homework 10

January 3, 2025

1 Homework 10 (DL Friday, November 29 at 12:00 PM)

ELEC-E8740 - Basics of sensor fusion - Autumn 2024

[1]:
```
import numpy as np
import scipy.linalg as linalg
import matplotlib.pyplot as plt
```

Consider the following 1D non-linear model

$$x_k = tanh(x_{k-1}) + q_{k-1}, y_k = sin(x_k) + r_k,$$

where $x_0 \sim \mathcal{N}(0, 1)$, $q_{k-1} \sim \mathcal{N}(0, 0.1^2)$, and $r_k \sim \mathcal{N}(0, 0.1^2)$.

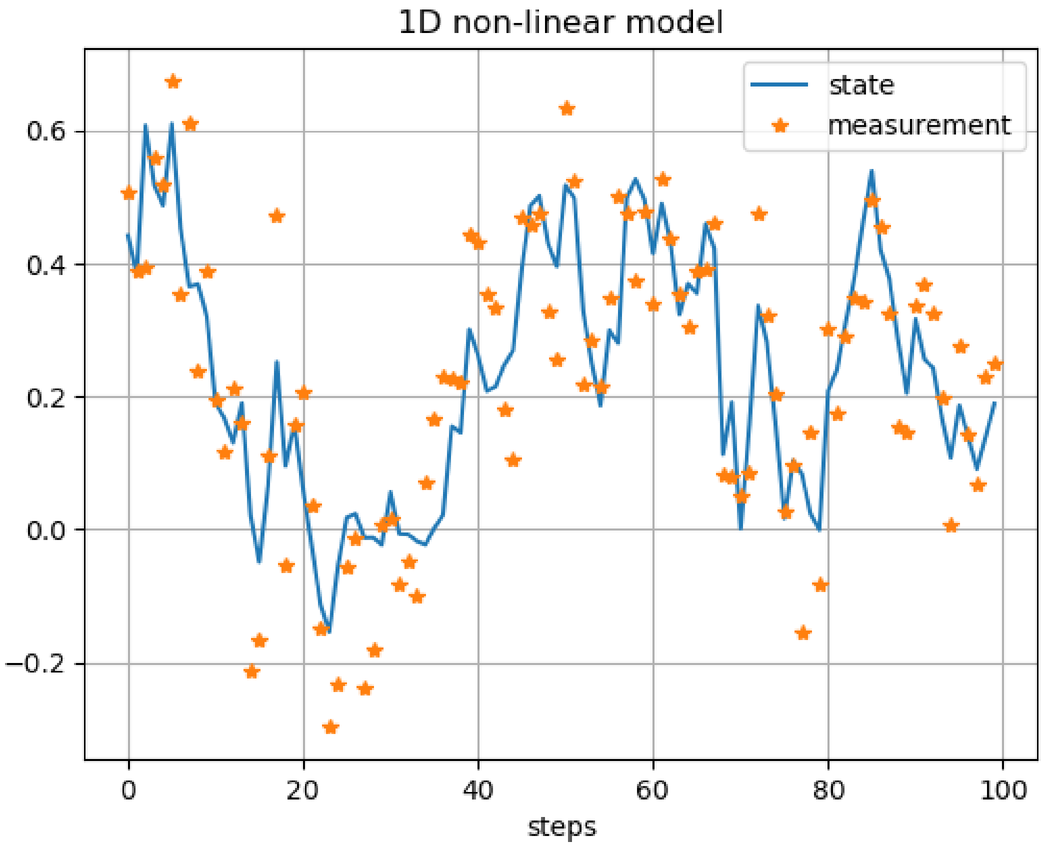
1.0.1 Part a (1 point): Simulate 100 steps of states and measurements from the model. Plot the data.

[3]:
```
def model_simulation(seed_number, steps):
    """
    1D non-linear model simulation
    -------
    Input:
        seed_number: it is used to generate the same sequence of random numbers
        steps: number of steps
    Output:
        xs: state trajectory
        ys: measurement tajectory

    """
    np.random.seed(seed_number)      # do not change this line
    xs = np.zeros((steps, 1))        # do not change this line
    ys = np.zeros((steps, 1))        # do not change this line
    # To draw random samples from a normal (Gaussian) distribution, you could␣
 ↪use np.random.normal function
    # Attention: the arguments of np.random.normal are mean and "Standard␣
 ↪deviation"
    # YOUR CODE HERE
     # Initialize the state with a random value from N(0, 1)
    xs[0, 0] = np.random.normal(0, 1)
```

```python
    for k in range(1, steps):
        # Generate process noise q_k ~ N(0, 0.1^2)
        q_k = np.random.normal(0, 0.1)

        # State update equation
        xs[k, 0] = np.tanh(xs[k-1, 0]) + q_k

    for k in range(steps):
        # Generate measurement noise r_k ~ N(0, 0.1^2)
        r_k = np.random.normal(0, 0.1)

        # Measurement equation
        ys[k, 0] = np.sin(xs[k, 0]) + r_k
    return xs, ys # do not change this line
```

Feel free to uncomment and run the given code below.

[4]:
```python
xs, ys = model_simulation(5, 100)
plt.plot(xs, label='state')
plt.plot(ys, '*', label='measurement')
plt.title('1D non-linear model')
plt.xlabel('steps')
plt.legend()
plt.grid();
```



[ ]:

1.0.2 Part b (1 point): Derive the necessary derivatives.

[5]:
```
def derivatives_ssm(x):
    """
    Derivatives of dynamic functin and measurement function
    -------
    Input:
        x: state
    Output:
        Fx: value of derivative of the dynamic function tanh(.) at state x
        Gx: value of derivative of the measurement function sin(.) at state x

    """
    # Fx = ?
    # Gx = ?
    # YOUR CODE HERE
    # Derivative of tanh(x)
    Fx = 1 - np.tanh(x)**2
```

```
    # Derivative of sin(x)
    Gx = np.cos(x)
    return Fx, Gx # do not change this line
```

[6]:
```
assert np.allclose(derivatives_ssm(0), (1.0, 1.0), rtol=1e-03, atol=1e-04)
assert np.allclose(derivatives_ssm(-np.pi/2), (0.159, 0.0), rtol=1e-03,␣
 ↪atol=1e-04)
```

1.0.3 Part b (1 point): and check that they are correct by using numerical finite differences.

To approximately find the derivative a function $f(.)$ using finite difference method, you could select a small step size h and compute $\frac{f(x+h)-f(x)}{h}$. In this part, use this method to find the derivatives of the dynamic function $tanh(.)$ and measurement function $sin()$.

[7]:
```
def derivatives_numerically(x, h):
    """
    Derivatives of dynamic functin and measurement function by using numerical␣
 ↪finite differences
    -------
    Input:
        x: state
        h: step size
    Output:
        Fx_n: value of numerical derivative of the dynamic function tanh(.) at␣
 ↪state x
        Gx_n: value of numerical derivative of the measurement function sin(.)␣
 ↪at state x

    """
    # YOUR CODE HERE
    # Numerical derivative of tanh(x)
    Fx_n = (np.tanh(x + h) - np.tanh(x)) / h

    # Numerical derivative of sin(x)
    Gx_n = (np.sin(x + h) - np.sin(x)) / h
    return Fx_n, Gx_n # do not change this line
```

[8]:
```
assert np.allclose(derivatives_numerically(0.0, 0.5), (0.924, 0.958),␣
 ↪rtol=1e-03, atol=1e-04)
assert np.allclose(derivatives_numerically(0.0, 1e-6), (0.999, 0.999),␣
 ↪rtol=1e-03, atol=1e-04)
assert np.allclose(derivatives_numerically(-np.pi/2, 1e-6), (0.159, 5e-7),␣
 ↪rtol=1e-03, atol=1e-04)
```

1.0.4 Part c (1 point): Implement and run EKF for the model. Plot the results.

Note: the input of the following "Extended_Kalman_Filter" function is only the measurements. Please do not change that and define any nesessary parameters inside the function.

The output should be Extended Kalman filter means and covariances of the whole trajectory.

[9]:
```
def Extended_Kalman_Filter(Y):
    """
    Extended Kalman filter state estimation for 1D non-linear state space model

    -------
    Input:
        Y: measurements
    Output:
        mean_ekf: Extended Kalman filter mean estimation
        cov_ekf: Extended Kalman filter covariance estimation

    """
    steps = Y.shape[0]
    mean_ekf = np.zeros((steps, 1))       # do not change this line
    cov_ekf = np.zeros((steps, 1, 1))     # do not change this line
    # YOUR CODE HERE
    # Initialize parameters
    Q = 0.1**2  # Process noise covariance
    R = 0.1**2  # Measurement noise covariance
    P = np.array([[1.0]])  # Initial state covariance
    x = np.array([[0.0]])  # Initial state

    for k in range(steps):
        # Prediction step
        x_pred = np.tanh(x)  # State prediction
        F_k = 1 - np.tanh(x)**2  # Jacobian of tanh(x)
        P_pred = F_k @ P @ F_k.T + Q  # Covariance prediction

        # Update step
        y_k = Y[k]  # Current measurement
        G_k = np.cos(x_pred)  # Jacobian of sin(x)
        S_k = G_k @ P_pred @ G_k.T + R  # Innovation covariance
        K_k = P_pred @ G_k.T @ np.linalg.inv(S_k)  # Kalman gain

        x = x_pred + K_k @ (y_k - np.sin(x_pred))  # State update
        P = (np.eye(1) - K_k @ G_k) @ P_pred  # Covariance update

        # Save results
        mean_ekf[k] = x
```

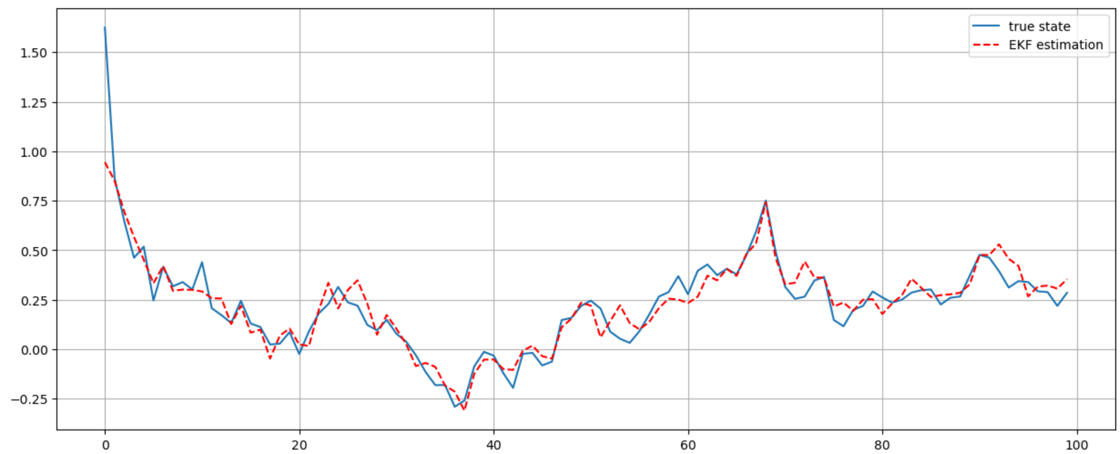
```
        cov_ekf[k] = P

    return mean_ekf, cov_ekf # do not change this line
```

[ ]:

Feel free to uncomment and run the given code below.

[10]:

```
observations =  model_simulation(1, 100)[1]
x_ekf, cov_ekf = Extended_Kalman_Filter(observations)
plt.figure(figsize=(15,6))
plt.plot(model_simulation(1, 100)[0], label='true state')
plt.plot(x_ekf[:,0], 'r--', label='EKF estimation')
plt.legend()
plt.grid();
```



[ ]: