resp = ct.input_output_response(
    estim, T_predict, U_predict, estim_resp.states[:, -1],
    params={'correct': False})

# Plot the estimated trajectory versus the actual trajectory
plt.subplot(2, 1, 1)
plt.errorbar(
    estim_resp.time, estim_resp.outputs[0],
    estim_resp.states[estim.find_state('P[0,0]')], fmt='b-', **ebarstyle)
plt.errorbar(
    predict_resp.time, predict_resp.outputs[0],
    predict_resp.states[estim.find_state('P[0,0]')], fmt='r-', **ebarstyle)
plt.plot(timepts, xd[0], 'k--')
plt.ylabel("$x$ position [m]")

plt.subplot(2, 1, 2)
plt.errorbar(
    estim_resp.time, estim_resp.outputs[1],
    estim_resp.states[estim.find_state('P[1,1]')], fmt='b-', **ebarstyle)
plt.errorbar(
    predict_resp.time, predict_resp.outputs[1],
    predict_resp.states[estim.find_state('P[1,1]')], fmt='r-', **ebarstyle)
# lims = plt.axis(); plt.axis([lims[0], lims[1], -5, 5])
plt.plot(timepts, xd[1], 'k--');
plt.ylabel("$y$ position [m]")
plt.xlabel("Time $t$ [s]");
```



In [8]:

```
# Plot the estimated errors
plt.subplot(2, 1, 1)
plt.errorbar(
    estim_resp.time, estim_resp.outputs[0] - xd[0],
    estim_resp.states[estim.find_state('P[0,0]')], fmt='b-', **ebarstyle)
plt.errorbar(
    predict_resp.time, predict_resp.outputs[0] - (xd[0] + xd[0, -1]),
    predict_resp.states[estim.find_state('P[0,0]')], fmt='r-', **ebarstyle)
lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2])
# lims = plt.axis(); plt.axis([lims[0], lims[1], -2, 0.2])

plt.subplot(2, 1, 2)
plt.errorbar(
    estim_resp.time, estim_resp.outputs[1] - xd[1],
    estim_resp.states[estim.find_state('P[1,1]')], fmt='b-', **ebarstyle)
plt.errorbar(
    predict_resp.time, predict_resp.outputs[1] - xd[1, -1],
    predict_resp.states[estim.find_state('P[1,1]')], fmt='r-', **ebarstyle)
lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2]);
```



Things to try

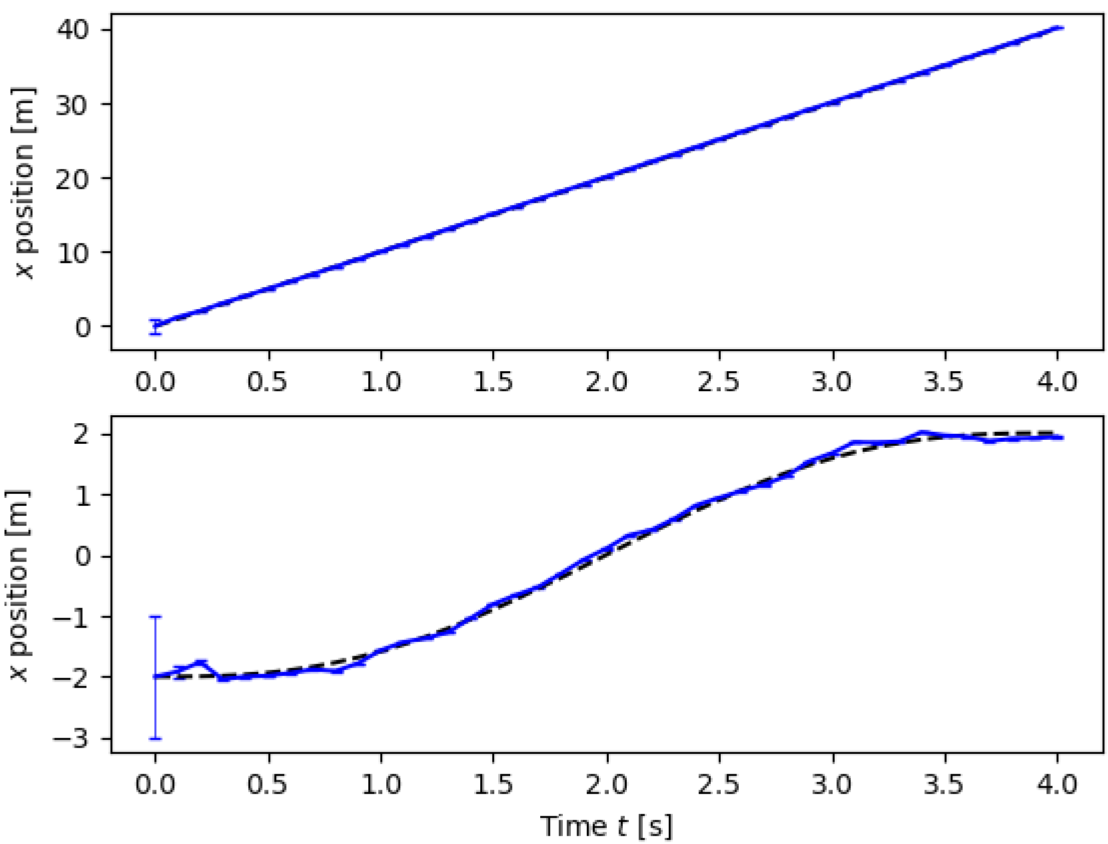
- Remove the input (and update P0 and Rv)
- Change the sampling rate

Predictor-corrector form

Instead of using create_estimator_iosystem, we can also compute out the estimate in a more manual fashion, done here using the predictor-corrector form.

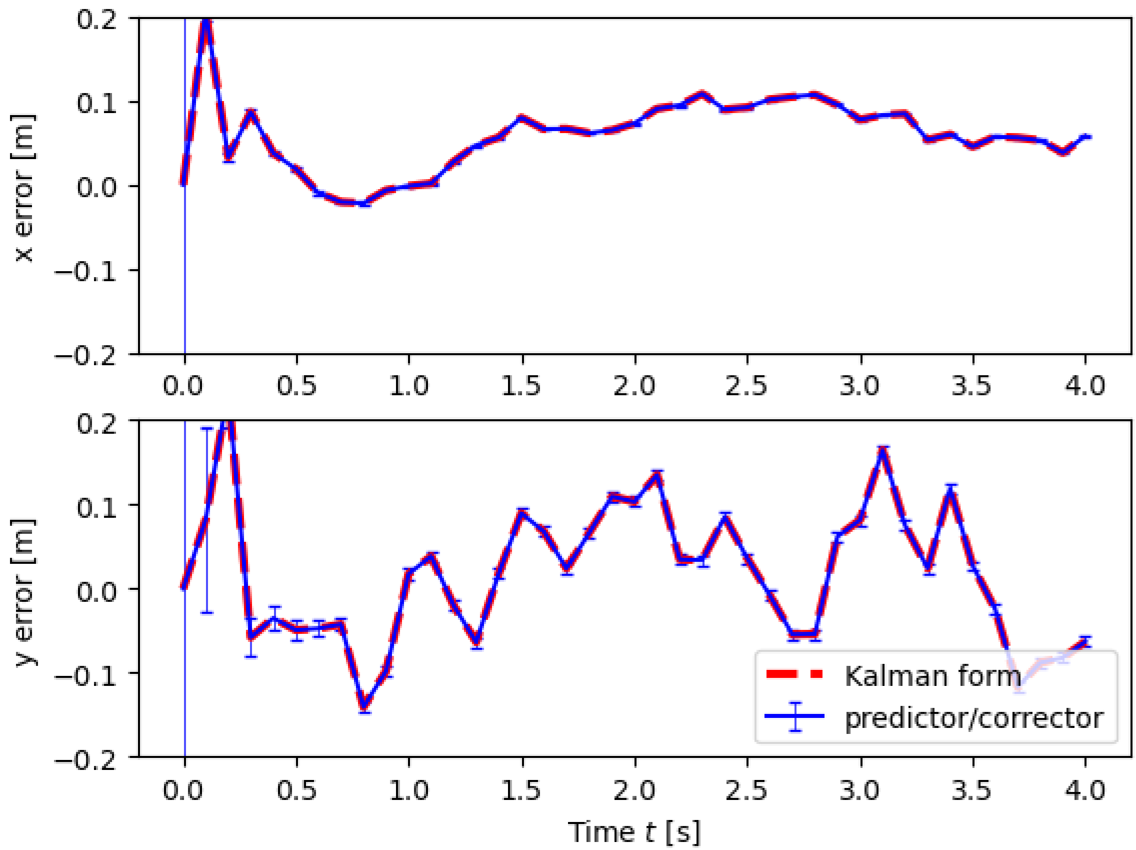
In [9]:
```
# System matrices
A, B, F = discsys.A, discsys.B, discsys.B

# Create an array to store the results
xhat = np.zeros((discsys.nstates, timepts.size))
P = np.zeros((discsys.nstates, discsys.nstates, timepts.size))

# Update the estimates at each time
for i, t in enumerate(timepts):
    # Prediction step
    if i == 0:
        # Use the initial condition
        xkkm1 = xd[:, 0]
        Pkkm1 = P0
    else:
        xkkm1 = A @ xkk + B @ ud[:, i-1]
```

```
        Pkkm1 = A @ Pkk @ A.T + F @ Rv @ F.T

    # Correction step (variant: apply only when sensor data is available)
    L = Pkkm1 @ C.T @ np.linalg.inv(Rw + C @ Pkkm1 @ C.T)
    xkk = xkkm1 - L @ (C @ xkkm1 - Y[:, i])
    Pkk = Pkkm1 - L @ C @ Pkkm1

    # Save the state estimate and covariance for later plotting
    xhat[:, i], P[:, :, i] = xkkm1, Pkkm1  # For comparison to Kalman form
    # xhat[:, i], P[:, :, i] = xkk, Pkk    # variant:

plt.subplot(2, 1, 1)
plt.errorbar(timepts, xhat[0], P[0, 0], fmt='b-', **ebarstyle)
plt.plot(timepts, xd[0], 'k--')
plt.ylabel("$x$ position [m]")

plt.subplot(2, 1, 2)
plt.errorbar(timepts, xhat[1], P[1, 1], fmt='b-', **ebarstyle)
plt.plot(timepts, xd[1], 'k--')
plt.ylabel("$x$ position [m]")
plt.xlabel("Time $t$ [s]");
```



In [10]:
```
# Plot the estimated errors (and compare to Kalman form)
plt.subplot(2, 1, 1)
plt.errorbar(timepts, xhat[0] - xd[0], P[0, 0], fmt='b-', **ebarstyle)
plt.plot(estim_resp.time, estim_resp.outputs[0] - xd[0], 'r--', linewidth=3)
lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2])
plt.ylabel("x error [m]")

plt.subplot(2, 1, 2)
```

```
plt.errorbar(timepts, xhat[1] - xd[1], P[1, 1], fmt='b-', **ebarstyle,
             label='predictor/corrector')
plt.plot(estim_resp.time, estim_resp.outputs[1] - xd[1], 'r--', linewidth=3,
         label='Kalman form')
lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2])
plt.ylabel("y error [m]")
plt.xlabel("Time $t$ [s]")
plt.legend(loc='lower right');
```

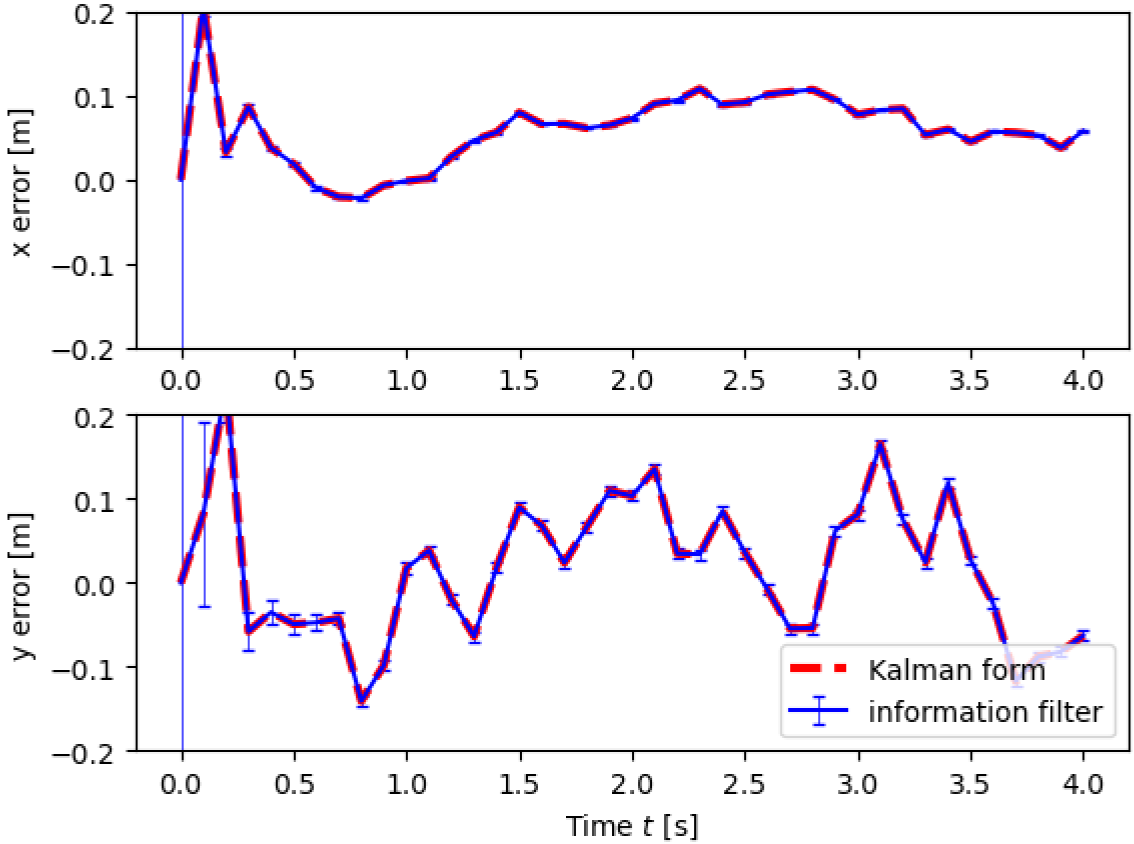


Information filter

An alternative way to implement the computation is using the information filter formulation.

In [11]:
```
from numpy.linalg import inv

# Update the estimates at each time
for i, t in enumerate(timepts):
    # Prediction step
    if i == 0:
        # Use the initial condition
        xkkm1 = xd[:, 0]
        Pkkm1 = P0
    else:
        xkkm1 = A @ xkk + B @ ud[:, i-1]
        Pkkm1 = A @ Pkk @ A.T + F @ Rv @ F.T
```

```
    # Correction step (variant: apply only when sensor data is available)
    Ikk, Zkk = inv(Pkkm1), inv(Pkkm1) @ xkkm1

    # Longitudinal sensor update
    Ikk += C_lon.T @ inv(Rw_lon) @ C_lon     # Omega_lon
    Zkk += C_lon.T @ inv(Rw_lon) @ Y[:2, i]  # Psi_lon

    # Lateral sensor update
    Ikk += C_lat.T @ inv(Rw_lat) @ C_lat     # Omega_lat
    Zkk += C_lat.T @ inv(Rw_lat) @ Y[2:, i]  # Psi_lat

    # Compute the updated state and covariance
    Pkk = inv(Ikk)
    xkk = Pkk @ Zkk

    # Save the state estimate and covariance for later plotting
    xhat[:, i], P[:, :, i] = xkkm1, Pkkm1

# Plot the estimated errors (and compare to Kalman form)
plt.subplot(2, 1, 1)
plt.errorbar(timepts, xhat[0] - xd[0], P[0, 0], fmt='b-', **ebarstyle)
plt.plot(estim_resp.time, estim_resp.outputs[0] - xd[0], 'r--', linewidth=3)
lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2])
plt.ylabel("x error [m]")

plt.subplot(2, 1, 2)
plt.errorbar(timepts, xhat[1] - xd[1], P[1, 1], fmt='b-', **ebarstyle,
            label='information filter')
plt.plot(estim_resp.time, estim_resp.outputs[1] - xd[1], 'r--', linewidth=3,
        label='Kalman form')
lims = plt.axis(); plt.axis([lims[0], lims[1], -0.2, 0.2])
plt.ylabel("y error [m]")
plt.xlabel("Time $t$ [s]")
plt.legend(loc='lower right');
```


0.2
0.1
0.0
−0.1
−0.2
x error [m]
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
y error [m]
Time t [s]
Kalman form
information filter

In [ ]: