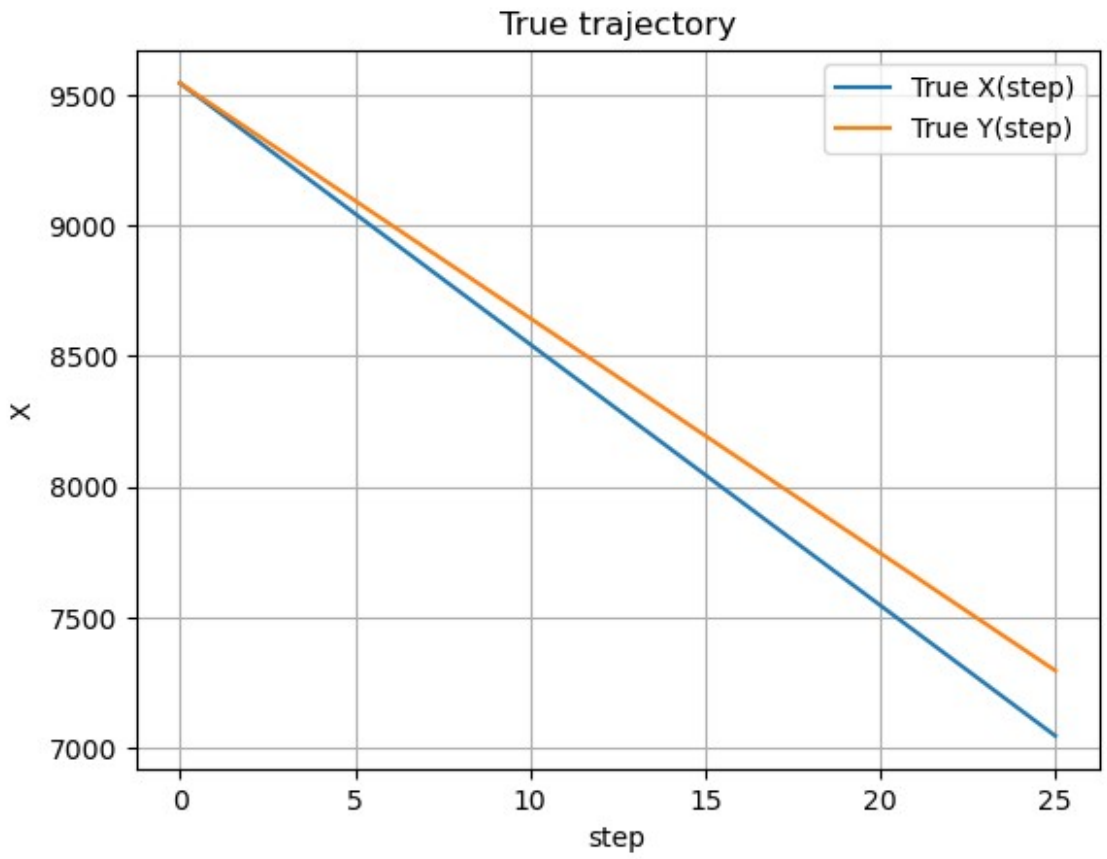
```
#Assignment 7
#Development of a tracking filter of a moving object when
measurements and motion
#models are in different coordinate systems
#Team 12
#Yaroslav Savotin, Elizaveta Pestova, Selamawit Asfaw
#Skoltech, 2023

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

#Function for genereting cartesian coordinates x,y and components of
velocity
def generate_Xt(x0,Fi,N):
    Xt = np.zeros((4, 1, N))
    Xt[:, :, 0] = x0
    for i in range(1, N):
        Xt[:, :, i] = np.dot(Fi,Xt[:, :, i-1])
    return Xt

#Function used for genereting polar trajectory
def generate_Polar_Xt(Xt, N):
    D = np.zeros((1, 1, N))
    Beta = np.zeros((1, 1, N))
    for i in range(N):
        D[:, :, i] = np.sqrt(Xt[0, 0, i]**2 + Xt[2, 0, i]**2)
        Beta[:, :, i] = np.arctan(Xt[0, 0, i] / Xt[2, 0, i])
    return D, Beta


#Function used for genereting polar measuement
def generate_Polar_Z(sigmaD, sigmaB, D, B, N):
    Dz = np.zeros((1, 1, N))
    Bz = np.zeros((1, 1, N))
    nuD = np.random.normal(0, sigmaD, (1, 1, N))
    nuB = np.random.normal(0, sigmaB, (1, 1, N))
    for i in range(N):
        Dz[:, :, i] = D[:, :, i] + nuD[:, :, i]
        Bz[:, :, i] = B[:, :, i] + nuB[:, :, i]
    return Dz, Bz

#Function for coordinate transformation(polar to cartesian)
def pseudo_measurement(Dz, Bz, N):
    Xm = np.zeros((1, 1, N))
    Ym = np.zeros((1, 1, N))
    Zm = np.zeros((2, 1, N))
    for i in range(N):
        Xm[:, :, i] = Dz[:, :, i] * np.sin(Bz[:, :, i])
        Ym[:, :, i] = Dz[:, :, i] * np.cos(Bz[:, :, i])
    for i in range(N):
        Zm[0,0,i] = Xm[0][0][i]
        Zm[1,0,i] = Ym[0][0][i]
```

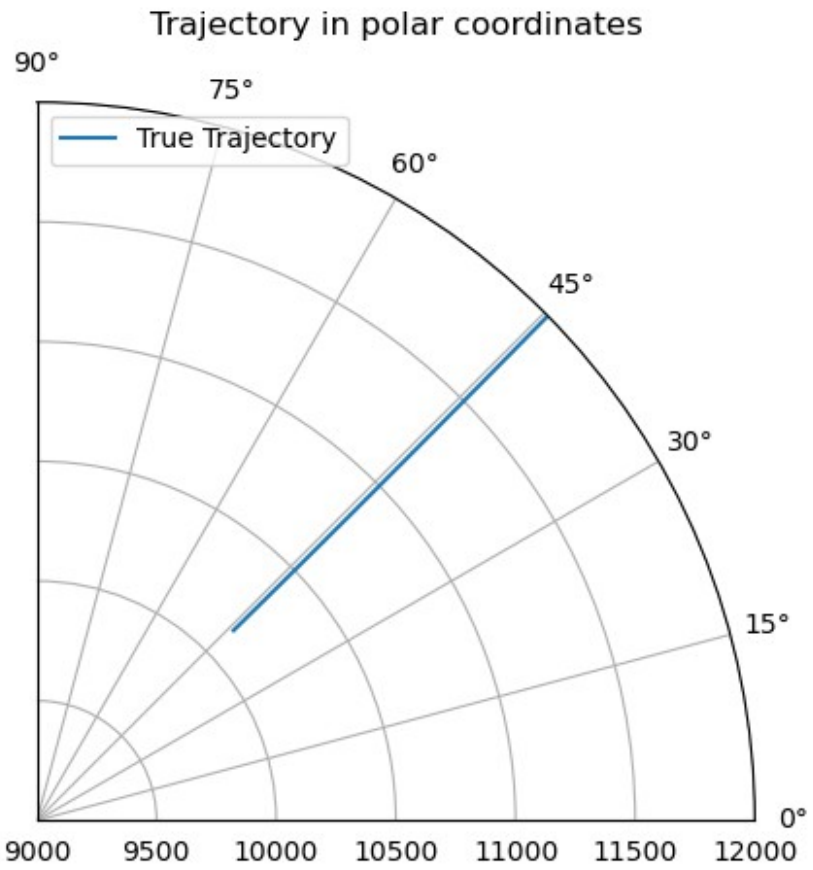
```
    return Zm


#Function to calculate R matrix
def Covariance_matrix(sigmaD, sigmaB, Zm, N):
    R = np.zeros((2,2,N))
    for i in range(N):
        R[0,0,i] = ((sigmaD**2)*np.sin(Zm[1,:,i])**2) +
(Zm[0,:,i]**2)*(np.cos(Zm[1,:,i])**2)*(sigmaB**2)
        R[0,1,i] = np.sin(Zm[1,:,i])*np.cos(Zm[1,:,i])*((sigmaD**2)
- (Zm[0,:,i]**2)*sigmaB**2)
        R[1,0,i] = np.sin(Zm[1,:,i])*np.cos(Zm[1,:,i])*((sigmaD**2)
- (Zm[0,:,i]**2)*sigmaB**2)
        R[1,1,i] = (np.cos(Zm[1,0,i])**2)*sigmaD**2 +
(Zm[0,:,i]**2)*(np.sin(Zm[1,:,i])**2)*sigmaB**2
    return R


#Kalman filter
def kalmanFilter_from_polar_Z(Fi, H, R, X0, P0, Zm, N):
    """
  Inputs:
  Fi: Transition matrix
  H: Observation matrix
  R: covariance matrix
  X0: initial velocity of trajectory
  P0: Initial filtration error covariance matrix
  Zm: measurment vector
  N: sampling period
  """
    Xp = np.zeros((4, 1, N))
    Pf = np.zeros((4, 4, N))
    Pp = np.zeros((4, 4, N))
    Xf = np.zeros((4, 1, N))
    K = np.zeros((4, 2, N))
    Ps = np.zeros((4, 1, N))
    Df = np.zeros((1, 1, N))
    Betaf = np.zeros((1, 1, N))
    Dp = np.zeros((1, 1, N))
    Betap = np.zeros((1, 1, N))
    Pf[:, :, 0] = P0
    Xf[:, :, 0] = X0

    for i in range(1, N):
        Xp[:, :, i] = np.dot(Fi, Xf[:, :, i-1])
        Pp[:, :, i] = np.dot(np.dot(Fi, Pf[:, :, i-1]),
np.transpose(Fi))
        K[:, :, i] = np.dot(np.dot(Pp[:, :, i], np.transpose(H)),
np.linalg.inv(np.dot(np.dot(H, Pp[:, :, i]), np.transpose(H)) + R[:,
:, i]))
        Xf[:, :, i] = Xp[:, :, i] + np.dot(K[:, :, i], (Zm[:, :, i]
- np.dot(H, Xp[:, :, i])))
        Pf[:, :, i] = np.dot((np.eye(4) - np.dot(K[:, :, i], H)),
Pp[:, :, i])
```

```
        Ps[:, :, i] = np.sqrt(np.abs(np.diag(Pf[:, :, i])[0]))

        Df[:, :, i] = np.sqrt(Xf[0, 0, i]**2 + Xf[2, 0, i]**2)
        Betaf[:, :, i] = np.arctan(Xf[0, 0, i] / Xf[2, 0, i])

        Dp[:, :, i] = np.sqrt(Xp[0, 0, i]**2 + Xp[2, 0, i]**2)
        Betap[:, :, i] = np.arctan(Xp[0, 0, i] / Xf[2, 0, i])

    return Xf, K, Dp, Betap, Df, Betaf


#Function that calculate MSE over M run of the Kalman filter

def calculate_MSE(N, M, x0, sigmaD, sigmaB, Fi, H, R, X0, P0):
    Error_run = np.zeros((2, M, N))
    final_Error = np.zeros((1, 2, N))
    Error_run1 = np.zeros((2, M, N))
    final_Error1 = np.zeros((1, 2, N))
    Xf = np.zeros((4, 1, N))
    K = np.zeros((4, 2, N))
    Pot = np.zeros((2, 1, N))
    Pof = np.zeros((2, 1, N))
    Pop = np.zeros((2, 1, N))

    for i in range(M):
        Xt = generate_Xt(x0, Fi, N)
        D, Beta = generate_Polar_Xt(Xt, N)
        Dz, Bz = generate_Polar_Z(sigmaD, sigmaB, D, Beta, N)
        Zm = pseudo_measurement(Dz, Bz, N)
        Xf, K, Dp, Betap, Df, Betaf = kalmanFilter_from_polar_Z(Fi,
H, R, X0, P0, Zm, N)
        for i in range(N):
            Pot[0,0,i] = D[0][0][i]
            Pot[1,0,i] = Beta[0][0][i]

            Pof[0,0,i] = Df[0][0][i]
            Pof[1,0,i] = Betaf[0][0][i]

            Pop[0,0,i] = Dp[0][0][i]
            Pop[1,0,i] = Betap[0][0][i]

        for j in range(N):
            Error_run[:, i, j] = ((Pot[:, :, j] - Pof[:, :, j]) **
2).T
            Error_run1[:, i, j] = ((Pot[:, :, j] - Pop[:, :, j]) **
2).T

    for i in range(N):
        for k in range(M):
            final_Error[:, :, i] += Error_run[:, k, i]
            final_Error1[:, :, i] += Error_run1[:, k, i]

        final_Error[:, :, i] = np.sqrt(final_Error[:, :, i] / (M -
```

```
1))
        final_Error1[:, :, i] = np.sqrt(final_Error1[:, :, i] / (M -
1))

    return final_Error, final_Error1

#Step1 Generate the true trajectory of an object

# Initial parameters
N = 26
step = np.arange(N)
T = 2

# Initial position and velocity
#[x0,Vx0,y0,Vy0]
x0 = np.array([[13500/np.sqrt(2)], [-50], [13500/np.sqrt(2)], [-
45]])


#Step 7 Generate transition and observation matrix
# Transition matrix
Fi = np.array([[1, T, 0, 0],
               [0, 1, 0, 0],
               [0, 0, 1, T],
               [0, 0, 0, 1]])
# Observation matrix
H = np.array([[1, 0, 0, 0],
              [0, 0, 1, 0]])

# Generate true trajectories
Xt = generate_Xt(x0, Fi, N)

# Plot
plt.figure()
plt.plot(step, Xt[0][0], label='True X(step)')
plt.plot(step, Xt[2][0], label='True Y(step)')
plt.title('True trajectory')
plt.xlabel('step')
plt.ylabel('X')
plt.legend()
plt.grid(True)
plt.show()
```

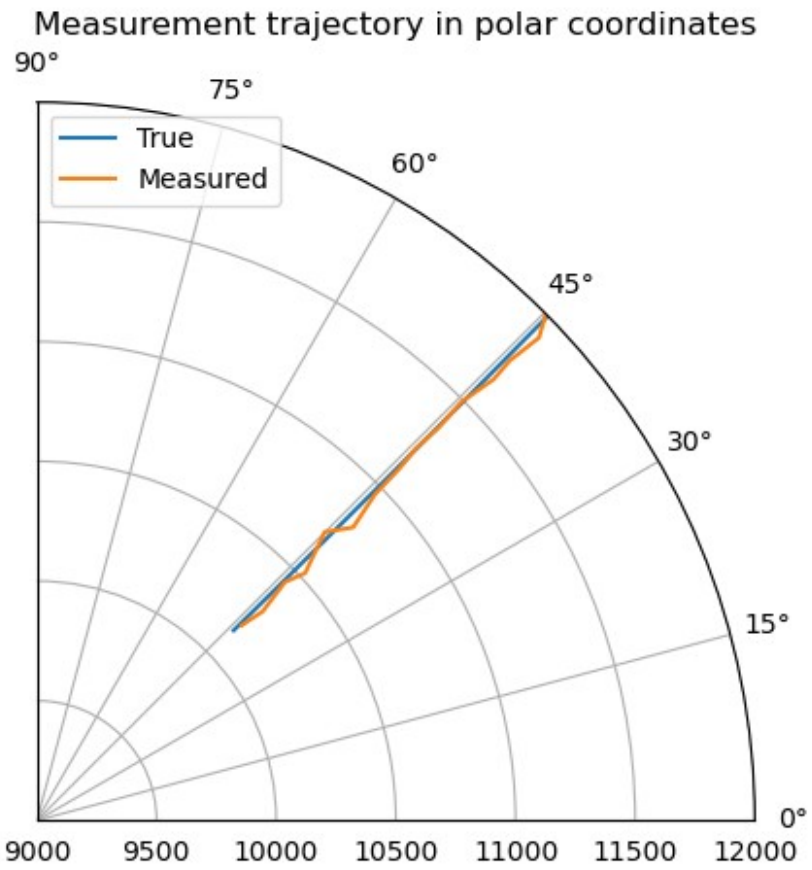



```
#Step 2 generation of true values of range D and azimuth Beta
D, Beta = generate_Polar_Xt(Xt, N)


# Plot
ax = plt.subplot(111, projection='polar')
ax.plot(Beta[0][0], D[0][0], label='True Trajectory')
ax.set_title('Trajectory in polar coordinates')
ax.set_xlim(0,np.radians(90))
ax.set_ylim(9000,12000)
ax.legend()

plt.show()
```

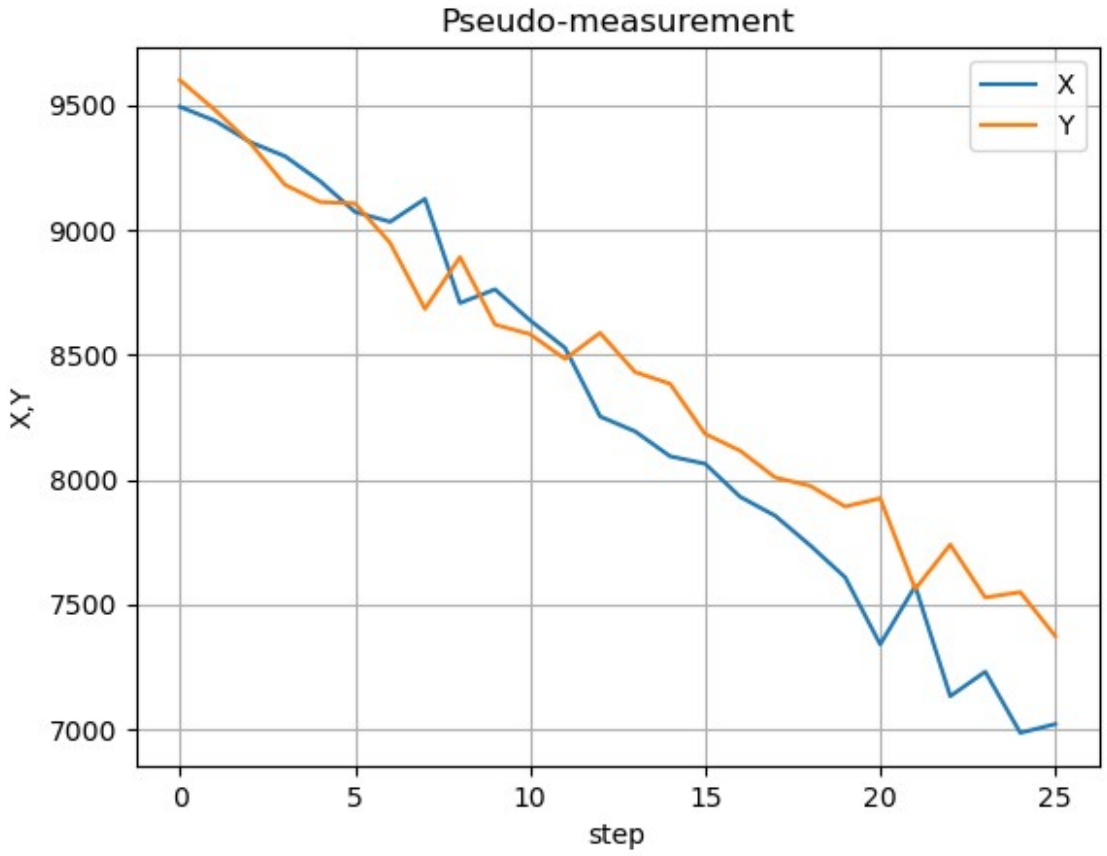


The generated trajectory corresponds to the cartesian generated coordinates (x and y are almost proportional meaning the angle of the trayectory is almost 45 degree, the starting point is 13500 as expected)

```
#-Step 3 Generate measurements data of range D and azimuth Beta

#Variances of measurement noises
sigmaD = 20
sigmaB = 0.02
Dz, BetaZ = generate_Polar_Z(sigmaD, sigmaB, D, Beta, N)

# Plot
ax = plt.subplot(111, projection='polar')
ax.plot(Beta[0][0],D[0][0], label='True')
ax.plot(BetaZ[0][0], Dz[0][0], label='Measured')
ax.set_title('Measurement trajectory in polar coordinates')
ax.set_xlim(0,np.radians(90))
ax.set_ylim(9000,12000)
ax.legend()
ax.grid(True)

plt.show()
```

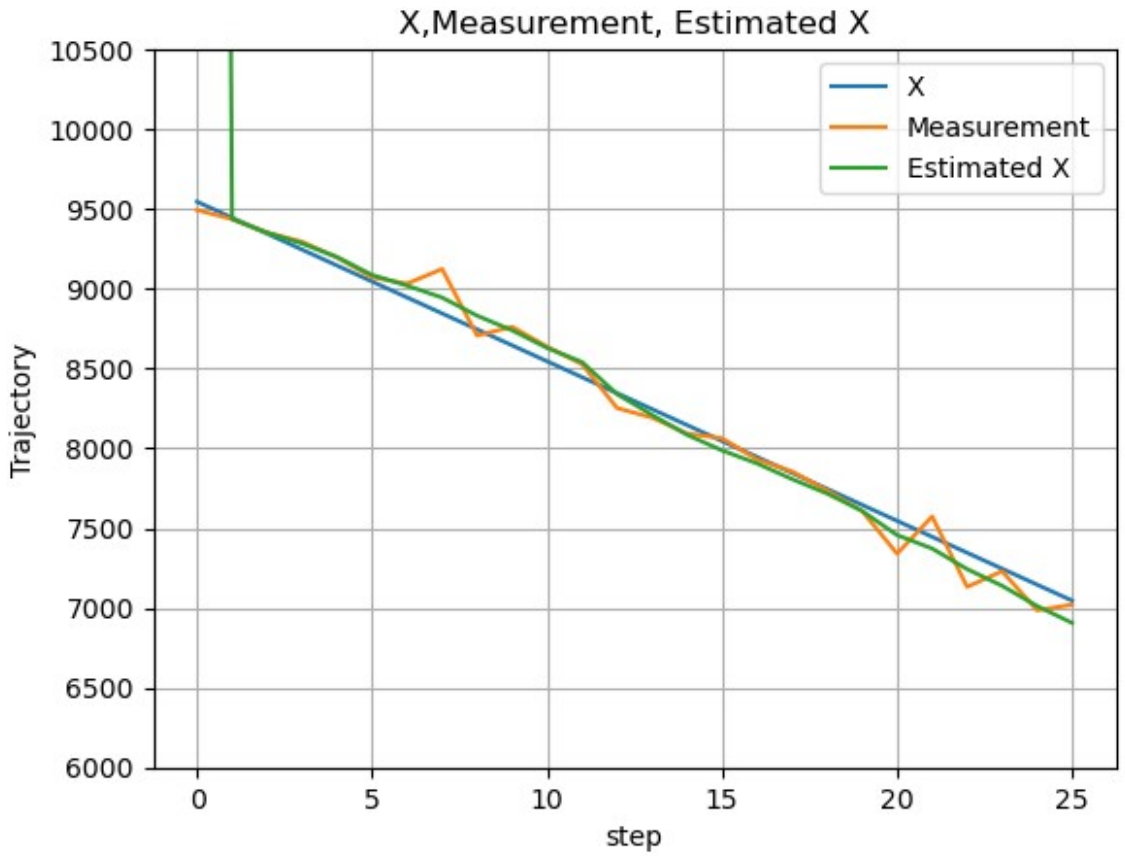



```
#Step 4,5 Transform polar coordinates of measurments of range D and
azimuth Beta to Cartesian ones. Get get
#pseudo-measurements of coordinates x, y. Create a measurment vector
z from pseudo-measurements of coordinates x, y.

Zm = pseudo_measurement(Dz, BetaZ, N)

# Plot
plt.plot(step, Zm[0][0], label='X')
plt.plot(step, Zm[1][0], label='Y')
plt.title('Pseudo-measurement')
plt.xlabel('step')
plt.ylabel('X,Y')
plt.legend()
plt.grid(True)
plt.show()
```

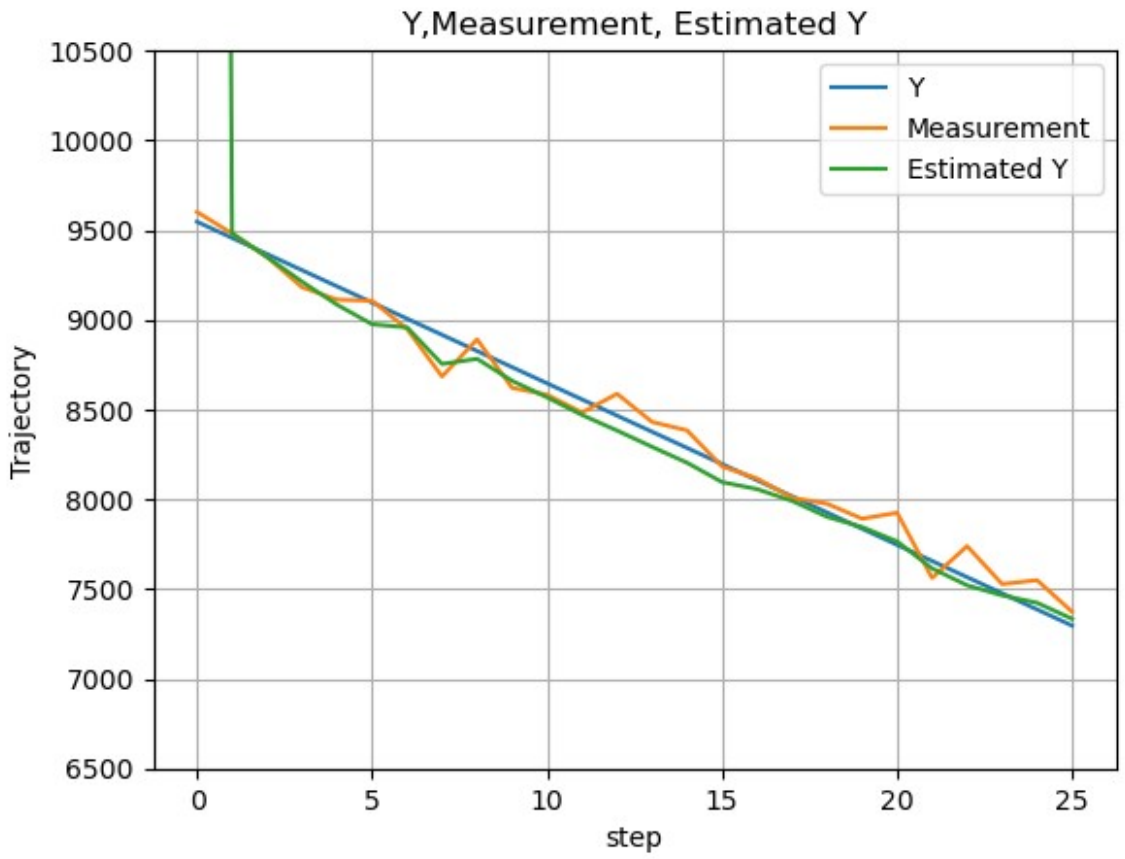



```
#Step 6 Initial conditions for Kalman filter algorithm
X0 = np.array([[40000],[-20],[40000],[-20]])
P0 = np.array([[10**10, 0, 0, 0],
               [0, 10**10, 0, 0],
               [0, 0, 10**10, 0],
               [0, 0, 0, 10**10]])

#Step 7 Is done earlier, near step 1

#Step 8 Create measurement error covariance matrix
#Measurement error covariance matrix
R = Covariance_matrix(sigmaD,sigmaB,Zm,N)

#Step 9 Develop Kalman algorithm to estimate state vector xi
(extrapolation and filtration)
Xf, K, Db, Betab, Df, Betaf = kalmanFilter_from_polar_Z(Fi, H, R,
X0, P0, Zm, N)

# Plot of the measured, true and filtered X
plt.plot(step, Xt[0][0], label='X')
plt.plot(step, Zm[0][0], label='Measurement')
plt.plot(step, Xf[0][0], label='Estimated X')
plt.title('X,Measurement, Estimated X')
plt.xlabel('step')
plt.ylabel('Trajectory')
plt.ylim(6000,10500)
plt.grid(True)
plt.legend()
```

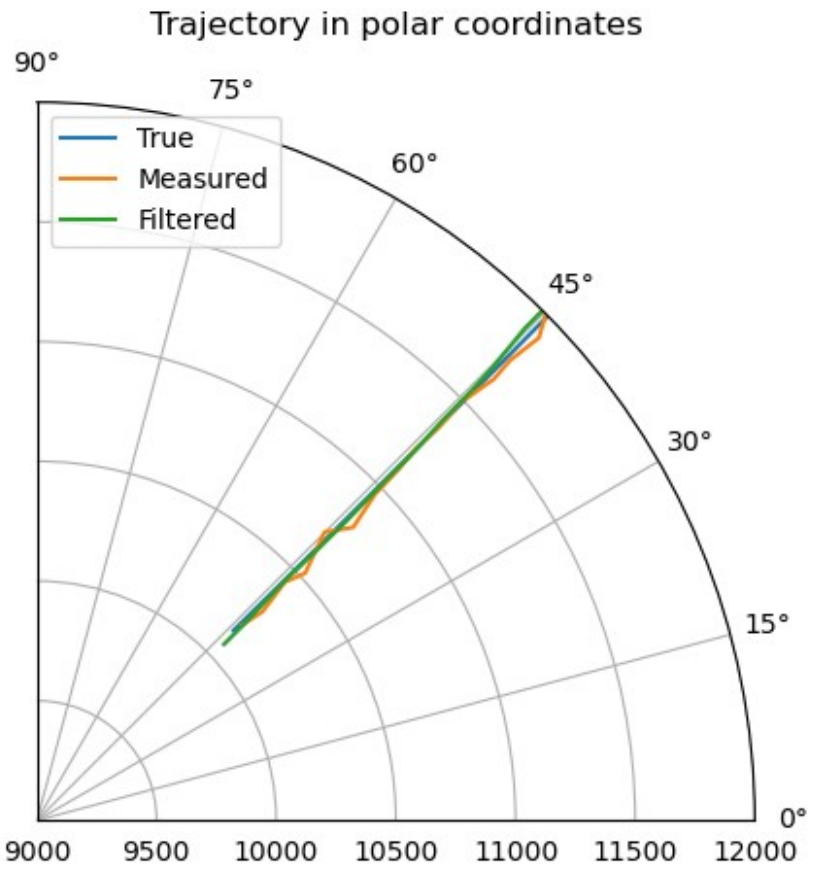
```
<matplotlib.legend.Legend at 0x21b8efdf110>
```



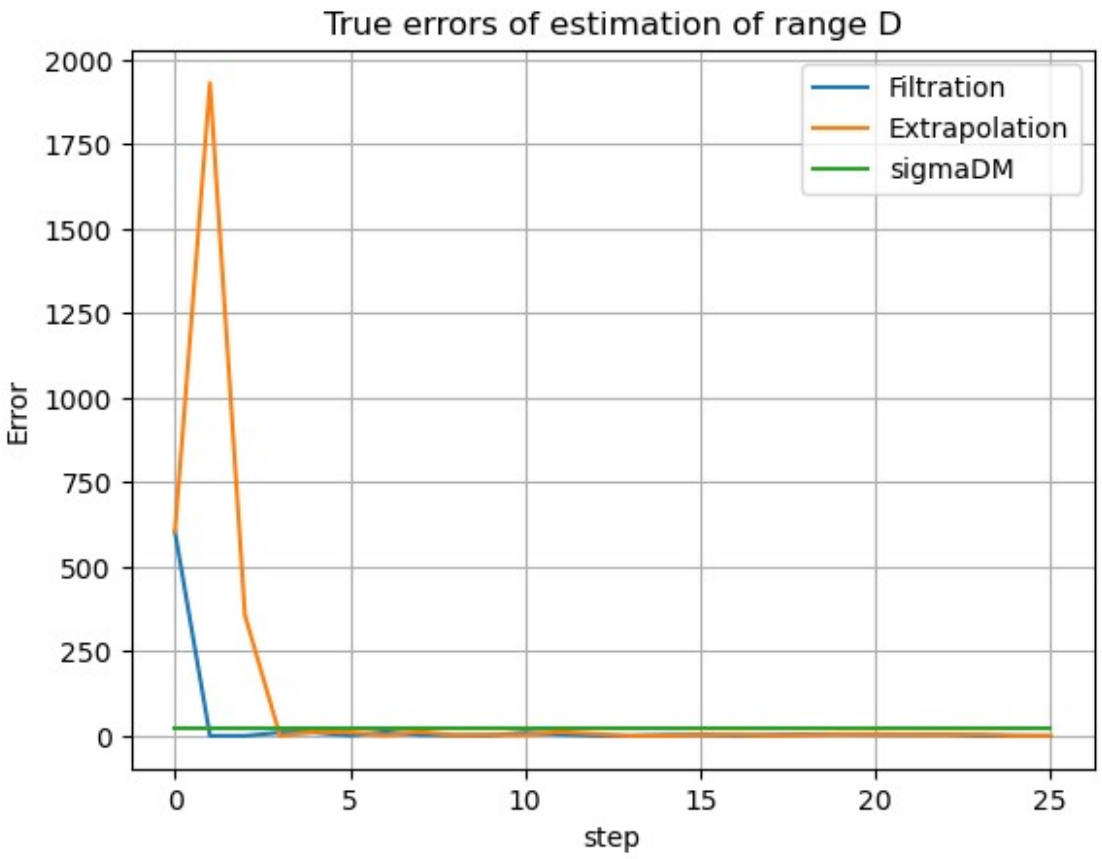

```
# Plot of the measured, true and filtered Y
plt.plot(step, Xt[2][0], label='Y')
plt.plot(step, Zm[1][0], label='Measurement')
plt.plot(step, Xf[2][0], label='Estimated Y')
plt.title('Y,Measurement, Estimated Y')
plt.xlabel('step')
plt.ylabel('Trajectory')
plt.ylim(6500,10500)
plt.grid(True)
plt.legend()

<matplotlib.legend.Legend at 0x21b8f155b90>
```



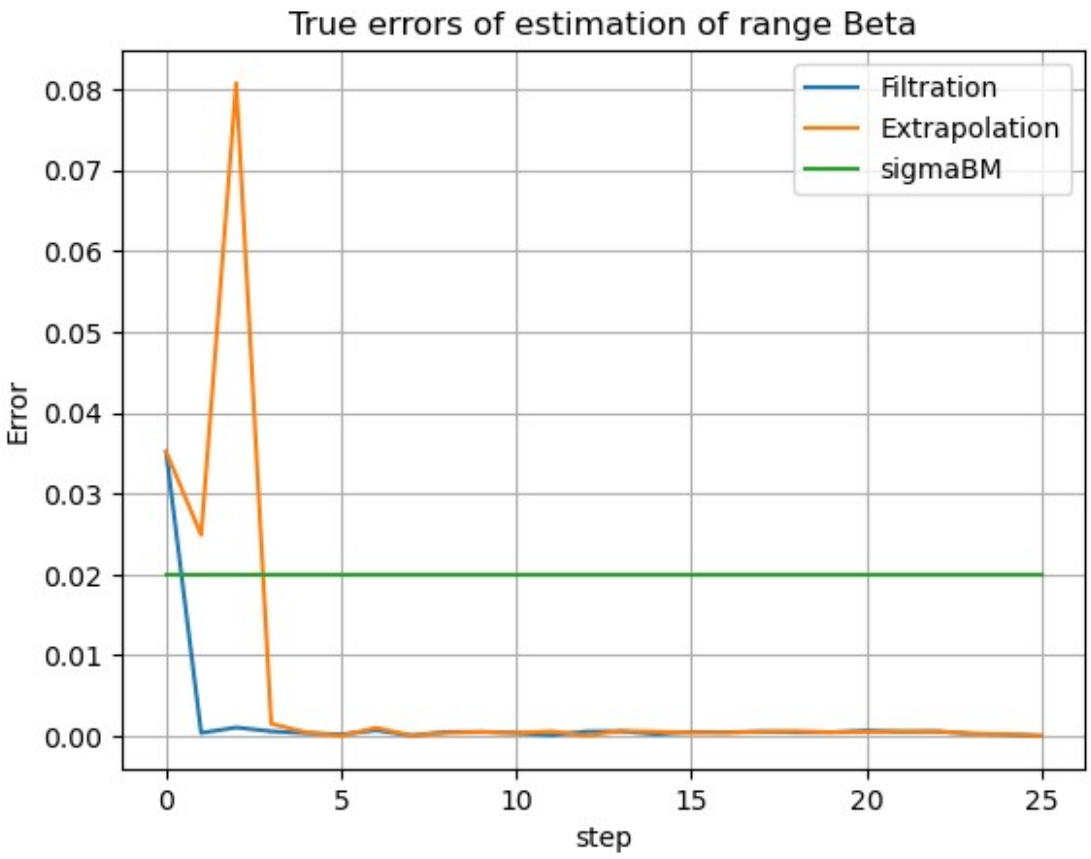

```
# Plot of the measured, true and filtered trajectories in polar coordinates

ax = plt.subplot(111, projection='polar')
ax.plot(Beta[0][0],D[0][0], label='True')
ax.plot(BetaZ[0][0], Dz[0][0], label='Measured')
ax.plot(Betab[0][0], Db[0][0], label='Filtered')
ax.set_title('Trajectory in polar coordinates')
ax.set_xlim(0,np.radians(90))
ax.set_ylim(9000,12000)
ax.legend()

plt.show()
```



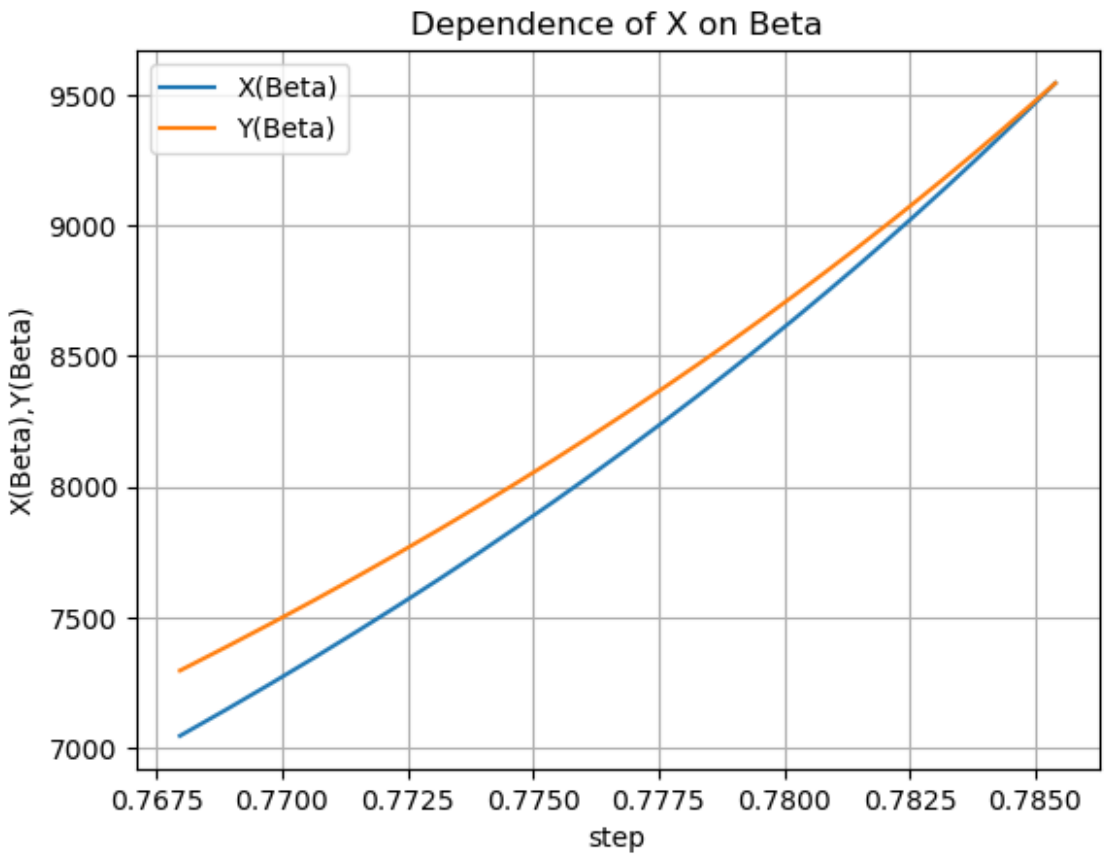

```
#Run Kalman filer 500 times
M = 500
sigmaDM = sigmaD * np.ones(N)
sigmaBM = sigmaB * np.ones(N)

# #mean-squared error of estimation of D, beta
MSE1, MSE2 = calculate_MSE(N, M, x0, sigmaD, sigmaB, Fi, H, R, X0,
P0)

# Plot True errors of filtration, extrapolation steps and sigma Dm
plt.plot(step, MSE1[0][0], label='Filtration')
plt.plot(step, MSE2[0][0], label='Extrapolation')
plt.plot(step, sigmaDM, label='sigmaDM')
plt.title('True errors of estimation of range D')
plt.xlabel('step')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```



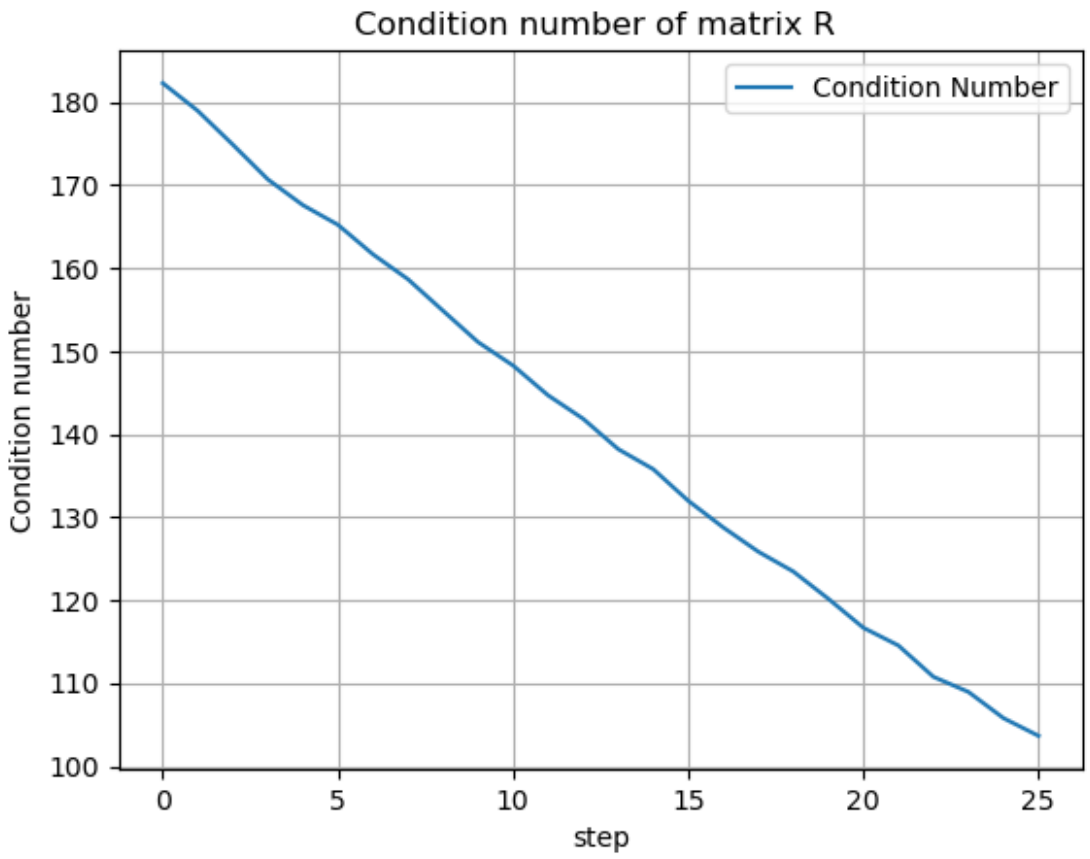

```
# Plot mean-squared error of filtration, extrapolation steps and
sigma beta
plt.plot(step, MSE1[0][1], label='Filtration')
plt.plot(step, MSE2[0][1], label='Extrapolation')
plt.plot(step, sigmaBM, label='sigmaBM')
plt.title('True errors of estimation of range Beta')
plt.xlabel('step')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```




```
#Dependence of X on Beta shown at plot
plt.plot(Beta[0][0], Xt[0][0], label='X(Beta)')
plt.plot(Beta[0][0], Xt[2][0], label='Y(Beta)')
plt.title('Dependence of X on Beta')
plt.xlabel('step')
plt.ylabel('X(Beta),Y(Beta)')
plt.legend()
plt.grid(True)
plt.show()
```

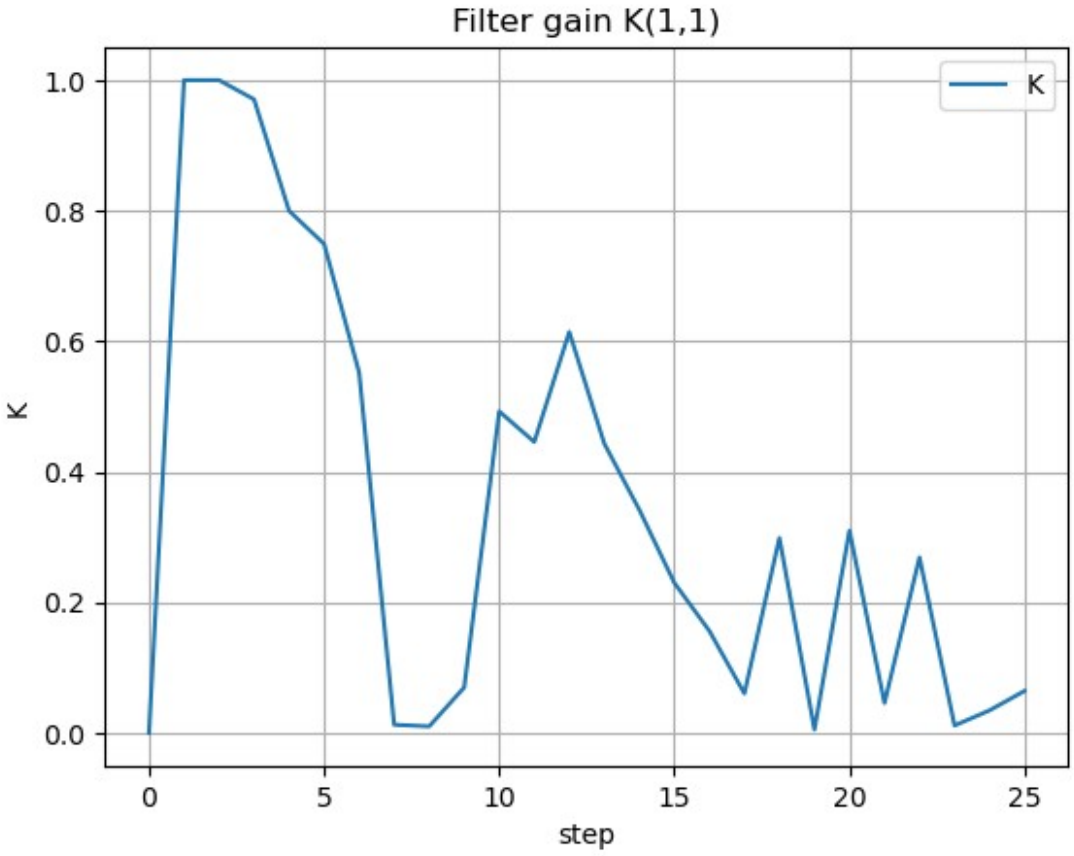



```
#Step 12 Calculation of the condition number of covariance matrix R
#np.linalg.eig(R[:, :, i])[0] - eigenvalues

#for i in range(N):
#    print(np.linalg.eig(R[:, :, i])[0])


ConN1 = np.zeros((2, 1, N))
ConN2 = np.zeros((2, 1, N))
for i in range(N):
    ConN1[:, :, i] = (sigmaD**2)/((Dz[:, :, i]**2)*sigmaB**2)
    ConN2[:, :, i] = ((Dz[:, :, i]**2)*sigmaB**2)/(sigmaD**2)
plt.plot(step, ConN2[0][0], label='Condition Number')
plt.title("Condition number of matrix R ")
plt.xlabel('step')
plt.ylabel('Condition number')
plt.legend()
plt.grid(True)
plt.show()
```

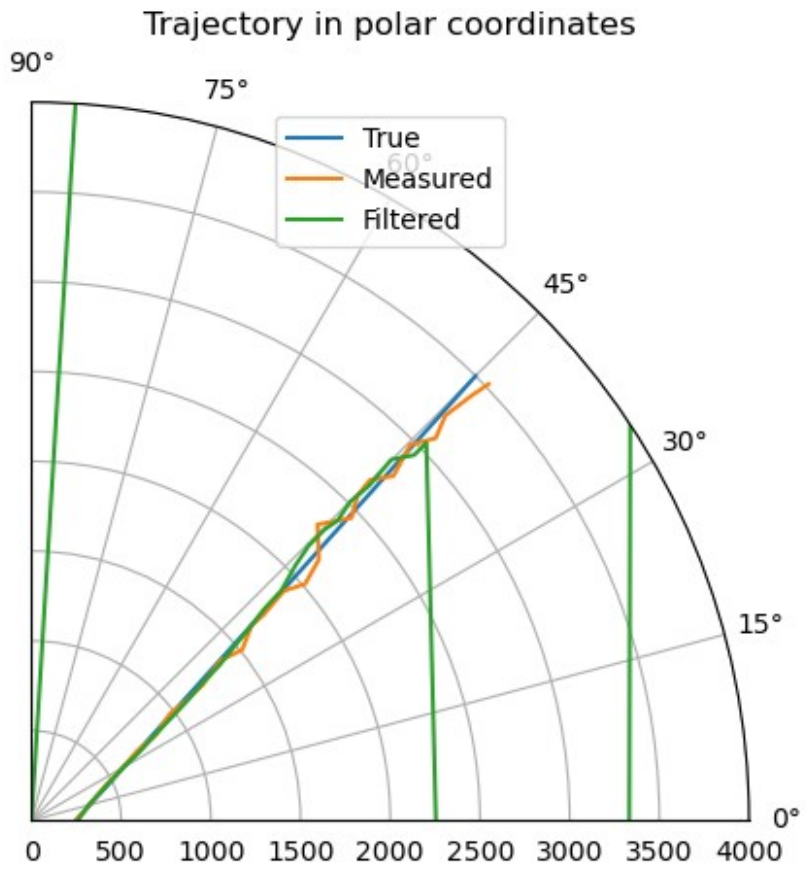


The condition number for every measurement is lower than 200, meaning the R matrix is not ill-conditioned, but the result shows that the conditions are also far from optimal. We can observe how the filter gives better results over time.

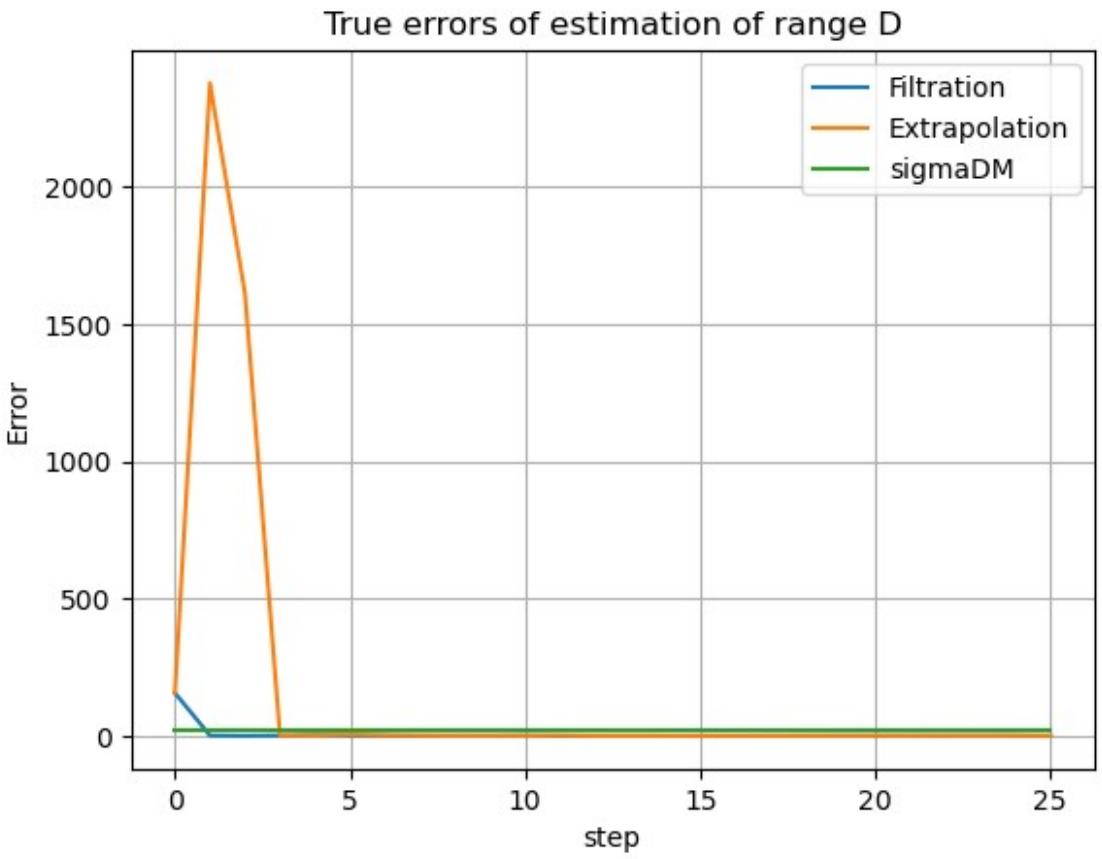
```
#Step 13. Filter gain analysis

# Plot
plt.plot(step, K[0][0], label='K')
plt.title('Filter gain K(1,1)')
plt.xlabel('step')
plt.ylabel('K')
plt.legend()
plt.grid(True)
plt.show()
```



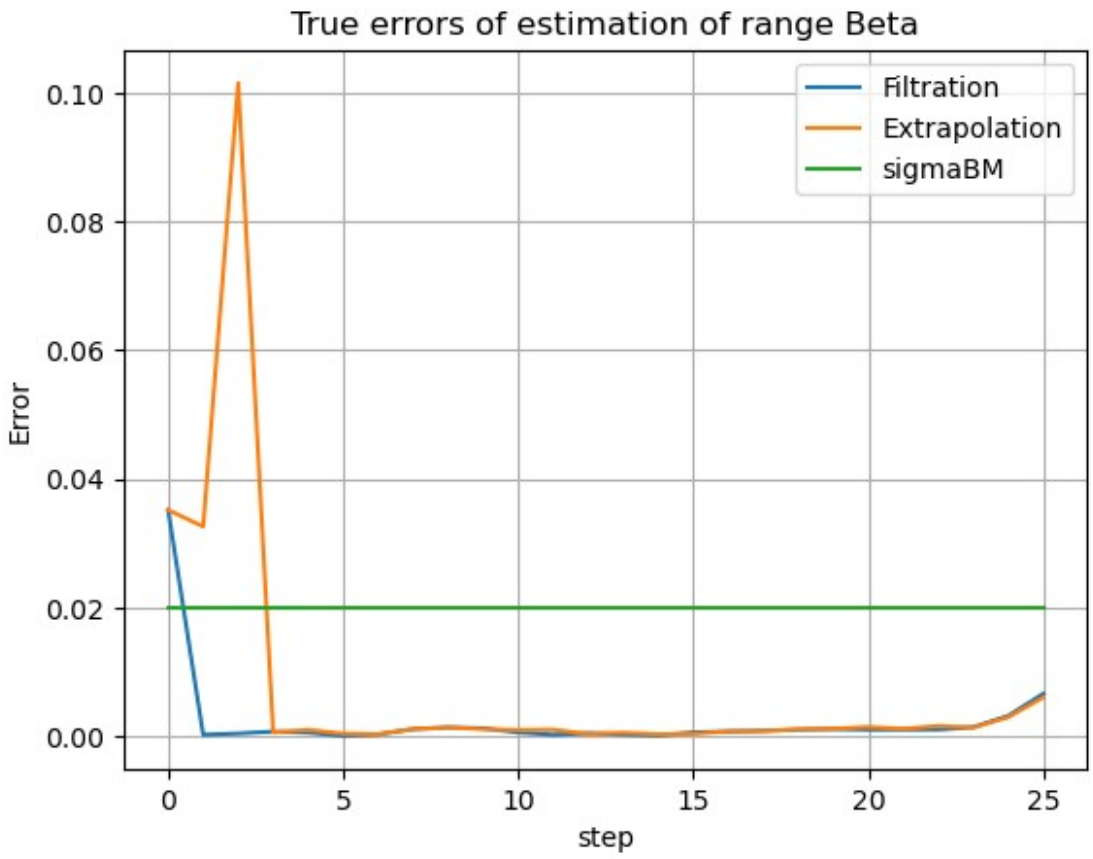

```
#Step 14. 500 runs with different initial conditions

x01 = np.array([[3500/np.sqrt(2)], [-50], [3500/np.sqrt(2)], [-45]])

Xt1 = generate_Xt(x01, Fi, N)
D1, Beta1 = generate_Polar_Xt(Xt1, N)
Dz1, BetaZ1 = generate_Polar_Z(sigmaD, sigmaB, D1, Beta1, N)
Zm1 = pseudo_measurement(Dz1, BetaZ1, N)
R1 = Covariance_matrix(sigmaD,sigmaB,Zm1,N)
Xf1, K1, Db1, Betab1, Df1, Betaf1 = kalmanFilter_from_polar_Z(Fi, H,
R1, X0, P0, Zm1, N)

#Plot polar
ax = plt.subplot(111, projection='polar')
ax.plot(Beta1[0][0],D1[0][0], label='True')
ax.plot(BetaZ1[0][0], Dz1[0][0], label='Measured')
ax.plot(Betab1[0][0], Db1[0][0], label='Filtered')
ax.set_title('Trajectory in polar coordinates')
ax.set_xlim(0,np.radians(90))
ax.set_ylim(0,4000)
ax.legend()

plt.show()
```



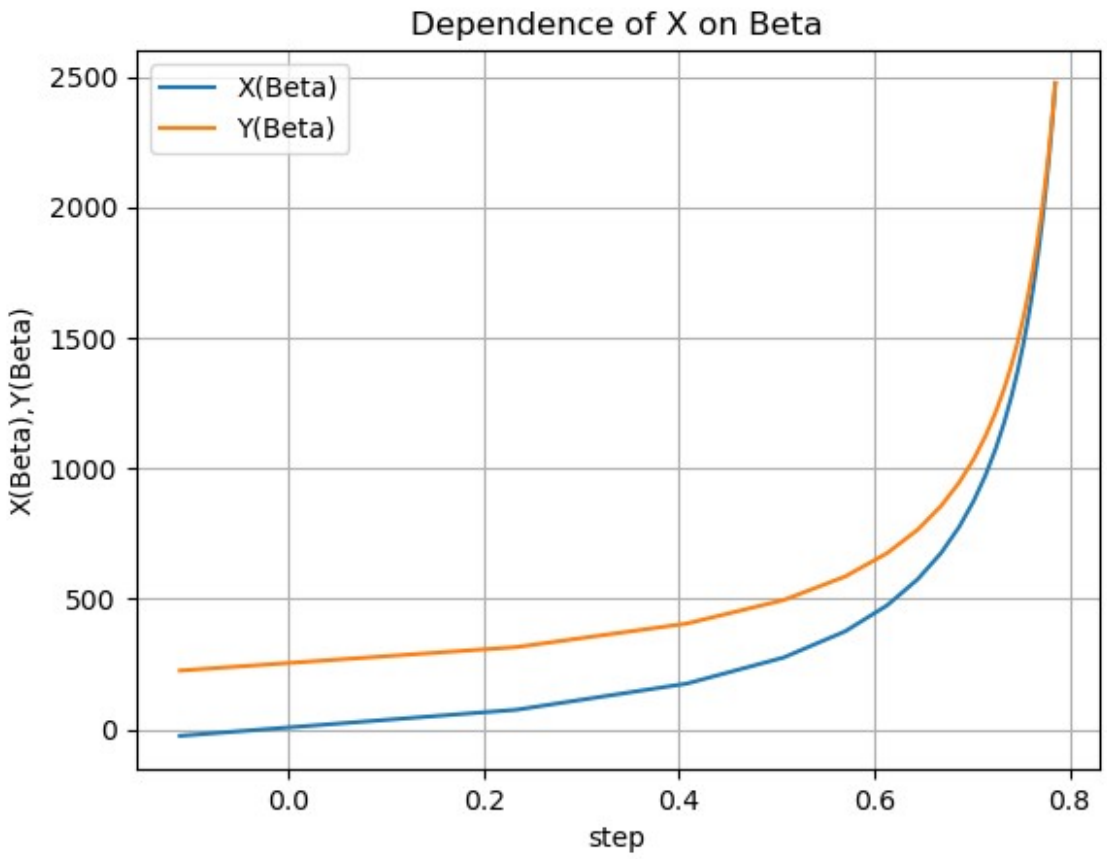

```
#Step 15. Calculation of errors of estimation.
#mean-squared error
MSE1_1, MSE2_1 = calculate_MSE(N, M, x01, sigmaD, sigmaB, Fi, H, R,
X0, P0)

# Plot true errors of estimation of range D at filtration and
extrapolation
plt.plot(step, MSE1_1[0][0], label='Filtration')
plt.plot(step, MSE2_1[0][0], label='Extrapolation')
plt.plot(step, sigmaDM, label='sigmaDM')
plt.title('True errors of estimation of range D')
plt.xlabel('step')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```



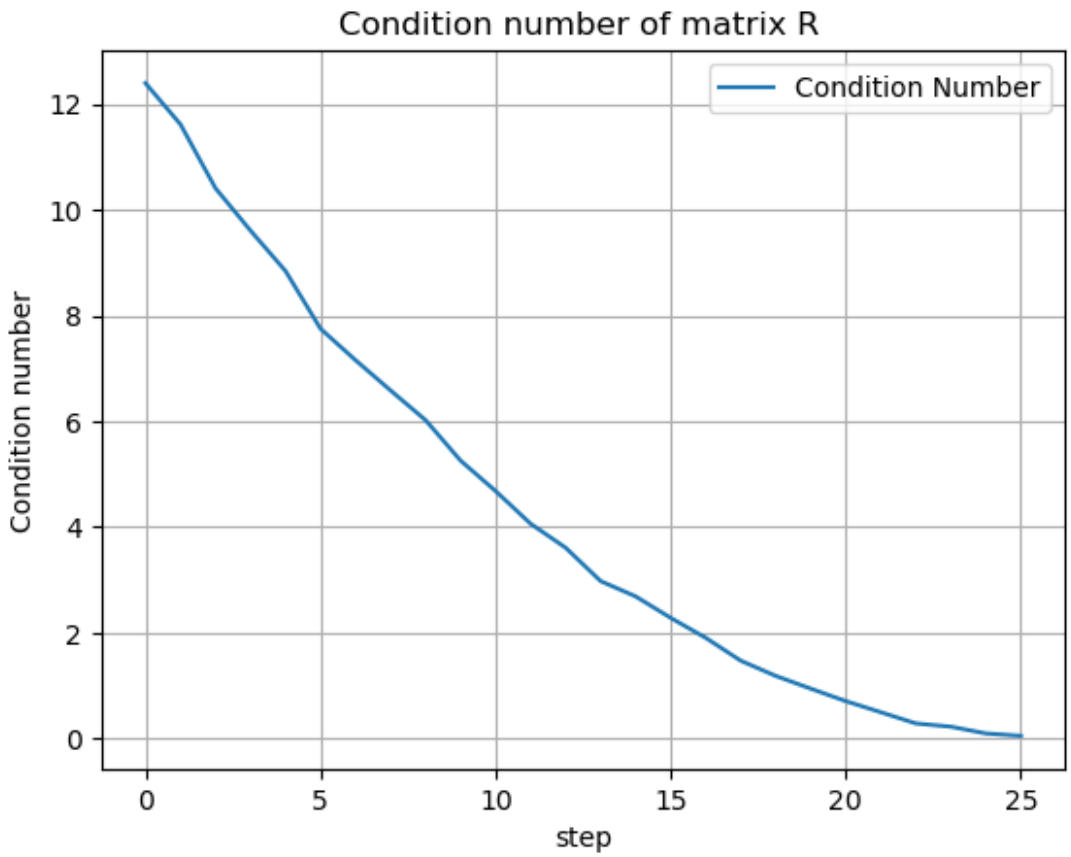

```
# Plot true errors of estimation of range Beta at filtration and
extrapolation
#mean-squared error
plt.plot(step, MSE1_1[0][1], label='Filtration')
plt.plot(step, MSE2_1[0][1], label='Extrapolation')
plt.plot(step, sigmaBM, label='sigmaBM')
plt.title('True errors of estimation of range Beta')
plt.xlabel('step')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```




```
#Step 16. Dependence of X on Beta shown at plot

plt.plot(Beta1[0][0], Xt1[0][0], label='X(Beta)')
plt.plot(Beta1[0][0], Xt1[2][0], label='Y(Beta)')
plt.title('Dependence of X on Beta')
plt.xlabel('step')
plt.ylabel('X(Beta),Y(Beta)')
plt.legend()
plt.grid(True)
plt.show()
```




```
#Step 17. Calculation of the condition number of covariance matrix R

ConN1_1 = np.zeros((2, 1, N))
ConN2_1 = np.zeros((2, 1, N))
for i in range(N):
    ConN1_1[:, :, i] = (sigmaD**2)/((Dz1[:, :, i]**2)*sigmaB**2)
    ConN2_1[:, :, i] = ((Dz1[:, :, i]**2)*sigmaB**2)/(sigmaD**2)
plt.plot(step, ConN2_1[0][0], label='Condition Number')
plt.title("Condition number of matrix R ")
plt.xlabel('step')
plt.ylabel('Condition number')
plt.legend()
plt.grid(True)
plt.show()
```



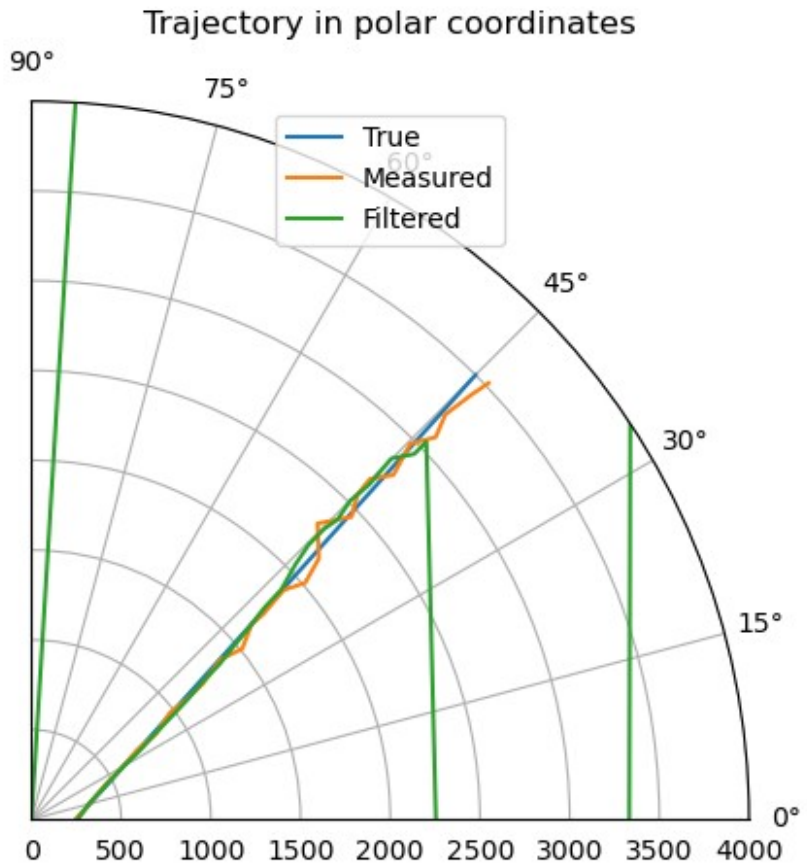
The condition number is significantly smaller than in the first case as it depends on the range D, and D is smaller than on the last step, we can observe that the number decreases over time (as the error of D is almost constant and the error of beta decreases), but when the object is too close the error os beta increases and so does the condition number.

Step 18. Conclusions how linearization errors affect tracking accuracy and how important for tracking accuracy is starting position of a moving object (close or far from an observer).

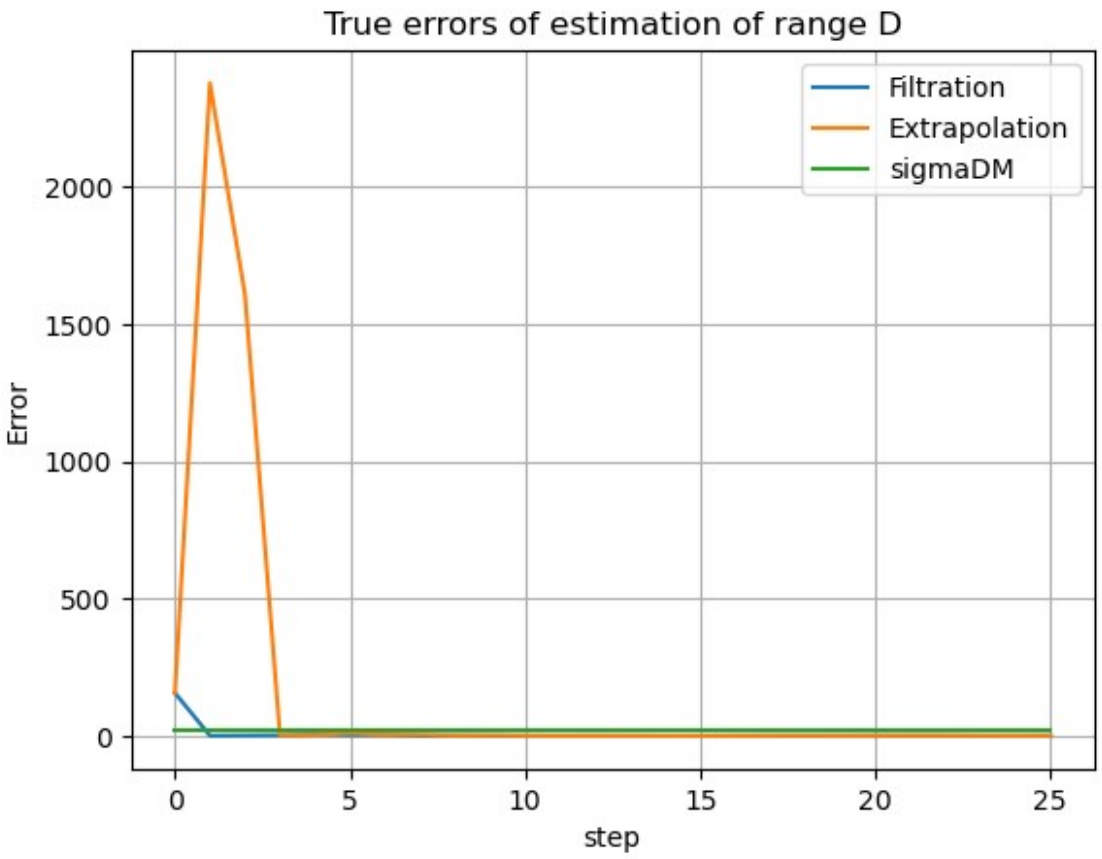
When an object is too close to the measuement device, the error increases highly. in the apropiate range our measurements are accurate enough to work with them, out of that range we start observing strong non-linearity, that leads to big linearization errors. Those errors would make the data too noisy to be used.

```
#Step 19 Run filter again at 500 run.
sigmaD19 = 50
sigmaB19 = 0.0015

Xt2 = generate_Xt(x01, Fi, N)
D2, Beta2 = generate_Polar_Xt(Xt2, N)
Dz2, BetaZ2 = generate_Polar_Z(sigmaD19, sigmaB19, D1, Beta2, N)
Zm2 = pseudo_measurement(Dz2, BetaZ2, N)
R2 = Covariance_matrix(sigmaD19,sigmaB19,Zm2,N)
Xf2, K2, Db2, Betab2, Df2, Betaf2 = kalmanFilter_from_polar_Z(Fi, H,
R2, X0, P0, Zm2, N)

#Plot polar
ax = plt.subplot(111, projection='polar')
ax.plot(Beta1[0][0],D1[0][0], label='True')
```

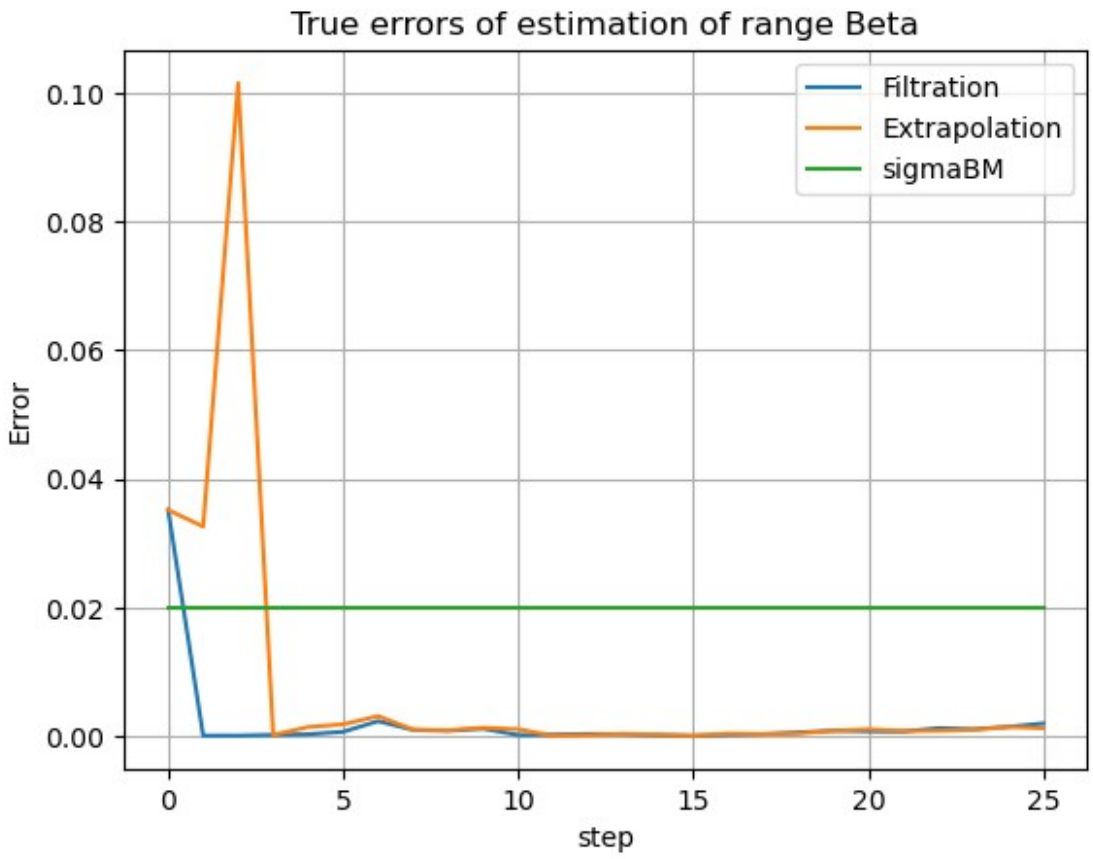
```
ax.plot(BetaZ1[0][0], Dz1[0][0], label='Measured')
ax.plot(Betab1[0][0], Db1[0][0], label='Filtered')
ax.set_title('Trajectory in polar coordinates')
ax.set_xlim(0,np.radians(90))
ax.set_ylim(0,4000)
ax.legend()

plt.show()
```



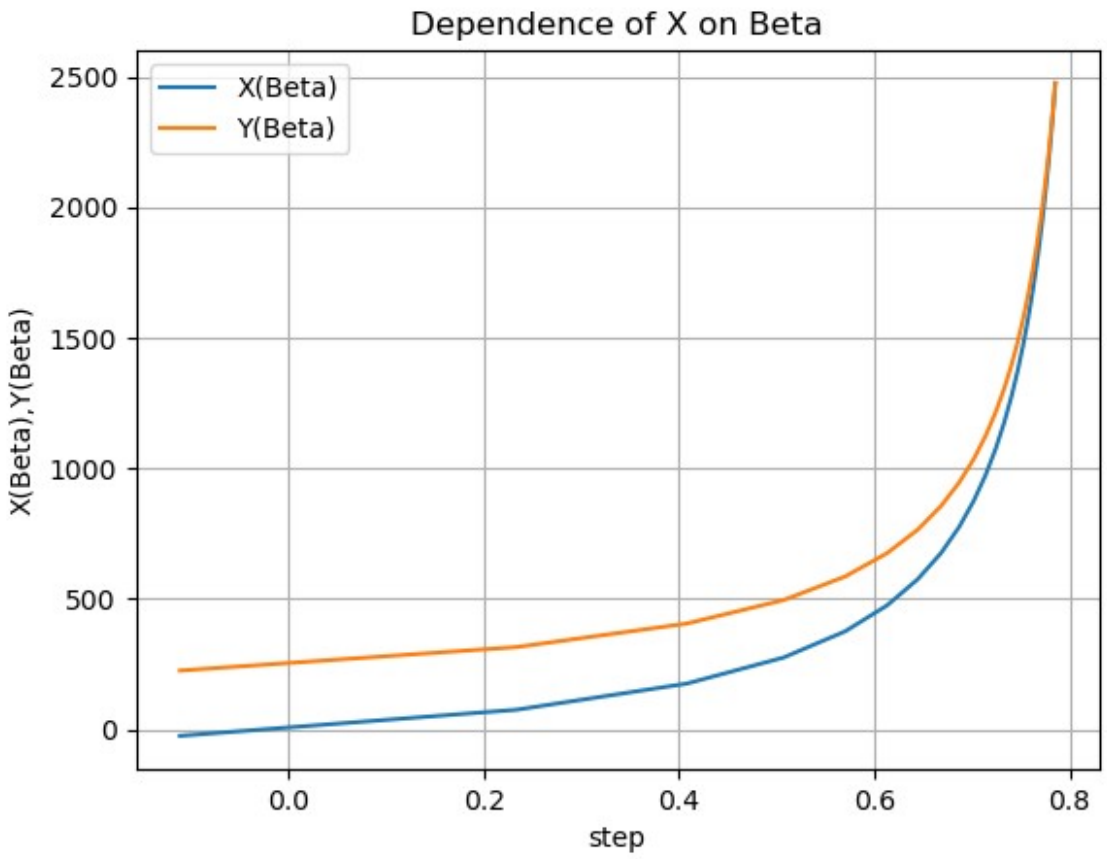

```
#mean-squared error
MSE1_2, MSE2_2 = calculate_MSE(N, M, x01, sigmaD19, sigmaB19, Fi, H,
R, X0, P0)

# Plot
plt.plot(step, MSE1_2[0][0], label='Filtration')
plt.plot(step, MSE2_2[0][0], label='Extrapolation')
plt.plot(step, sigmaDM, label='sigmaDM')
plt.title('True errors of estimation of range D')
plt.xlabel('step')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```



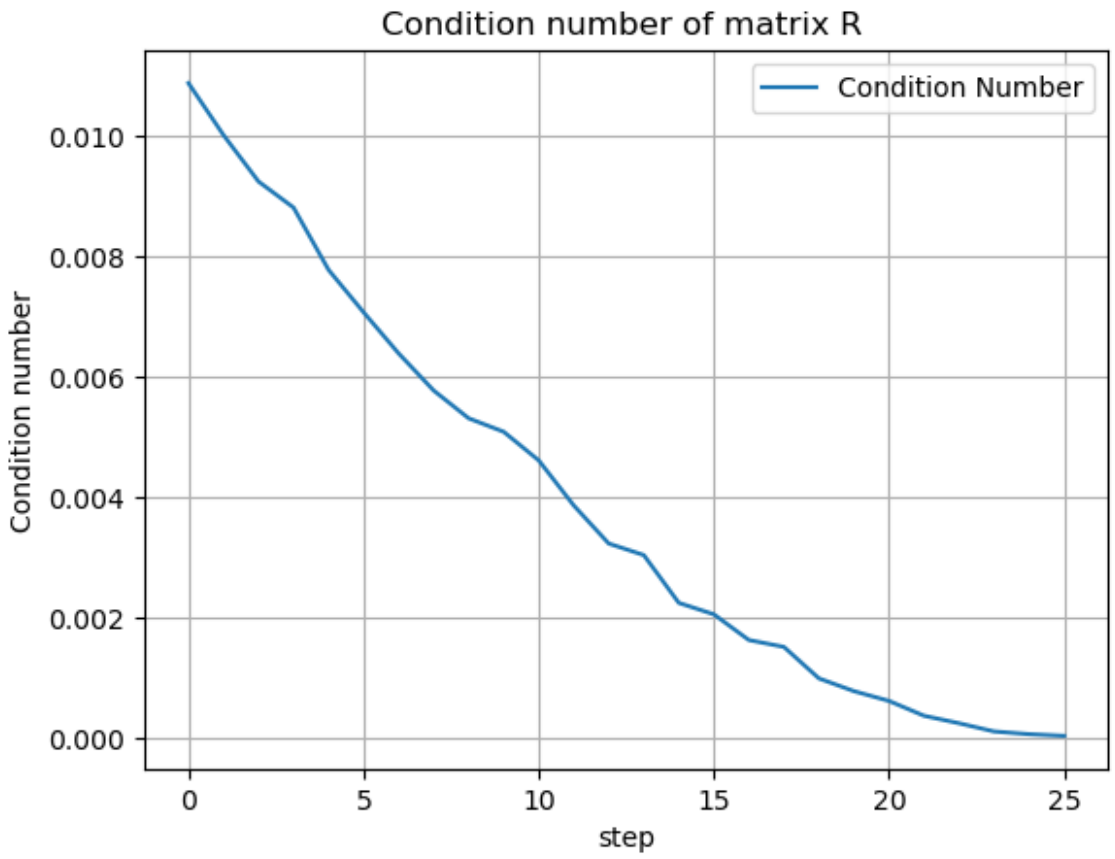

```
#mean-squared error
plt.plot(step, MSE1_2[0][1], label='Filtration')
plt.plot(step, MSE2_2[0][1], label='Extrapolation')
plt.plot(step, sigmaBM, label='sigmaBM')
plt.title('True errors of estimation of range Beta')
plt.xlabel('step')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```




```
plt.plot(Beta2[0][0], Xt2[0][0], label='X(Beta)')
plt.plot(Beta2[0][0], Xt2[2][0], label='Y(Beta)')
plt.title('Dependence of X on Beta')
plt.xlabel('step')
plt.ylabel('X(Beta),Y(Beta)')
plt.legend()
plt.grid(True)
plt.show()
```




```
ConN1_2 = np.zeros((2, 1, N))
ConN2_2 = np.zeros((2, 1, N))
for i in range(N):
    ConN1_2[:, :, i] = (sigmaD19**2)/((Dz2[:, :, i]**2)*sigmaB19**2)
    ConN2_2[:, :, i] = ((Dz2[:, :, i]**2)*sigmaB19**2)/(sigmaD19**2)
plt.plot(step, ConN2_2[0][0], label='Condition Number')
plt.title("Condition number of matrix R ")
plt.xlabel('step')
plt.ylabel('Condition number')
plt.legend()
plt.grid(True)
plt.show()
```



Step 21. Make final conclusions under which conditions navigation system may become blind and filter may diverge. Which factors has the greatest influence? Linearization errors or ill-conditioned problem? Which solution can help to overcome this particular ill-conditioned problem?

If the object is close to observer navigation system may become blind, which means that the condition number of matrix R has become extremely large.

We can not compare the effect of last experiments as we've got the same results.

```
Personal Learning log

#Yaroslav: Learned about blind zones of navigation systems and
influence of the positioning.

#Lisa: I learned how to create graphs in polar coordinates, trained
in developing functions and writing the clean code as in this work
we did a lot of runs of the same functions

#Selamawit Asfaw: I learned how to transform data in different
coordinate systems
```