Time Series Analysis and Recurrent Neural Network
Giacomo Barzon - 3626438
Exercise 11

January 19, 2020

1 Task 1 - Extended Kalman Filter

In file par.mat, you are given matrices A, W, B, Γ, Σ, and vector μ_0, which specify the parameters of the following non-linear recurrent neural network:

$$z_t = f(z_{t-1}, \epsilon_t) = Az_{t-1} + W\phi(z_{t-1}) + h + \epsilon_t, \ \epsilon_t \approx N(0, \Sigma)$$

where $\phi(z_t) = max(z_t, 0)$. We will assume that observations drawn from this (latent) dynamical system are just a linear transformation of the latent states z_t:

$$x_t = g(z_t, \eta_t) = Bz_t + \eta_t, \ \eta_t \approx N(0, \Gamma)$$

The initial state estimate is μ_0.

a) Assuming your latent state dimension to be equal to M = 3, and the observation dimension to be N = 10, let this system run for T = 500 time steps, and create "true" latent states $\{z_t\}_{1:T}$ and observations $\{x_t\}_{1:T}$ based on this system and the given parameters.
b) Run the Kalman filter that we have previously implemented (tutorial 5) to obtain latent state estimates based on your created observations.
c) Implement the extended Kalman filter (EKF), as described in (9.46) of the script. Use it to obtain latent state estimates based on your created observations.
d) Compare the obtained estimate of $\{z_t\}_{1:T}$ between the Kalman filter and the EKF by quantifying the mean squared error (MSE) between true and estimated latent states for both methods.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat                    #import mat data
```

1.0.1 a)

In [2]:
```
# Load data
mat_file = loadmat('par.mat')
#print(mat_file)
```

```
# model matrices
A = mat_file['A']
W = mat_file['W']
B = mat_file['B']
Sigma = mat_file['Sigma']
Gamma = mat_file['Gamma']

# model offset vector
h = mat_file['h'].T

# initial conditions
mu0 = mat_file['mu0']
```

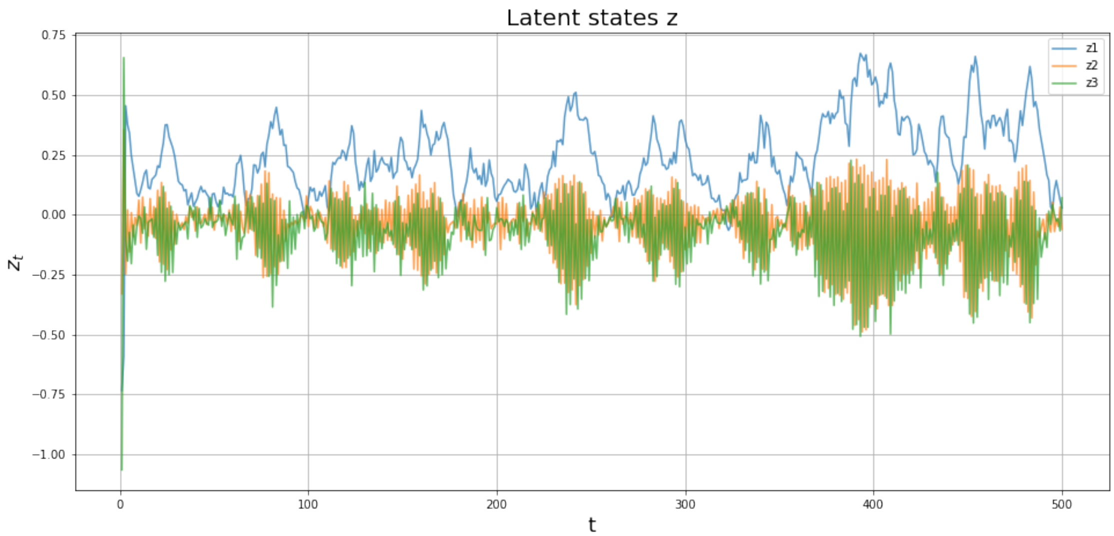
In [3]:
```
# latent state dimension
M = 3

# observation dimension
N = 10

# time steps
T = 500
```

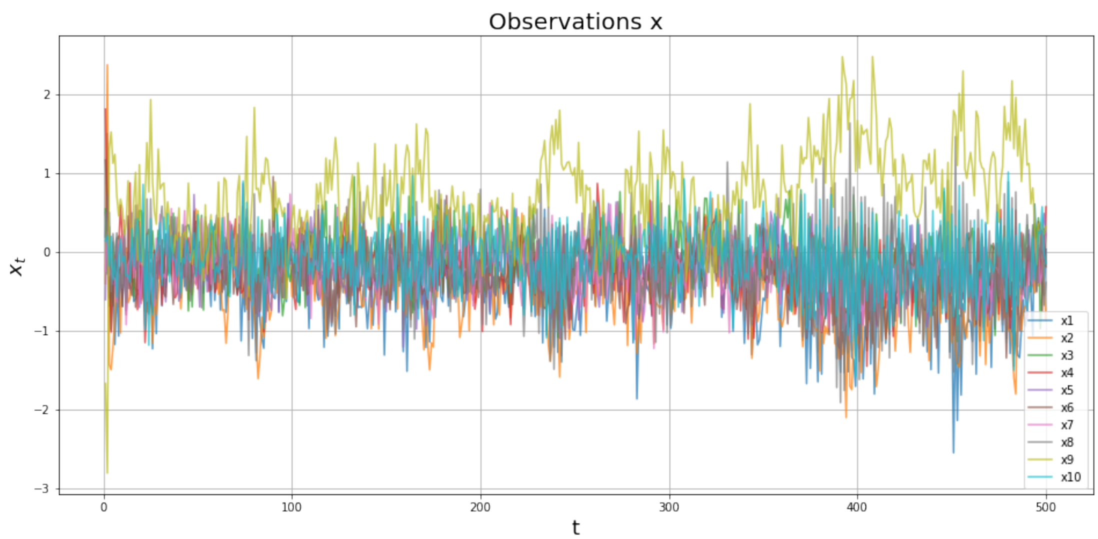
In [4]:
```
# relu function
def relu(z):
    return np.maximum(z,0)

# latent state evolution
def f(z, epsilon):
    return A@z + W@relu(z) + h + epsilon

# observation model
def g(z, eta):
    return B@z + eta

# time series generation
def generate(steps, mu0):
    # set random seed
    np.random.seed(1)

    # create empty arrays
    zt = np.zeros((M, steps))
    xt = np.zeros((N, steps))

    # set initial condition
    zt[:,0] = np.random.multivariate_normal(mu0.squeeze(), Sigma, 1).squeeze()
    etas0 = np.random.multivariate_normal(np.zeros(Gamma.shape[0]), Gamma, 1).T
    xt[:,0] = g(zt[:,0], etas0.T ).squeeze()
```

```
    # generate random numbers
    eps = np.random.multivariate_normal(np.zeros(Sigma.shape[0]), Sigma, steps-1).T
    etas = np.random.multivariate_normal(np.zeros(Gamma.shape[0]), Gamma, steps-1).T

    # loop over steps
    for t in range(1,steps):
        zt[:,t] = f( zt[:,t-1], eps[:,t-1].T ).squeeze()
        xt[:,t] = g( zt[:,t], etas[:,t-1].T ).squeeze()

    return zt, xt
```

In [5]:
```
# generate time series
z, x = generate(T, mu0)
```

In [6]:
```
fig1 = plt.subplots(figsize=[16,16])

# plot latent states
plt.subplot(2,1,1)

for i in range(z.shape[0]):
    plt.plot(np.arange(z.shape[1])+1, z[i], label='z%i' %(i+1), alpha=0.75 )
plt.xlabel('t', fontsize = 18)
plt.ylabel(r'$z_t$', fontsize = 18)
plt.grid()
plt.legend()
plt.title('Latent states z', fontsize = 20)

# plot observations
plt.subplot(2,1,2)
for i in range(x.shape[0]):
    plt.plot(np.arange(x.shape[1])+1, x[i], label='x%i' %(i+1), alpha=0.75 )
plt.xlabel('t', fontsize = 18)
plt.ylabel(r'$x_t$', fontsize = 18)
plt.grid()
plt.legend()
plt.title('Observations x', fontsize = 20)

plt.show(fig1)
```





1.0.2 b)

In [7]:
```
# run Kalman filter
from kalmanfilter import kalmanfilter

mu_kf, _, _ = kalmanfilter(x, mu0, Sigma, A, B, Gamma, Sigma)
```

1.0.3 c)

For this specific model we have:

$$\nabla_{t-1} = \frac{\partial g}{\partial f}(\mu_{t-1}) = B$$

$$J_{t-1} = \frac{\partial f}{\partial \mu_{t-1}}(\mu_{t-1}) = A + W\phi'(\mu_{t-1})$$

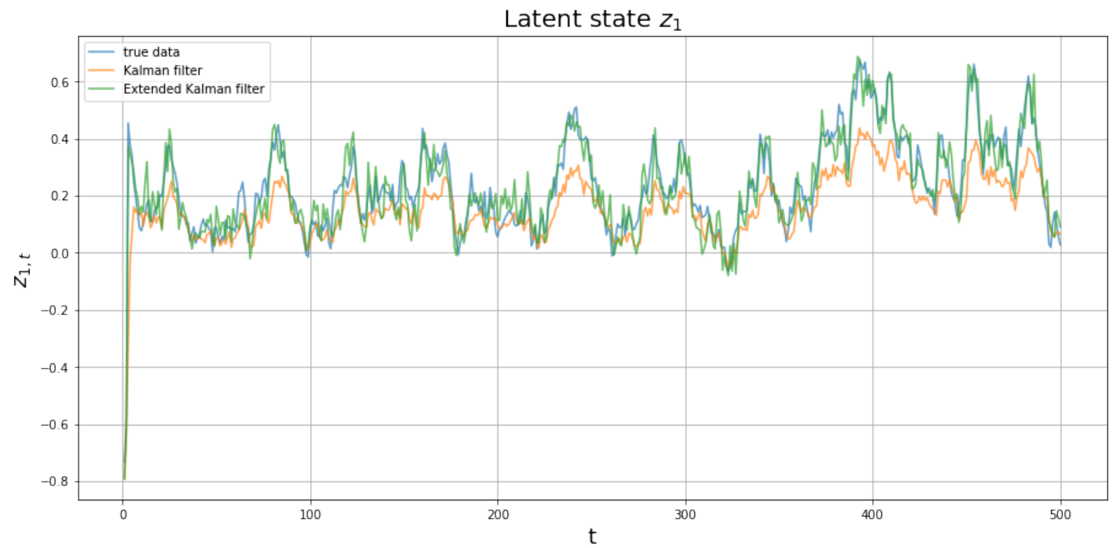
```
In [8]: # relu derivative function
        def d_relu(z):
            return (z > 0).astype(int)

        # define Extended Kalman Filter
        def ekf(x, mu0, L0, A, B, W, Gamma, Sigma, C=None, u=None):
            """
            :param x: observed variables (p x n, with p= number obs., n= number time steps)
            :param mu0: initial values
            :param L0: initial values
            :param A: transition matrix
            :param B: observation matrix
            :param Gamma: observation covariance
            :param Sigma: transition covariance
            :param C: control variable matrix
            :param u: control variables
            :return: mu, V, K
            """
            p = Sigma.shape[0]
            q = x.shape[0]
            n = x.shape[1]
            # are control variables entered? if not, set to 0
            if C is None and u is None:
                C = np.zeros((p, q))
                u = np.zeros((q, n))

            # initialize variables for filter and smoother
            L = np.zeros((p, p, n))  # measurement covariance matrix
            L[:, :, 0] = L0  # prior covariance
            mu_p = np.zeros((p, n))  # predicted expected value
            mu_p[:, 0] = np.squeeze(mu0)  # prior expected value
            mu = np.zeros((p, n))  # filter expected value
            V = np.zeros((p, p, n))  # filter covariance matrix
            K = np.zeros((p, q, n))  # Kalman Gain

            # first step
            K[:, :, 0] = L[:, :, 0] @ B.T @ np.linalg.inv(B @ L[:, :, 0] @ B.T + Gamma)  # Kalma
            mu[:, 0] = mu_p[:, 0] + K[:, :, 0] @ (x[:, 0] - g( mu_p[:, 0], np.zeros(q) ) )
            V[:, :, 0] = (np.eye(p) - K[:, :, 0] @ B) @ L[:, :, 0]

            # go forwards
            for t in range(1, n):
                J = A + W@d_relu(mu[:, t - 1])
                L[:, :, t] = J @ V[:, :, t - 1] @ J.T + Sigma
                K[:, :, t] = L[:, :, t] @ B.T @ np.linalg.inv(B @ L[:, :, t] @ B.T + Gamma)  # K
                mu_p[:, t] = f( mu[:, t - 1], np.zeros(p) ) # model prediction
                mu[:, t] = mu_p[:, t] + K[:, :, t] @ (x[:, t] - g( mu_p[:, t], np.zeros(q) ) ) #
                V[:, :, t] = (np.eye(p) - K[:, :, t] @ B) @ L[:, :, t]  # filtered covariance
```

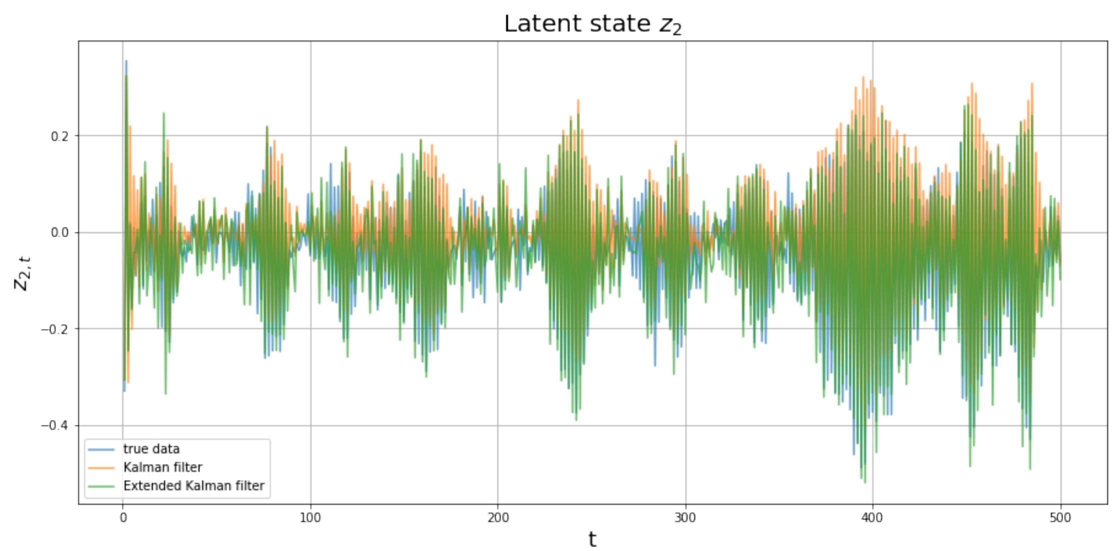
```
        return mu, V, K
```

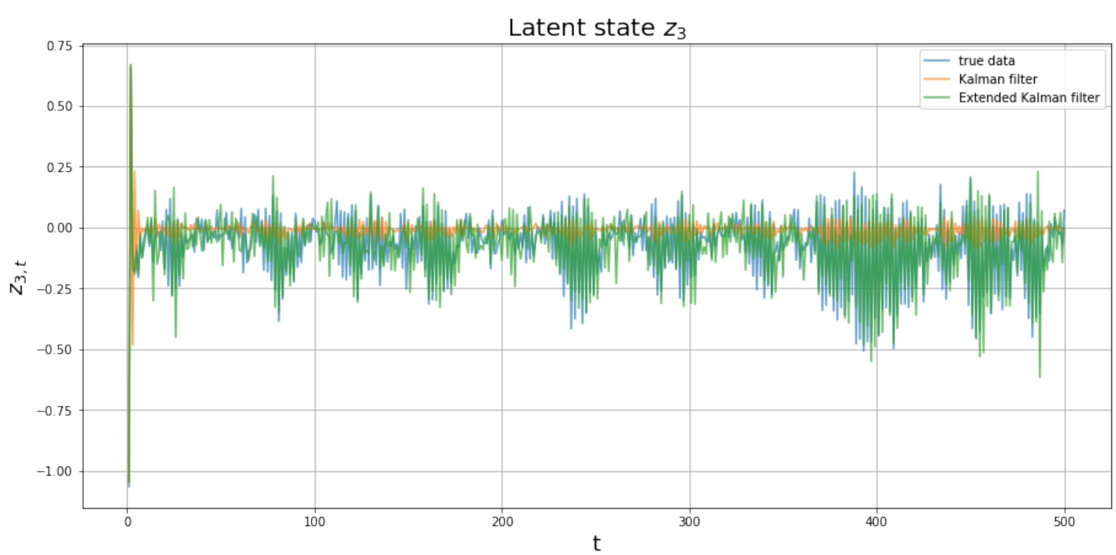
In [9]:
```
# run Extended Kalman filter
mu_ekf, _, _ = ekf(x, mu0, Sigma, A, B, W, Gamma, Sigma)
```

1.0.4 d)

In [10]:
```
fig1 = plt.subplots(figsize=[15,25])

# plot latent states' predictions vs. true data
for i in range(M):
    plt.subplot(M,1,i+1)
    plt.plot(np.arange(z.shape[1])+1, z[i], label='true data', alpha=0.7 )
    plt.plot(np.arange(z.shape[1])+1, mu_kf[i], label='Kalman filter', alpha=0.7 )
    plt.plot(np.arange(z.shape[1])+1, mu_ekf[i], label='Extended Kalman filter', alpha=
    plt.xlabel('t', fontsize = 18)
    plt.ylabel(r'$z_{%i,t}$' %(i+1), fontsize = 18)
    plt.grid()
    plt.legend()
    plt.title(r'Latent state $z_{%i}$' %(i+1), fontsize = 20)

plt.subplots_adjust(hspace = 0.3)
plt.show(fig1)
```


Latent state z_1
true data
Kalman filter
Extended Kalman filter
$z_{1,t}$
t


Latent state z_2
true data
Kalman filter
Extended Kalman filter
$z_{2,t}$
t


Latent state z_3
true data
Kalman filter
Extended Kalman filter
$z_{3,t}$
t

```
In [11]: # compute MSE
         mse_kf = np.mean( (mu_kf - z)**2 )
         mse_ekf = np.mean( (mu_ekf - z)**2 )

         print('MSE for Kalman Filter: {:.4f}'.format(mse_kf))
         print('MSE for Extended Kalman Filter: {:.4f}'.format(mse_ekf))
```

```
MSE for Kalman Filter: 0.0119
MSE for Extended Kalman Filter: 0.0045
```

As we can expected, the reconstructed signal obtained with the Extended Kalman Filter is better than the one obtained with the simple Kalman Filter: this statement is supported by the lower value of MSE.