Table of Contents

```
▼ 1 [1-Q1] Optimal estimator of V from U of the form \alpha U
    1.1 MSE numerical estimate
  2 [1-Q2] Optimal estimator of V from U of the form \alpha + \beta U
  3 [1-Q3] Optimal estimator of V from U of the form \alpha + \beta U + \gamma U^2
▼ 4 [2-Q6] EKF implementation
    4.1 Plot the measured and truth data for visualization
    4.2 Initialize the necessary parameters/variables

<u>4.3 Implement the EKF for measurements.npy</u>

      4.3.1 RMS for y

▼ 4.4 Implement the EKF for measurements2.npy

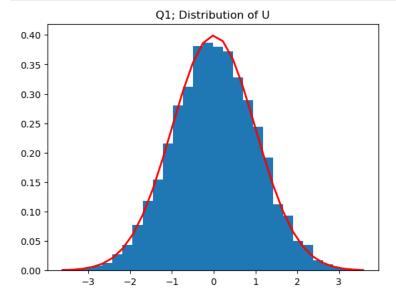
      4.4.1 RMS for y2
▼ 5 Particle Filter
    5.1 The Particle Filter algorithm (with resampling step)
    5.2 Plot mean and variance superposed to trajectory
```

ECE6555 HW5

Author: Teo Wilkening Due Date: 2022-12-16

1 [1-Q1] Optimal estimator of V from U of the form αU

```
In [1]: ▶
              1 import matplotlib.pyplot as plt
              2 import numpy as np
In [2]: ▶
              1 u_mu, u_sigma = 0, 1
              2 n = 10000 # number of samples
              3 u = np.random.normal(u_mu, u_sigma,n)
              4 count, bins, ignored = plt.hist(u, 30, density=True)
                 plt.plot(bins, 1/(u_sigma * np.sqrt(2 * np.pi)) * np.exp( - (bins - u_mu)**2 / (2 * u_sigma**2) ),
                           linewidth=2, color='r')
              8 plt.title('Q1; Distribution of U')
              9 plt.show()
```



1.1 MSE numerical estimate

```
In [3]: ▶
            1 v = np.sqrt(u**2)
             2 Rv = np.sum(v**2)/n
             3 Rvu = np.sum(u*v)/n
             4 v_mu = np.sum(v)/n
             5 MSE_linear = Rv - Rvu**2 - v_mu**2
             6 print(f"""The Mean Square error of the linear estimate (for v centered) is: {MSE_linear}""")
            8 MSE uncentered = Rv - Rvu**2
               print(f"""The Mean Square error of the linear estimate (for v un-centered) is: {MSE_uncentered}""")
```

The Mean Square error of the linear estimate (for v centered) is: 0.362708823099285 The Mean Square error of the linear estimate (for v un-centered) is: 1.0130574309689977

2 [1-Q2] Optimal estimator of V from U of the form $\alpha + \beta U$

```
In [4]: ► 1 MSE affine = Rv - Rvu**2 - v mu**2
             2 print(f"""The Mean Square error of the affine estimate (for v centered) is: {MSE_affine}""")
```

The Mean Square error of the affine estimate (for v centered) is: 0.362708823099285

3 [1-Q3] Optimal estimator of V from U of the form $\alpha + \beta U + \gamma U^2$

```
1 alpha = v_mu
In [5]: ▶
             2 beta = Rvu
             3 | phi = np.sum((u^{**2})*v)/n
             4 gamma = 1/3*(phi - v_mu)
             5 MSE_quadratic = Rv - 2*(beta*Rvu + gamma*phi + v_mu**2) + (v_mu**2 + beta**2 + 3*gamma**2 + 2*v_mu*gamma)
             6 print(f"""The Mean Square error of the quadratic estimate (for v centered) is: {MSE_quadratic}""")
```

The Mean Square error of the quadratic estimate (for v centered) is: 0.1423163012004771

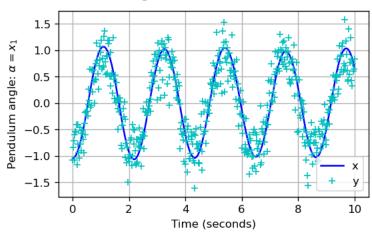
4 [2-Q6] EKF implementation

(will be borrowing my code from HW #4)

4.1 Plot the measured and truth data for visualization

```
In [6]: ▶
                 1 import numpy as np
                 2 from scipy import signal
                 4 tscale, x = np.load("groundtruth.npy") # ground truth at 1ms
                   tscale_measurement, y = np.load("measurements.npy") # sampled at 20ms tscale_measurement2, y2 = np.load("measurements2.npy") # sampled at 2ms
                 8 fig, ax = plt.subplots(figsize=(5,3), dpi=120)
                10 ax.plot(tscale,x,'b')
               11 ax.grid(True)
               12 ax.plot(tscale_measurement,y,'c+')
               13 ax.legend(['x','y'])
14 ax.set_ylabel(r'Pendulum angle: $\alpha = x_1$')
15 ax.set_xlabel(r'Time (seconds)')
               16 fig.suptitle('Angle and Measurements')
               17
                18 plt.show()
```

Angle and Measurements



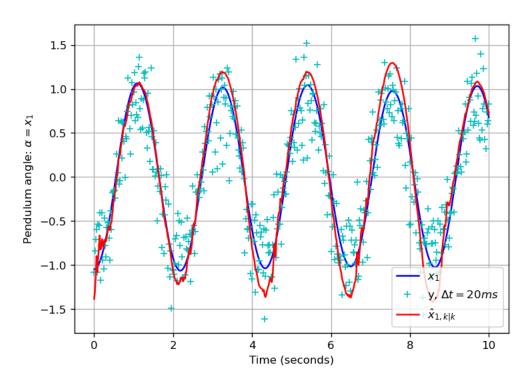
4.2 Initialize the necessary parameters/variables

```
In [7]: ▶
             1 # initialize all of the variables that we're going to need
              2 Delta = 0.020 # 20 ms
              3 \text{ sigma_m} = 0.3
              4 sigma_p = 0.1
              5 Qk = (sigma_p**2)*np.matrix([[Delta**3/3,Delta**2/2],
                                              [Delta**2/2,Delta]])
              7 Rk = np.matrix([sigma_m**2])
              8 g = 9.8 \# m/s^2
              9
             10 def f(xk,dt):
             11
                    fk_x = np.matrix([[xk[0][0] + xk[1][0]*dt],
                                    [xk[1][0] - g*dt*np.sin(xk[0][0])]])
             12
                     return fk_x
             13
             14
             15 def F(xhat,dt):
                    F_x = np.matrix([[1
                                                                 , dt],
             16
             17
                                     [-g*dt*np.sin(xhat[0][0]), 1]])
             18
                     return F_x
             19
             20 def h(xk,dt=None):
                    return np.sin(xk[0][0])
             21
             22
             23 def H(xhat,dt=None):
             24
                    return np.matrix([np.cos(xhat[0][0]), 0])
             25
             26 # # display Qk
27 # print(f"""Qk = {Qk} \n""")
28 # display(Rk)
             29
             30 # # test function f:
             31 # xhat_kk = np.array([[1],[2]])
             32 # print("f")
             33 # display(f(xhat_kk,Delta))
             34
             35 # # test function F:
             36 # print("xhat_kk")
             37 # display(xhat_kk[0][0])
             38 # print("F:")
             39 # display(F(xhat_kk,Delta))
             40
             41 # # test function h
             42 # xhat_km1 = np.array([[1],[2]])
43 # print("h:")
             44 # display(h(xhat_km1))
             45
             46 # # test function K
             47 # print("H:")
             48 | # display(H(xhat_km1))
```

4.3 Implement the EKF for measurements.npy

```
In [8]: ▶
           1 # the initial guesses of x and P
            2 xhat_init = np.array([[y[0]],[0]]) # set alpha = y[0] and d(alpha)/dt = 0
            3 P_init = np.identity(2)
            4 NumSteps = len(y)
            6 # initializing the matrices for computing Kalman filter state evolution over time
            7 xhat_k_pred = np.zeros((NumSteps,2,1))
            8 xhat_k_curr = np.zeros((NumSteps,2,1))
            9 P_k_pred = np.zeros((NumSteps,2,2))
           10 P_k_curr = np.zeros((NumSteps,2,2))
           11 Kfk_curr = np.zeros((NumSteps,2,1))
           12 I = np.identity(2)
           13
           14 \# setup the initial states of the prediction steps, xhat 0|-1 and P 0|-1
           15 xhat_k_pred[0,:,:] = xhat_init
           16 P_k_pred[0,:,:] = P_init[:,:]
           17
           18 # start running the Extended Kalman Filter, using the NoisyMeasurements
           19 for t in np.arange(1,NumSteps):
                 ## update step, given the measurement
           20
                  # K f,i-1
           21
           22
                  Hkm1 = H(xhat_k_pred[t-1,:,:])
           23
                  Kfk_curr[t-1,:,:] = P_k_pred[t-1,:,:] @ Hkm1.T @ np.linalg.inv(Hkm1 @ P_k_pred[t-1,:,:] @ Hkm1.T + Rk)
           24
                  # P i-1/i-1
           25
                  P_k_curr[t-1,:,:] = (I - Kfk_curr[t-1,:,:] @ Hkm1) @ P_k_pred[t-1,:,:]
           26
                  # x i-1/i-1
           27
                  xhat_k_curr[t-1,:,:] = xhat_k_pred[t-1,:,:] + Kfk_curr[t-1,:,:] * (y[t-1] - h(xhat_k_pred[t-1,:,:]) )
           28
           29
                  ## Predicition Step
           30
                  # x i/i-1
                  xhat_k_pred[t,:,:] = f(xhat_k_curr[t-1,:,:],Delta)
           31
           32
                  # P i/i-1
           33
                  34
           35 ## and set the Last Update step
           36 t = NumSteps - 1
           37 # K f,t
           38 Hkm1 = H(xhat k pred[t,:,:])
           40 # P t/t
           41 P_k_curr[t,:,:] = (I - Kfk_curr[t,:,:] @ Hkm1) @ P_k_pred[t,:,:]
           42 # x t/t
           43 xhat_k_curr[t,:,:] = xhat_k_pred[t,:,:] + Kfk_curr[t,:,:] * (y[t] - h(xhat_k_pred[t,:,:]) )
           44
           45 fig, ax = plt.subplots(figsize=(7,5), dpi=120)
           46
           47 ax.plot(tscale,x,'b')
           48 ax.grid(True)
           49 ax.plot(tscale_measurement,y,'c+')
           50 ax.plot(tscale_measurement,xhat_k_curr[:,0,:],'r-')
           51 ax.legend([r'$x_1$',r'y, $\Delta t = 20ms$',r'$\hat{x}_{1,k|k}$'])
           52 ax.set_ylabel(r'Pendulum angle: $\alpha = x_1$')
           53 ax.set_xlabel(r'Time (seconds)')
           54 fig.suptitle(r'EKF for $\Delta t = 20ms$')
           55
           56 plt.show()
           57
```

EKF for $\Delta t = 20ms$



4.3.1 RMS for y

```
1 x_20ms = x[::20]
2 error = y - x_20ms
3 error_kalman_f = xhat_k_curr[:,0,:].reshape(-1) - x_20ms
4 rms_error = np.sqrt(np.sum(error**2)/len(y))
5 rms_kalman_f = np.sqrt(np.sum(error kalman_f**2)/len(y))
In [9]: ▶
                           5 rms_kalman_f = np.sqrt(np.sum(error_kalman_f**2)/len(y))
                           print(f'''RMS of the measurement y error: {rms_error}''')
print(f'''RMS of the EKF estimates: {rms_kalman_f}''')
```

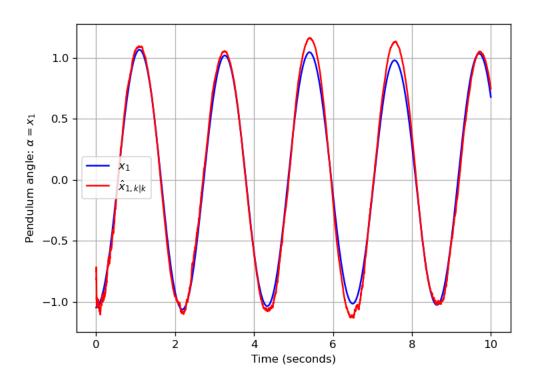
RMS of the measurement y error: 0.31853835229944727 RMS of the EKF estimates: 0.18003291734915677

4.4 Implement the EKF for measurements2.npy

```
In [10]: N
                         1 # initialize all of the variables that we're going to need
                          2 Delta = 0.002 # 2 ms
                          3 | sigma_m = 0.3
                          4 sigma_p = 0.1
                              Qk = (sigma_p**2)*np.matrix([[Delta**3/3,Delta**2/2],
                                                                                [Delta**2/2,Delta]])
                          7 Rk = np.matrix([sigma_m**2])
                          8 \mid g = 9.8 \# m/s^2
                          9
                        10 def f(xk,dt):
                                     fk_x = np.matrix([[xk[0][0] + xk[1][0]*dt],
                        11
                                                                [xk[1][0] - g*dt*np.sin(xk[0][0])]])
                        12
                        13
                                      return fk x
                        14
                        15 def F(xhat,dt):
                        16
                                     F_x = np.matrix([[1
                                                                                                                , dt],
                                                                  [-g*dt*np.sin(xhat[0][0]), 1]])
                        17
                        18
                                      return F x
                        19
                        20 def h(xk,dt=None):
                                     return np.sin(xk[0][0])
                        21
                        22
                        23 def H(xhat,dt=None):
                        24
                                      return np.matrix([np.cos(xhat[0][0]), 0])
                        25
                        26 \mid# the initial guesses of x and P
                        27 xhat_init = np.array([[y2[0]],[0]]) # set alpha = y2[0] and d(alpha)/dt = 0
                        28 P_init = np.identity(2)
                        29 NumSteps = len(y2)
                        30
                        31 # initializing the matrices for computing Kalman filter state evolution over time
                        32 xhat_k_pred = np.zeros((NumSteps,2,1))
                        33 xhat_k_curr = np.zeros((NumSteps,2,1))
                        34 P_k_pred = np.zeros((NumSteps,2,2))
                        35 P_k_curr = np.zeros((NumSteps,2,2))
                        36 Kfk_curr = np.zeros((NumSteps,2,1))
                        37 I = np.identity(2)
                        38
                        39 # setup the initial states of the prediction steps, xhat 0/-1 and P 0/-1
                        40 xhat_k_pred[0,:,:] = xhat_init
                        41 P_k_pred[0,:,:] = P_init[:,:]
                        42
                        43 # start running the Extended Kalman Filter, using the NoisyMeasurements
                        44 for t in np.arange(1, NumSteps):
                        45
                                      ## update step, given the measurement
                        46
                                      # K f, i-1
                        47
                                      Hkm1 = H(xhat_k_pred[t-1,:,:])
                                       Kfk\_curr[t-1,:,:] = P\_k\_pred[t-1,:,:] @ Hkm1.T @ np.linalg.inv(Hkm1 @ P\_k\_pred[t-1,:,:] @ Hkm1.T + Rk) 
                        48
                        49
                                      # P i-1/i-1
                        50
                                      P_k_{curr}[t-1,:,:] = (I - Kfk_{curr}[t-1,:,:] @ Hkm1) @ P_k_{pred}[t-1,:,:]
                        51
                        52
                                     xhat_k\_curr[t-1,:,:] = xhat_k\_pred[t-1,:,:] + Kfk\_curr[t-1,:,:] * (y2[t-1] - h(xhat_k\_pred[t-1,:,:]) )
                        53
                        54
                                      ## Predicition Step
                        55
                                      # x i/i-1
                        56
                                     xhat_k_pred[t,:,:] = f(xhat_k_curr[t-1,:,:],Delta)
                        57
                                      # P i/i-1
                                      P_k pred[t,:,:] = F(xhat_k\_curr[t-1,:,:], Delta) @ P_k\_curr[t-1,:,:] @ F(xhat_k\_curr[t-1,:,:], Delta).T + Qk pred[t,:,:] = F(xhat_k\_curr[t-1,:,:], Delta).T + Qk pred[t,:] = F(xhat_k\_curr[t-1,:,:], 
                        58
                        59
                              ## and set the last Update step
                        60
                        61 t = NumSteps - 1
                        62 # K f,t
                        63 | Hkm1 = H(xhat_k_pred[t,:,:])
                        64 Kfk_curr[t,:,:] = P_k_pred[t,:,:] @ Hkm1.T @ np.linalg.inv(Hkm1 @ P_k_pred[t,:,:] @ Hkm1.T + Rk)
                        65 # P t/t
                        66 P_k_curr[t,:,:] = (I - Kfk_curr[t,:,:] @ Hkm1) @ P_k_pred[t,:,:]
                        67 # x + l +
                        68 | xhat_k_curr[t,:,:] = xhat_k_pred[t,:,:] + Kfk_curr[t,:,:] * (y2[t] - h(xhat_k_pred[t,:,:]) )
                        69
                        70 fig, ax = plt.subplots(figsize=(7,5), dpi=120)
                        71
                        72 ax.plot(tscale,x,'b')
                        73 ax.grid(True)
                        74 # ax.plot(tscale_measurement2,y2,'c+')
                        75 ax.plot(tscale_measurement2,xhat_k_curr[:,0,:],'r-')
                        76 # ax.legend([r'$x_1$','y2',r'$\hat\x}_{1,k|k}$'])
77 ax.legend([r'$x_1$',r'$\hat\x}_{1,k|k}$'])
                        78 ax.set_ylabel(r'Pendulum angle: $\alpha = x_1$')
                        79 ax.set_xlabel(r'Time (seconds)')
                        80 fig.suptitle(r'EKF for $\Delta t = 2ms$')
                        81
                        82 plt.show()
```

83

EKF for $\Delta t = 2ms$



4.4.1 RMS for y2

```
x_2ms = x[::2]
error = y2 - x_2ms
error_kalman_f = xhat_k_curr[:,0,:].reshape(-1) - x_2ms

rms_error = np.sqrt(np.sum(error**2)/len(y2))
In [11]: ▶
                   5 rms_kalman_f = np.sqrt(np.sum(error_kalman_f**2)/len(y2))
                   7 print(f'''RMS of the measurement y2 error: {rms_error}''')
                   8 print(f'''RMS of the EKF estimates: {rms_kalman_f}''')
```

RMS of the measurement y2 error: 0.31134887639065506 RMS of the EKF estimates: 0.0677126618156732

- **5 Particle Filter**
- 5.1 The Particle Filter algorithm (with resampling step)

```
In [12]: ▶
                       1 Delta = 0.020 # 20 ms
                        2 sigma m = 0.3
                        3 | sigma_p = 0.1
                        4 Qk = (sigma_p**2)*np.matrix([[Delta**3/3,Delta**2/2],
                                                                            [Delta**2/2,Delta]])
                        6 NumSteps = len(y)
                        8 # 1) draw n samples from the prior
                        9 # 2) for each k = 1...T
                       10 #
                                      a) draw samples x_k(i) from the importance distribution
                       11 #
                                      b) compute the new weights
                                       c) normalize the new weights
                       12 #
                       13
                       14 # initialize x^i_k and w^i_k matrices to keep track of state estimation distributions and weights
                       15 n = 200 # number of particles
                       16 | xki = np.zeros((NumSteps,2,n),dtype=float)
                       17 | wki = np.zeros((NumSteps,n),dtype=float)
                       18 pi_ki = np.zeros((NumSteps,2,n))
                       19
                       20 # 1) draw n samples from the prior
                       21 x0_{mu}, x0_{sigma} = y[0], np.sqrt(0.5)
                       22 x0 = np.random.normal(x0_mu, x0_sigma, n)
                       23 w0 = 1/n*np.ones(n)
                       25 # insert the samples from the prior into our matrices for keeping track of things
                       26 | xki[0,:] = x0
                       27 wki[0,:] = w0
                       28
                       29 # initialize noise Gaussian parameters
                       30 q_mu, q_cov = [0,0], Qk
                       31 v_mu, v_sigma = 0, sigma_m
                       32
                       33 # 2) for each k = 1...T
                       34 mean = np.zeros(NumSteps) # keep track of the mean of the particles
                       35 var = np.zeros(NumSteps) # keep track of the variance of the particles at each step
                       36 neff = np.zeros(NumSteps)
                       37
                       38 mean[0] = x0_mu
                       39 var[0] = x0_sigma**2
                       40
                       41 #################
                       42
                           # Need to figure out how to accurately sample and push 2nd state through the PF as well
                       43 # the following code will not function without more work/massaging being done.
                       44 ##################
                       45
                       46 # for k in np.arange(1, NumSteps, 1):
                       47
                                       # a) draw samples x_k(i) from the importance distribution
                                       # pi_ki[k,:,:] = np.matrix([[]])
                       48 #
                                       xki[k,:] = 1/2*xki[k-1,:] + 25*xki[k-1,:]/(1 + xki[k-1,:]**2) + 8*np.cos(1.2*(k-1)) + \\ 1/2*xki[k-1,:] + 2/2*xki[k-1,:]/(1 + xki[k-1,:]**2) + 8*np.cos(1.2*(k-1)) + \\ 1/2*xki[k-1,:] + 2/2*xki[k-1,:]/(1 + xki[k-1,:]**2) + 8*np.cos(1.2*(k-1)) + \\ 1/2*xki[k-1,:] + 2/2*xki[k-1,:]/(1 + xki[k-1,:]**2) + 8*np.cos(1.2*(k-1)) + \\ 1/2*xki[k-1,:] + 2/2*xki[k-1,:]/(1 + xki[k-1,:]**2) + 8*np.cos(1.2*(k-1)) + \\ 1/2*xki[k-1,:] + 2/2*xki[k-1,:]/(1 + xki[k-1,:]**2) + 8*np.cos(1.2*(k-1)) + \\ 1/2*xki[k-1,:] + 2/2*xki[k-1,:]/(1 + xki[k-1,:]**2) + 8*np.cos(1.2*(k-1)) + \\ 1/2*xki[k-1,:] + 2/2*xki[k-1,:]/(1 + xki[k-1,:]**2) + 8*np.cos(1.2*(k-1)) + \\ 1/2*xki[k-1,:]/(1 + xki[k-1,:]/(1 
                       49 #
                       50
                            #
                                                          np.random.multivariate_normal(q_mu, q_cov,n)
                                       # print(sum(xki[k,:]))
                       51 #
                       52 #
                                       # b) compute the new weights
                                       wki[k,:] = wki[k-1,:]*1/np.sqrt(2*np.pi)*np.exp(-0.5*(y[k] - 1/20*(xki[k-1,:]**2))**2)
                       53 #
                       54 #
                                       # c) normalize the new weights
                       55 #
                                       wki[k,:] = wki[k,:]/sum(wki[k,:])
                       56
                                       mean[k] = np.average(xki[k,:],weights=wki[k,:])
                                       var[k] = np.average((xki[k,:] - mean[k])**2, weights=wki[k,:])
neff[k] = 1/sum(wki[k,:]**2)
                       57
                            #
                       58 #
                       59 #
                                        # draw new samples if the number of effective weights is < 20
                                       if neff[k] < 20:
                       60
                            #
                                             print(f""Effective particles < 20 for step {k}""")</pre>
                       61 #
                       62 #
                                              ind = np.argsort(xki[k,:]) # index sort of the particles
                       63 #
                                              xki[k,:] = np.take_along_axis(xki[k,:],ind,axis=0)
                       64 #
                                              wki[k,:] = np.take\_along\_axis(wki[k,:],ind,axis=0) # sort the weights according to the particles
                       65
                                              bins = np.cumsum(wki[k,:]) # bins from which we are going to sample; cumulative sum of the weights
                                             uni = np.random.uniform(0,1,n) # uniform distribution used for re-sampling
                       66 #
                       67 #
                                             uni2 = np.random.uniform(0,1,n) # secondary random sampling for within bins
                       68 #
                                             for i in np.arange(0,n):
                       69 #
                                                    for j in np.arange(n-1,-1,-1):
                                                           if uni[i] >= bins[j]:
                       70
                                                                xki[k,i] = xki[k,j] + (xki[k,j+1] - xki[k,j])*uni2[i]
                       71 #
                                             # and reset the weights:
                       72 #
                       73 #
                                              wki[k,:] = w0
                       75 # # track mean for later analysis
                       76 # mean_pf_resamp = mean
```

5.2 Plot mean and variance superposed to trajectory

```
In [13]: ► 1 # # Plot the trajectory
                2 # fig, ax = plt.subplots(figsize=(8,5), dpi=120)
                4 # ax.plot(np.insert(TimeScale,0,0),x,'b--')
                5 # ax.grid(True)
                6 # ax.plot(np.insert(TimeScale,0,0),x,'bs',markersize=6)
                7 # #plt.legend(['line', 'markers'])
                8 # ax.set_ylabel(r'State $x_k$')
9 # ax.set_xlabel(r'Time (sample $k$)')
               10 # for k in np.arange(1,NumSteps,1):
               11 #
                       for i in np.arange(n):
                              if wki[k,i] > 1e-3:
               12 #
                                  ax.plot(k,xki[k,i],'ro',markersize=10*wki[k,i],alpha=0.3)
               13 #
               14 # ax.plot(mean, 'r+')
               15 # ax.fill_between(np.arange(NumSteps), mean-2*np.sqrt(var), mean+2*np.sqrt(var), alpha=0.25, color='r')
16 # fig.suptitle('Particle Filter with re-sampling')
               17
               18 # plt.show()
```