midterm_Q2_MarkAlex_Finalized

November 1, 2018

1 ASEN 6519 Fall 2018 - Midterm Project

1.1 Extended KF and Ensemble Filters with Lorenz-63

1.1.1 The Lorenz-63 Model

Low-dimensional model: the state is made of 3 variables $x = (x1, x2, x3)$ and dynamical system evolution is given by the following equations:

$$\frac{\partial x_1}{\partial t} = -Ax_1 + Ax_2$$
$$\frac{\partial x_2}{\partial t} = Bx_1 - x_1x_3 - x_2$$
$$\frac{\partial x_3}{\partial t} = x_1x_2 - Cx_3$$

where $A = 10$, $B = 28$, and $C = 8/3$.

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

Define Helper Functions

```
In [2]: def dfdt(r, coeffs):
            '''
            Returns dx1/dt, dx2/dt, dx3/dt
            '''
            # Current position
            x1 = r[0]
            x2 = r[1]
            x3 = r[2]

            # Extract coefficients
            A = coeffs[0]
            B = coeffs[1]
            C = coeffs[2]

            # Calculate the derivative
            dx1 = -A*x1 + A*x2
```

```
    dx2 = B*x1 - x1*x3 - x2
    dx3 = x1*x2 - C*x3

    return np.array([dx1, dx2, dx3], float)
```

In [3]:
```
def dPdt(P, A, Q):
    '''
    Returns dP/dt; From Simon's "Optimal State Estimation", Eqn 13.28

    Inputs:
        - P: covariance of the state
        - A: the TLM
        - Q: noise
    '''

    return A@P + P@A.T + Q
```

In [4]:
```
def advance_state(r, coeffs, h):
    '''
    Calculate the state of the system at time t+1 using RK2

    Inputs:
        r - the state
        coeffs - the standard Lorenz system coefficients
        h - timstep
    '''

    # Calculate Runge-Kutta2 coefficients
    k1 = h*dfdt(r, coeffs)
    k2 = h*dfdt(r+0.5*k1, coeffs)

    # Update current position
    r += k2

    return r
```

In [5]:
```
def advance_cov(P, M, Q, h):
    '''
    Calculate the covariance P at time t+1 using RK2
    '''

    # Calculate Runge-Kutta2 coefficients
    k1 = h*dPdt(P, M, Q)
    k2 = h*dPdt(P+0.5*k1, M, Q)

    # Update current position
    P += k2

    return P
```

In [6]:
```
def calc_TLM(x):
    global A_lor, B_lor, C_lor

    x1 = x[0]
    x2 = x[1]
    x3 = x[2]

    return np.array([
        [-A_lor,   A_lor,  0.0],
        [B_lor-x3, -1.0,  -x1],
        [x2,       x1,    -C_lor]])
```

1.1.2 1. Generate *true* state and synthetic observations using Lorenz63 setup with the default inputs.

In [7]:
```
## Background error variance
B_struct_Identity = np.array([
    [1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]])

## Set up necessary matrixies
H = np.eye(3)
I = np.eye(3)

## State initial default input parameters
x1_0 = 1.5
x2_0 = -1.5
x3_0 = 25.5

## Model error covariance (ie: process noise), from Evensen 1997
Q = np.zeros((3,3))
Q[0,0] = 0.1491
Q[0,1] = 0.1505
Q[0,2] = 0.0007
Q[1,0] = 0.1505
Q[1,1] = 0.9048
Q[1,2] = 0.0014
Q[2,0] = 0.0007
Q[2,1] = 0.0014
Q[2,2] = 0.9180
L, V = np.linalg.eig(Q)
Qsqrt_cov = V*np.sqrt(L)


varobs = 3   # Observation error variance
svar = 0.3
```

```
## Measurement covariance
sobs = np.sqrt(varobs)
R_struct = np.eye(3)
R = varobs * R_struct
L, V = np.linalg.eig(R)
Msqrt_cov = V*np.sqrt(L)


## Model setup
N_state = 3 # Number of state variables
MO = 3      # Number of measurement variables
# Default Lorenz system parameters
A_lor = 10
B_lor = 28
C_lor = 8/3
coeffs = np.array([A_lor, B_lor, C_lor],float)


# Numerical parameters
t0 = 0          # sec, initial time
tf_assim = 3    # sec, final time   of forecast
tf_fore = 1     # sec, final length of forecast
h_sys = 0.01    # sec, timestep for system dynamics
```

1.1.3 2. Implement your own EKF to estimate x_a. The tangent linear and adjoint codes are provided.

- Investigate the performance of filter in terms of the bias and mean-square-error of data assimilation analysis. How does the estimate-error computed from x_a and the truth value of x compare to the error variance suggested by the posterior covariance?
- Examine the sensitivity of the filter performance to the errors in the background. (Compare at least 10 cases)
- Examine the sensitivity of the filter performance to the accuracy and frequency of observations. Is it better to have frequent but inaccurate observations or infrequent but accurate ones? (Compare at least 10×10 cases.)

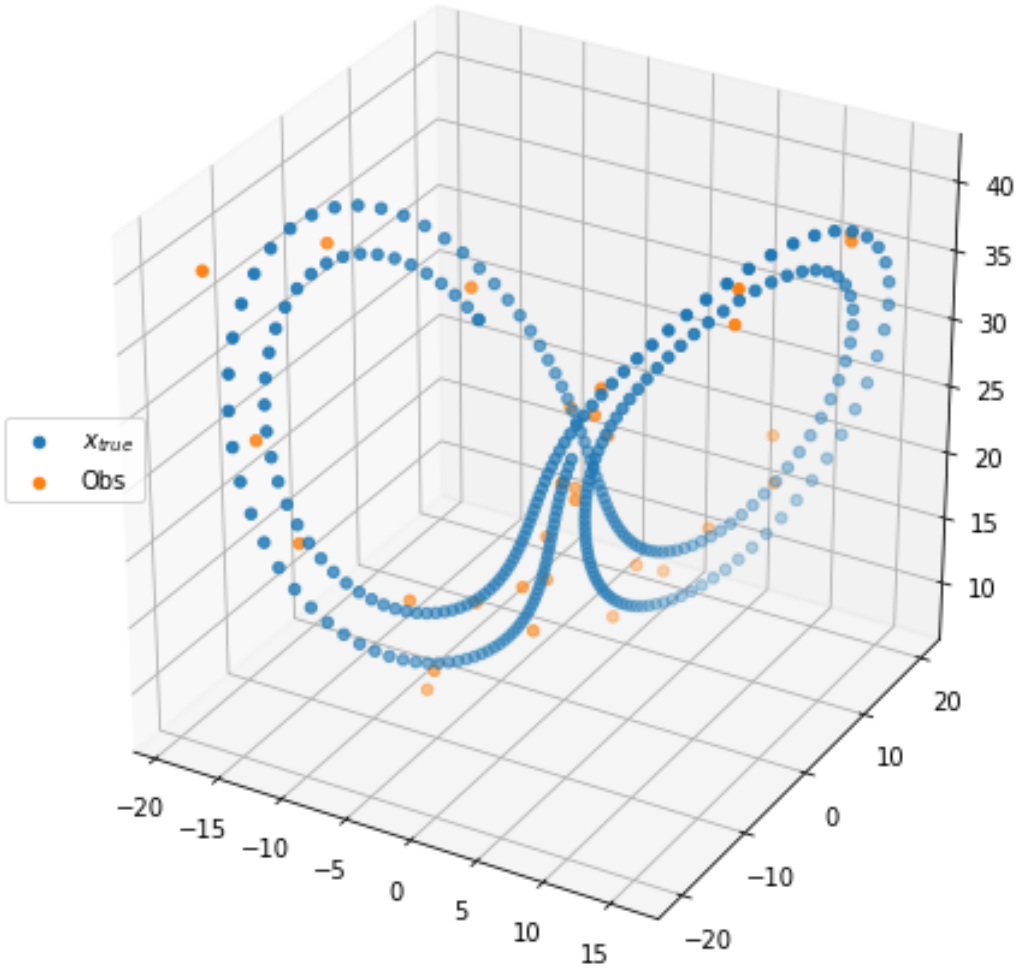
EKF Algorithm (Lecture 6, Slide 9) * Note * M_{nl} and H_nl are the true nonlinear forward model and observation models, and M and H are the tangent linear models of the nonlinear operators * where $M_t = \frac{\delta M_{nl}}{\delta x}$, evaluated at $x = x^a_{t-1}$ * Our H is linear for this problem * *Forecast* * $x^f_t = M_{nl}(x^a_{t-1})$ * $P^f_t = M_t P^a_{t-1} M^T_t + Q_t$ * *Update* * $x^a_t = x^f_t + K_t(y_t - H_{nl}(x^f_t))$ * $P^a_t = (I - K_t H_t)P^f_t$ * where $K_t = P^f_t H^T_t (R_t + H_t P^f_t H^T_t)^{-1}$

Run the System: Generate States (x_{true}) and Observations (y)

In [8]:
```
# ================== To change observation size ==================
# Examine the sensitivity of the filter performance to the errors in the background.
# (Compare at least 10 cases)
B_cov = B_struct_Identity * 0.1
```

```
        # ================== To change observation size ==================
        # Examine the sensitivity of the filter performance to the accuracy and frequency of obs
        # Number of dynamics updates per measurement (changes obs size)
        # Change frequency of observations.
        dyn2meas = 10

        # Change frequency of observations.
        varobs = 1

        h_obs = h_sys * dyn2meas     # sec; an observation is generated every h_obs sec
        t_sys = np.arange(t0,tf_assim,h_sys)     # time stamps for system dynamics
        t_obs = np.arange(t0,tf_assim,h_obs)     # time stamps for observations
        N_sys = len(t_sys)               # Number of system timesteps
        N_obs = len(t_obs)               # Number of observations
        print (f'N-obs={N_obs} N-state={N_sys}')

        ## Initial conditions
        y = np.zeros([M0, N_obs])  # Measurements of x1 (with measurement noise)

        # ================== Generate True state ==================
        r = np.array([[x1_0, x2_0, x3_0]], float).T   # "Current" position
        x_true = np.zeros([N_state, N_sys])  # True history of x1 (without any noise)
        x_true[:,0] = np.squeeze(r)

        for i in range(1,N_sys):
            x_true[:,i] = advance_state(x_true[:,i-1], coeffs, h_sys)

        # ================== Generate Measurements ==================
        # add in variance "varobs" which was np.random.randn(3,1)
        for i in range(N_obs):
            curr_state = x_true[:,i*dyn2meas][:,np.newaxis]
            y[:,i] = np.squeeze(curr_state + Msqrt_cov @ (np.random.randn(3,1)))

        from mpl_toolkits.mplot3d import Axes3D
        fig = plt.figure(figsize=[8,8])
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(x_true[0,:],x_true[1,:],x_true[2,:],label='$x_{true}$')
        ax.scatter(y[0,:],y[1,:],y[2,:],label='Obs')#,s=100,marker='x')
        ax.legend(loc=6)
        plt.show()
```

```
N-obs=30 N-state=300
```

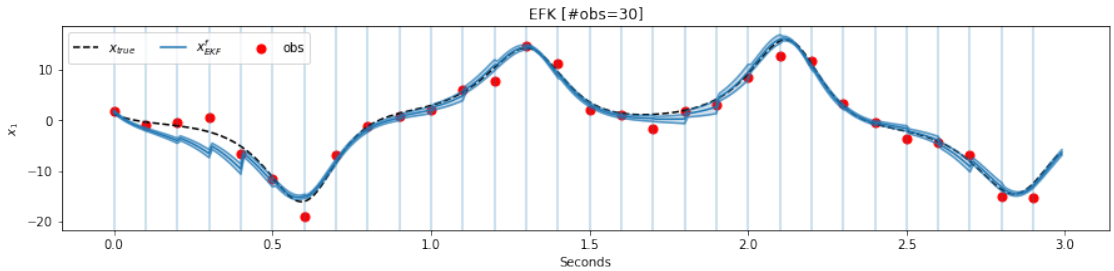


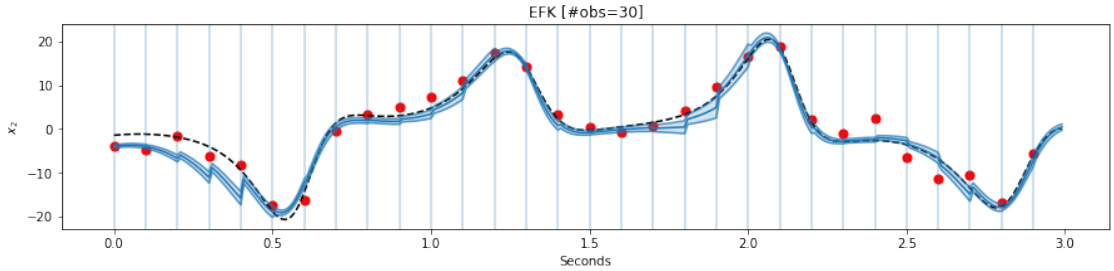
Run the EKF

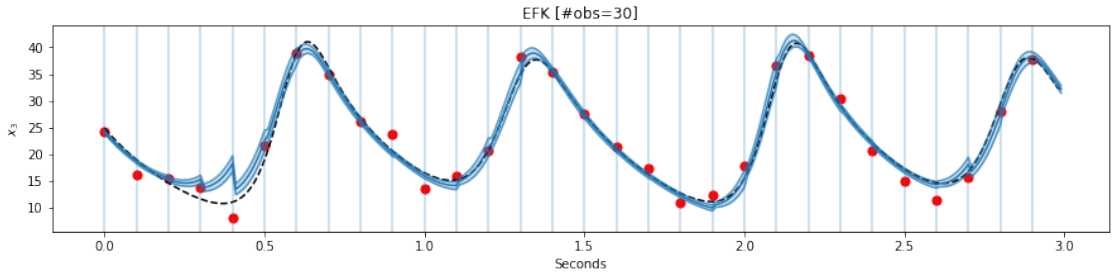
```
In [9]: ## Store histories
        xf_EKF = np.array(np.zeros([3, N_sys]))
        Pf_EKF = np.zeros(shape=[3,3,N_sys])
        xa_EKF = np.array(np.zeros([3, N_obs]))
        Pa_EKF = np.zeros(shape=[3,3, N_obs])

        ## ICs for the EKF
        xa = np.array([[y[0,0], y[1,0], y[2,0]]], float).T
        Pa = B_cov.copy()
        Pf = B_cov.copy()

        xf_EKF[:,0] = y[:,0].copy()
```

```
Pf_EKF[:,:,0] = Pa.copy()
xa_EKF[:,0] = y[:,0].copy()
Pa_EKF[:,:,0] = Pa.copy()

#### Run the EKF
## Initialization
## Current values
for j in range(1,dyn2meas+1):
    curr_state = xf_EKF[:,j-1]
    curr_P = Pf_EKF[:,:,j-1]
    prev_analysis_state = xa_EKF[:,0]
    M = calc_TLM(prev_analysis_state.copy())

    ## Forecast to next timestep
    xf = advance_state(curr_state.copy(), coeffs, h_sys)  # Forecast to next time step
    Pf = advance_cov(curr_P.copy(), M, Q, h_sys)
    xf_EKF[:,j], Pf_EKF[:,:,j] = xf.copy(), Pf.copy()

for i in range(1,N_obs):
    #### Update
    x1, x2, x3 = xf[0], xf[1], xf[2]
    y_t = y[:,i]
    M = np.array([          # The tangent linear model evaluated at x=xf_k
        [-A_lor  ,  A_lor  ,  0.0],
        [B_lor-x3, -1.0,  -x1],
        [x2  , x1  ,  -C_lor]])

    K = Pf @ H.T @ np.linalg.inv(R + H@Pf@H.T)  # Kalman gain
    xa = xf + K @ (y_t - H @ xf)   # Update mean
    Pa = (I - K @ H) @ Pf   # Update covar
    xa_EKF[:,i], Pa_EKF[:,:,i] = xa.copy(), Pa.copy()  # Store update

    #### Forecast
    for j in range(1,dyn2meas+1):

        ## Special case: end condition
        if i*dyn2meas+j >= N_sys:
            break

        ## Special case: first forecast after an obs
        if j == 1:
            ## Current values
            curr_state = xa_EKF[:,i]
            curr_P = Pa_EKF[:,:,i]
            M = calc_TLM(curr_state.copy())

            ## Forecast
            xf = advance_state(curr_state.copy(), coeffs, h_sys)  # Forecast to next tim
```

```
            Pf = advance_cov(curr_P.copy(), M, Q, h_sys)
            xf_EKF[:,i*dyn2meas+j], Pf_EKF[:,:,i*dyn2meas+j] = xf.copy(), Pf.copy()

        else:
            ## Current values
            curr_state = xf_EKF[:,i*dyn2meas+j-1]
            curr_P = Pf_EKF[:,:,i*dyn2meas+j-1]
            prev_analysis_state = xa_EKF[:,i]
            M = calc_TLM(prev_analysis_state.copy())

            ## Forecast to next timestep
            xf = advance_state(curr_state.copy(), coeffs, h_sys)  # Forecast to next tim
            Pf = advance_cov(curr_P.copy(), M, Q, h_sys)
            xf_EKF[:,i*dyn2meas+j], Pf_EKF[:,:,i*dyn2meas+j] = xf.copy(), Pf.copy()

## Plot EKF analysis

## Plot x1
for S in range(3):
    plt.figure(S+1,figsize=[15,3])#plt.subplot(3,1,S+1)
    plt.title(f'EFK [#obs={N_obs}]')
    plt.plot(t_sys, x_true[S,:], color='black', label='$x_{true}$', ls='--')
    plt.scatter(t_obs, y[S,:], s=50, label='obs',color='red')
    plt.plot(t_sys, np.ravel(xf_EKF[S,:]),color='C0',label='$x^f_{EKF}$')
    plt.plot(t_sys, np.ravel(xf_EKF[S,:] + np.sqrt(Pf_EKF[S,S,:])),color='C0', alpha=0.7
    plt.plot(t_sys, np.ravel(xf_EKF[S,:] - np.sqrt(Pf_EKF[S,S,:])),color='C0', alpha=0.7
    plt.fill_between(t_sys, np.ravel(xf_EKF[S,:] + np.sqrt(Pf_EKF[S,S,:])), \
                            np.ravel(xf_EKF[S,:] - np.sqrt(Pf_EKF[S,S,:])), color='C0',
    #plt.scatter(t_obs, np.ravel(xa_EKF[S,:]),color='orange',label='$x^a_{EKF}$')
    plt.ylabel(f'$x_{S+1}$')
    plt.xlabel('Seconds')
#     plt.ylim(-25,50)
    for v in t_obs:
        plt.axvline(v,alpha=0.3)

    if S==0:
        plt.legend(loc=2,ncol=4)

plt.show()
```



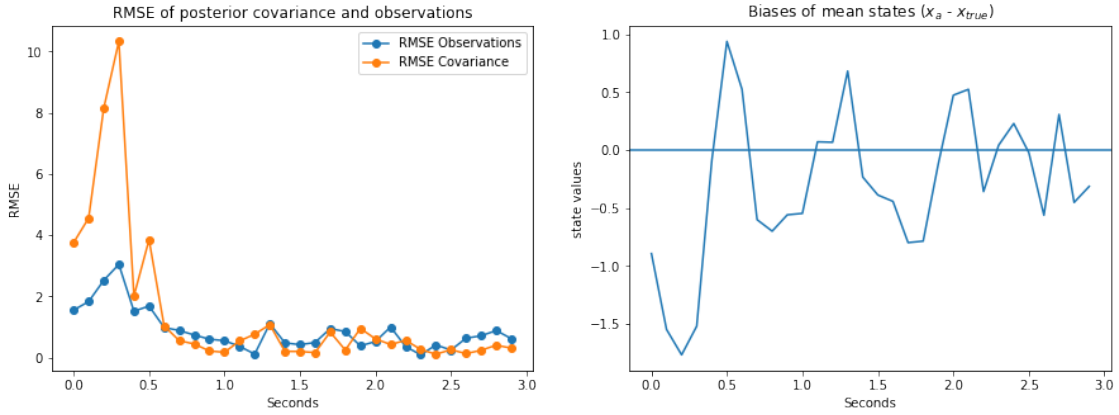




Analyze the EKF

```
In [10]: obs_times = np.arange(0,N_sys,dyn2meas)
         Xa, Xtrue =  xa_EKF , x_true[:,obs_times]
         RMSE = np.sqrt(np.mean((Xa - Xtrue)**2 , axis=0))
         P_true = Pa_EKF.copy()

         covlist = []
         for i in range(N_obs):
             dX = np.matrix(Xa[:,i] - Xtrue[:,i]).T # makes a (3x1)
             Ptrue = dX @ dX.T
             Ptrue_diag = np.diag(Ptrue)
             Pa_diag    = np.diag(Pa_EKF[:,:,i])
             RMSE_cov = np.sqrt(np.mean((Pa_diag - Ptrue_diag)**2 , axis=0))
             covlist.append(RMSE_cov)

         plt.figure(figsize=[15,5])
         plt.subplot(1,2,1)
         plt.title('RMSE of posterior covariance and observations')
         plt.plot(t_obs,RMSE,marker='o',label='RMSE Observations')
         plt.plot(t_obs,covlist,marker='o',label='RMSE Covariance')
         plt.ylabel('RMSE');plt.xlabel('Seconds')
```

```
plt.legend()

plt.subplot(1,2,2)
plt.title('Biases of mean states ($x_a$ - $x_{true}$)')
plt.plot(t_obs,np.mean(Xa,axis=0)-np.mean(Xtrue,axis=0))
plt.ylabel('state values');plt.xlabel('Seconds')
plt.axhline(0)

plt.show()
```



Investigate the performance of filter in terms of the bias and mean-square-error of data assimilation analysis. How does the estimate-error computed from x_a and the truth value of x compare to the error variance suggested by the posterior covariance? Analysis given 15 observations: In the forecast step, the root mean-square-error (RMSE) from the model (denoted as $\sqrt{(x_i - x_{i,true})^2}$), the x_a diverges from the true states ($x_{i,true}$). At the point of assimilation, the simulated x_a is adjusted closer to the observational value and resembles closer the true state. This adjustment of RMSE during assimilation is proportional to the observational noise generated. Here, each observation was generated by the true state plus random noise. The posterior variance ($\sqrt{P_{i,i}}$) grows during forecast steps until point of assimilation update step. The magnitude of posterior variance and state RMSE in time do not have significant biases. However, due to non-linearity in the model, some forecast propagations of x_i diverges dramatically from the true states ($x_{i,true}$).

```
In [11]: def sensitivity(B_var,R_var,dyn2meas):

             B = np.eye(3) * B_var
             R = np.eye(3) * R_var

             # ================== To change observation size ==================
             # Examine the sensitivity of the filter performance to the accuracy and frequency o
             dyn2meas #= 20   # Change frequency of observations.

             h_obs = h_sys * dyn2meas      # sec; an observation is generated every h_obs sec
```

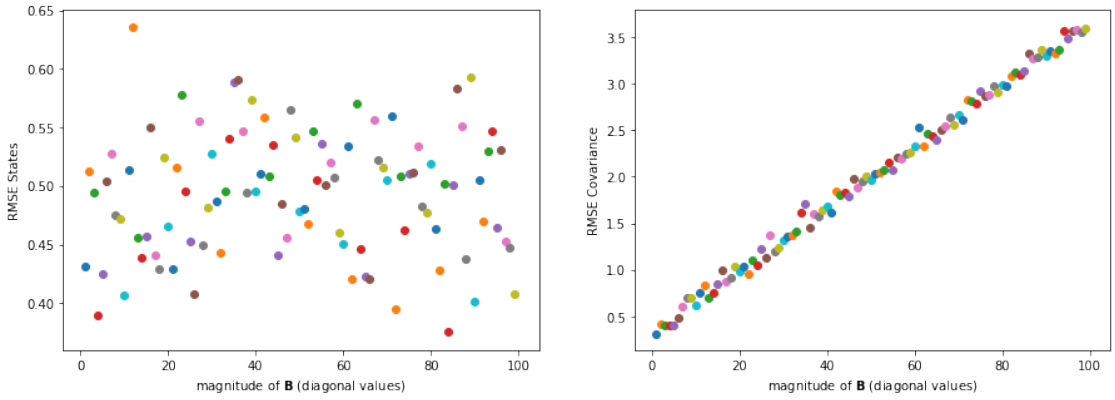
```
t_sys = np.arange(t0,tf_assim,h_sys)     # time stamps for system dynamics
t_obs = np.arange(t0,tf_assim,h_obs)     # time stamps for observations
N_sys = len(t_sys)                # Number of system timesteps
N_obs = len(t_obs)                # Number of observations
#print (f'N-obs={N_obs} N-state={N_sys}')

## Initial conditions
y = np.zeros([M0, N_obs])  # Measurements of x1 (with measurement noise)

# ================== Generate True state ==================
r = np.array([[x1_0, x2_0, x3_0]], float).T   # "Current" position
x_true = np.zeros([N_state, N_sys])  # True history of x1 (without any noise)
x_true[:,0] = np.squeeze(r)

for i in range(1,N_sys):
    x_true[:,i] = advance_state(x_true[:,i-1], coeffs, h_sys)

# ================== Generate Measurements ==================
for i in range(N_obs):
    curr_state = x_true[:,i*dyn2meas][:,np.newaxis]
    y[:,i] = np.squeeze(curr_state + R @ (np.random.randn(3,1)))

## Store histories
xf_EKF = np.array(np.zeros([3, N_sys]))
Pf_EKF = np.zeros(shape=[3,3,N_sys])
xa_EKF = np.array(np.zeros([3, N_obs]))
Pa_EKF = np.zeros(shape=[3,3, N_obs])

## ICs for the EKF
xa = np.array([[y[0,0], y[1,0], y[2,0]]], float).T
Pa = B.copy()
Pf = B.copy()

xf_EKF[:,0] = y[:,0].copy()
Pf_EKF[:,:,0] = Pa.copy()
xa_EKF[:,0] = y[:,0].copy()
Pa_EKF[:,:,0] = Pa.copy()

#### Run the EKF
## Initialization
## Current values
for j in range(1,dyn2meas+1):
    curr_state = xf_EKF[:,j-1]
    curr_P = Pf_EKF[:,:,j-1]
    prev_analysis_state = xa_EKF[:,0]
    M = calc_TLM(prev_analysis_state.copy())

    ## Forecast to next timestep
```

```
    xf = advance_state(curr_state.copy(), coeffs, h_sys)  # Forecast to next time s
    Pf = advance_cov(curr_P.copy(), M, Q, h_sys)
    xf_EKF[:,j], Pf_EKF[:,:,j] = xf.copy(), Pf.copy()

for i in range(1,N_obs):
    #### Update
    x1, x2, x3 = xf[0], xf[1], xf[2]
    y_t = y[:,i]
    M   = np.array([          # The tangent linear model evaluated at x=xf_k
        [-A_lor  ,  A_lor  ,  0.0],
        [B_lor-x3, -1.0,  -x1],
        [x2  , x1  ,  -C_lor]])

    K = Pf @ H.T @ np.linalg.inv(R + H@Pf@H.T)  # Kalman gain
    xa = xf + K @ (y_t - H @ xf)   # Update mean
    Pa = (I - K @ H) @ Pf   # Update covar
    xa_EKF[:,i], Pa_EKF[:,:,i] = xa.copy(), Pa.copy()  # Store update

    #### Forecast
    for j in range(1,dyn2meas+1):

        ## Special case: end condition
        if i*dyn2meas+j >= N_sys:
            break

        ## Special case: first forecast after an obs
        if j == 1:
            ## Current values
            curr_state = xa_EKF[:,i]
            curr_P = Pa_EKF[:,:,i]
            M = calc_TLM(curr_state.copy())

            ## Forecast
            xf = advance_state(curr_state.copy(), coeffs, h_sys)  # Forecast to nea
            Pf = advance_cov(curr_P.copy(), M, Q, h_sys)
            xf_EKF[:,i*dyn2meas+j], Pf_EKF[:,:,i*dyn2meas+j] = xf.copy(), Pf.copy()

        else:
            ## Current values
            curr_state = xf_EKF[:,i*dyn2meas+j-1]
            curr_P = Pf_EKF[:,:,i*dyn2meas+j-1]
            prev_analysis_state = xa_EKF[:,i]
            M = calc_TLM(prev_analysis_state.copy())

            ## Forecast to next timestep
            xf = advance_state(curr_state.copy(), coeffs, h_sys)  # Forecast to nea
            Pf = advance_cov(curr_P.copy(), M, Q*5, h_sys)
            xf_EKF[:,i*dyn2meas+j], Pf_EKF[:,:,i*dyn2meas+j] = xf.copy(), Pf.copy()
```

```
        obs_times = np.arange(0,N_sys,dyn2meas)
        Xa, Xtrue =  xa_EKF , x_true[:,obs_times]
        RMSE = np.sqrt(np.mean((Xa - Xtrue)**2 , axis=0))
        P_true = Pa_EKF.copy()
        covlist = []
        for i in range(N_obs):
            dX = np.matrix(Xa[:,i] - Xtrue[:,i]).T # makes a (3x1)
            Ptrue = dX @ dX.T
            Ptrue_diag = np.diag(Ptrue)
            Pa_diag    = np.diag(Pa_EKF[:,:,i])
            RMSE_cov = np.sqrt(np.mean((Pa_diag - Ptrue_diag)**2 , axis=0))
            covlist.append(RMSE_cov)

        mean_RMSE = np.mean(RMSE)
        mean_RMSE_Cov = np.mean(covlist)
        return mean_RMSE,mean_RMSE_Cov
```

In [12]:
```
plt.figure(figsize=[15,5])
for i in range(1,100):
    mean_RMSE,mean_RMSE_Cov = sensitivity(B_var=i,R_var=1,dyn2meas=10)
    plt.subplot(121);plt.scatter(i,mean_RMSE)
    plt.subplot(122);plt.scatter(i,mean_RMSE_Cov)

plt.subplot(121);plt.xlabel('magnitude of $\mathbf{B}$ (diagonal values)');plt.ylabel('
plt.subplot(122);plt.xlabel('magnitude of $\mathbf{B}$ (diagonal values)');plt.ylabel('
plt.show()
```

```
/anaconda3/lib/python3.7/site-packages/matplotlib/cbook/deprecation.py:107: MatplotlibDeprecatio
  warnings.warn(message, mplDeprecation, stacklevel=1)
```

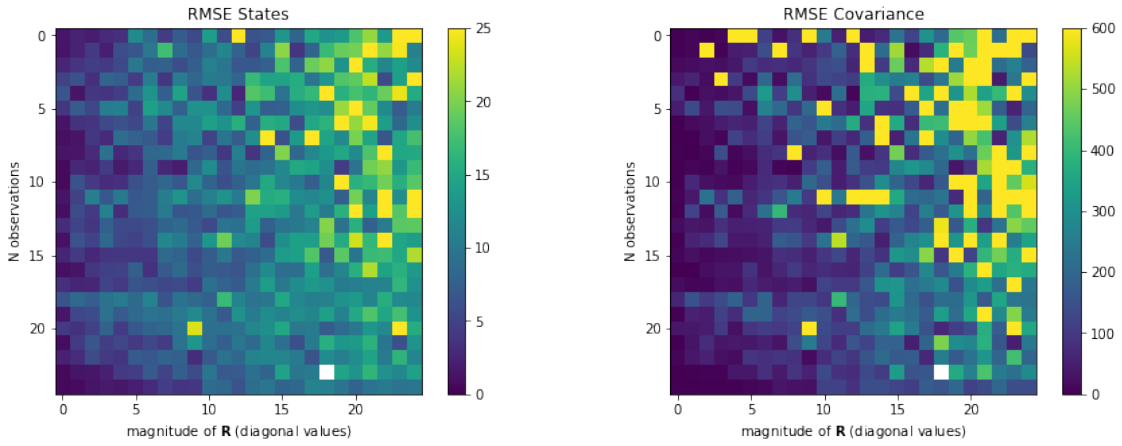


Examine the sensitivity of the filter performance to the errors in the background. (Compare at least 10 cases) We find the RMSE in the update states (again, $\sqrt{(x_{i,a} - x_{i,true})^2}$) is somewhat invariant to an increasing background covariance **B**. If anything, the spread in RMSE increases more with larger background covariance. In these plots above, we increase our background diagonal elements from 1 to 100 over 100x times and have a **R** magnitude of 1 and there is an observations taken every 10 steps out of 300 (30 observations). As we increase the magnitude of the background covariance, our posterior covariance becomes larger in magnitude.

1.1.4 Examine the sensitivity of the filter performance to the accuracy and frequency of observations. Is it better to have frequent but inaccurate observations or infrequent but accurate ones? (Compare at least 10×10 cases.)

```
In [13]: RMSE_mat,P_RMSE_mat = np.zeros((25,25)), np.zeros((25,25))

         for i,forecast_steps in enumerate(np.arange(250,0,-10)):
             for rvar in range(25):
                 RMSE_mat[i,rvar] , P_RMSE_mat[i,rvar] = sensitivity(B_var=1.,R_var=rvar+1.,dyn2

         plt.figure(figsize=[15,5])
         plt.subplot(121);plt.imshow(RMSE_mat,vmin=0, vmax=25)
         plt.ylabel('N observations');plt.xlabel('magnitude of $\mathbf{R}$ (diagonal values)');
         plt.colorbar()

         plt.subplot(122);plt.imshow(P_RMSE_mat,vmin=0, vmax=600)
         plt.ylabel('N observations');plt.xlabel('magnitude of $\mathbf{R}$ (diagonal values)');
         plt.colorbar()

         plt.show()
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:17: RuntimeWarning: overflow encoun
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:18: RuntimeWarning: overflow encoun
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:17: RuntimeWarning: invalid value e
```



Heats maps show the RMSE decreases with more accurate observations. This plot is not perfect because instead of increasing observations in each iteration, I am decreasing the forecasting steps by 10 each. IN our code, this increases the observations. Also, the noise in these heat maps are due to observational noise that is generated randomly. However, a trend is clear in these heat maps.

1.1.5 3. Implement your own EnKF (with perturb observations) or EnSRF to estimate x_a.

- Initialize the ensemble by introducing small perturbations to the state and integrating the model until its error growth saturates.
- Compare the performance of the ensemble method to EKF with respect to each sub-question above. Launch ensemble forecasts (instead of deterministic forecasts) at the end of the assimilation period.
- Evaluate the ensemble by using rank histograms.
- Examine the impact of increasing or decreasing ensemble size on the filter performance as well as the forecast performance.

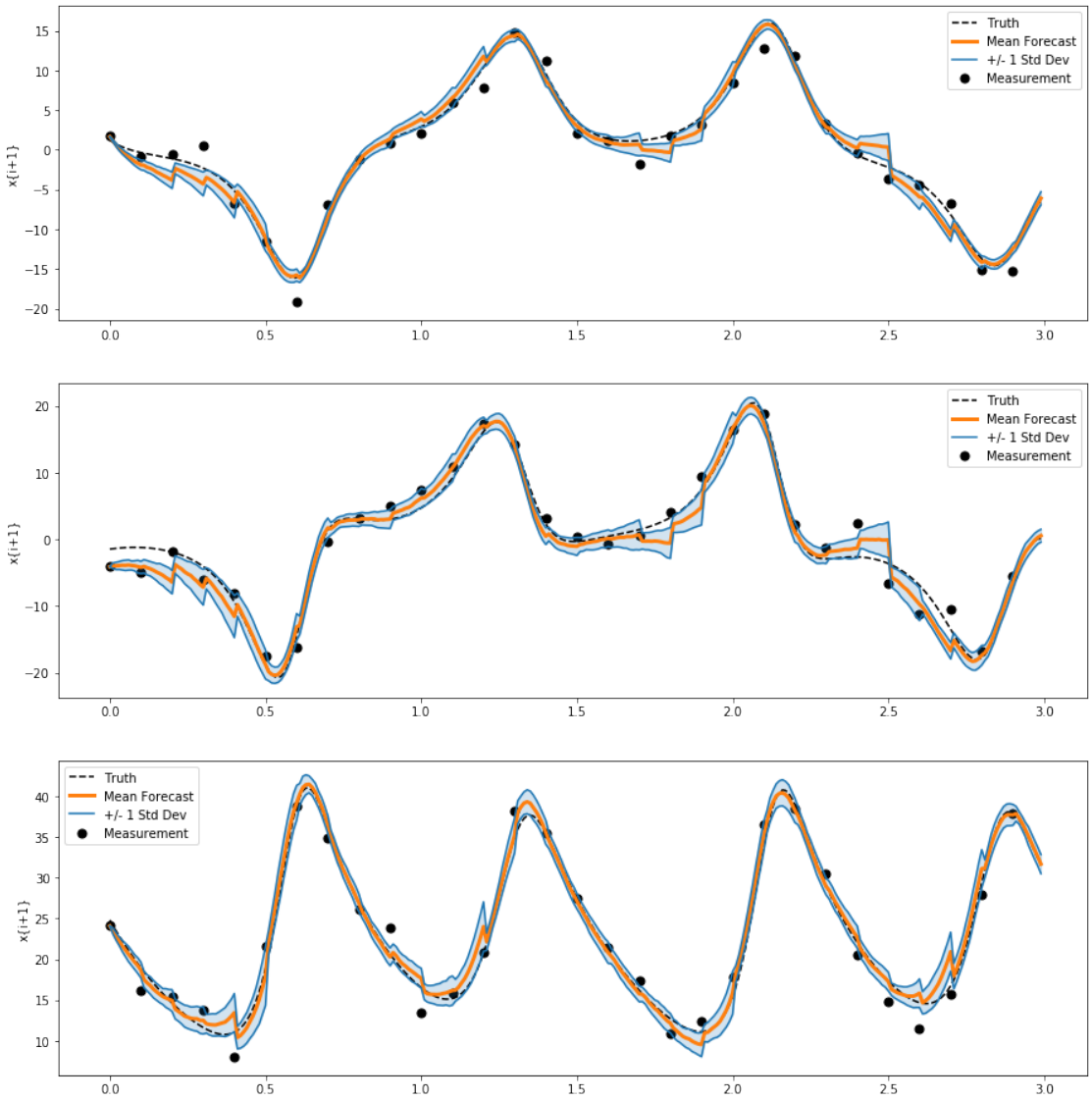
Stochastic EnKF Algorithm (Katzfuss 2016) * *Note* — Notational consistency: $\widetilde{x} = x^f$ and $\hat{x} = x^a$ * Start with an initial ensemble $\hat{x}_0^{(1)}, ..., \hat{x}_0^{(N)}$ * Then at each time $t = 1, 2, ...$, given an ensemble $\hat{x}_{t-1}^{(1)}, ..., \hat{x}_{t-1}^{(N)}$ of draws from the filtering (posterior) distribution at time $t-1$ the stochastic EnKF carries out the following two steps for $i = 1, ..., N$. * *Forecast* — Draw process noise $w_t^{(i)} \sim N_n(0, Q_t)$ and calculate $\widetilde{x}_t^{(i)} = M_t \hat{x}_{t-1}^{(i)} + w_t^{(i)}$ * *Update* — Draw sensor noise $v_t^{(i)} \sim N_{m_t}(0, R_t)$ and calculate $\hat{x}_t^{(i)} = \widetilde{x}_t^{(i)} + \hat{K}_t(y_t - \widetilde{y}_t^{(i)})$ * where y_t is the actual observation * and $\widetilde{y}_t^{(i)} = H_t \widetilde{x}_t^{(i)} - v_t^{(i)}$ is the simulated observation * and where $\hat{K}_t = C_t H_t^T (H_t C_t H_t^T + R_t)^{-1}$ * and C_t is an estimate of the state forecast covariance matrix $\widetilde{\Sigma}_t$. The simplest example is $C_t = \widetilde{S}_t$, where $\widetilde{S}_t$ is the sample covariance matrix of $\widetilde{x}_t^{(1)}, ..., \widetilde{x}_t^{(N)}$

In [14]:
```
## Generate initial ensemble
N_ens = 25   # number of ensemble members
xtld_hist = np.zeros((3, N_ens, N_sys))  # History of forecasts (x tilde)
xhat_hist = np.zeros((3, N_ens, N_obs))  # History of updated x's (x hat)
Ptld_hist = np.zeros((3, 3, N_sys))
muw = np.zeros(3)   # the mean of process noise
mvw = np.zeros(3)   # the mean of sensor noise
Q_ens = Q.copy()    # Process noise for the EnKF (tuned to give flat rank histogram)
Q_ens = 0.1*Q_ens
Qh0 = Q_ens         # initlal Q
w0 = np.random.multivariate_normal(muw, Qh0, size=N_ens).T  # Perturbations for the IC
x0_pure = np.array([[y[0,0], y[1,0], y[2,0]]]).T   # Unperturbed IC
xh0 = x0_pure + w0 # Initial position of ensemble members
xtld_hist[:,:,0] = xh0
xhat_hist[:,:,0] = xh0

# #### Run the EnKF
# ## Initialization
# ## Current values
for j in range(1,dyn2meas+1):
    curr_state = xtld_hist[:,:,j-1].copy()
```

```
        w = np.random.multivariate_normal(muw, Q_ens, size=N_ens).T
        xtld = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy forecast
        Ptld = np.cov(xtld)      # Sample covariance calculation
        xtld_hist[:,:,j] = xtld.copy()
        Ptld_hist[:,:,j] = Ptld

    ## Move through time
    for i in range(1,N_obs):
        ## Update
        Ct = np.cov(xtld)    # Calculate sample covariance
        Kt = Ct @ H.T @ np.linalg.inv(H @ Ct @ H + R)  # Calculate Kalman gain
        vt = np.random.multivariate_normal(mvw, R, size=N_ens).T  # Draw measurement noise
        ytld = H@xtld - vt    # Calculate simulated observations
        yact = y[:,i][:,np.newaxis]  # Actual observation
        xhat = xtld + Kt @ (yact - ytld)
        xhat_hist[:,:,i] = xhat

        ## Forecast
        for j in range(1,dyn2meas+1):
            ## Special case: end condition
            if i*dyn2meas+j >= N_sys:
                break

            ## Special case: First forecast after observation
            if j == 1:  # For first forecast after observation, forecast from the obtained
                curr_state = xhat_hist[:,:,i].copy()
                w = np.random.multivariate_normal(muw, Q_ens, size=N_ens).T
                xtld = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy forecast
                Ptld = np.cov(xtld)      # Sample covariance calculation
                xtld_hist[:,:,i*dyn2meas+j] = xtld.copy()
                Ptld_hist[:,:,i*dyn2meas+j] = Ptld
            else:
                curr_state = xtld_hist[:,:,i*dyn2meas+j-1].copy()
                w = np.random.multivariate_normal(muw, Q_ens, size=N_ens).T
                xtld = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy forecast
                Ptld = np.cov(xtld)      # Sample covariance calculation
                xtld_hist[:,:,i*dyn2meas+j] = xtld.copy()
                Ptld_hist[:,:,i*dyn2meas+j] = Ptld
```

In [15]:
```
## Plot true state; true state + measurement noise; EnKF update predictions
plt.figure(figsize=[15,16])

## Plot x1
for i in range(3):
    plt.subplot(3,1,i+1) ; plt.plot(t_sys, x_true[i,:], ls='--', color='k', label='Trut
    plt.scatter(t_obs, y[i,:], s=50, color='k', label='Measurement')
    plt.plot(t_sys, np.mean(xtld_hist, axis=1)[i,:], color='C1', linewidth=3, label='Me
    plt.plot(t_sys, np.mean(xtld_hist, axis=1)[i,:] + np.sqrt(Ptld_hist[i,i,:]), color=
```

```
        plt.plot(t_sys, np.mean(xtld_hist, axis=1)[i,:] - np.sqrt(Ptld_hist[i,i,:]), color=
        plt.fill_between(t_sys, np.mean(xtld_hist, axis=1)[i,:] + np.sqrt(Ptld_hist[i,i,:])
                                np.mean(xtld_hist, axis=1)[i,:] - np.sqrt(Ptld_hist[i,i,:])
        plt.ylabel('x{i+1}')
        plt.legend()

    plt.show()
```

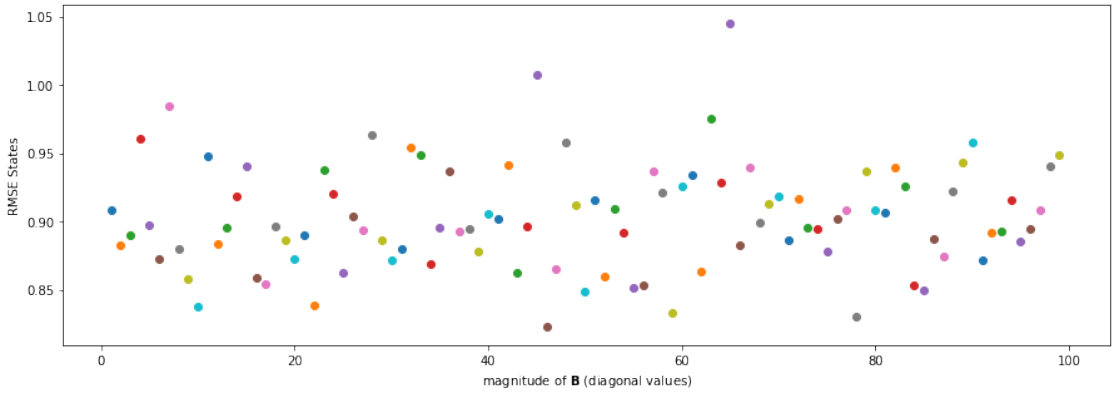



```
In [16]: def sensitivity_ens(B_var,R_var,dyn2meas,N_ens):

        B = np.eye(3) * B_var
        R = np.eye(3) * R_var
```

```
# ================== To change observation size ==================
# Examine the sensitivity of the filter performance to the accuracy and frequency o
dyn2meas #= 20   # Change frequency of observations.

h_obs = h_sys * dyn2meas     # sec; an observation is generated every h_obs sec
t_sys = np.arange(t0,tf_assim,h_sys)     # time stamps for system dynamics
t_obs = np.arange(t0,tf_assim,h_obs)     # time stamps for observations
N_sys = len(t_sys)               # Number of system timesteps
N_obs = len(t_obs)               # Number of observations
#print (f'N-obs={N_obs} N-state={N_sys}')

## Generate initial ensemble
#N_ens = 25   # number of ensemble members
xtld_hist = np.zeros((3, N_ens, N_sys))  # History of forecasts (x tilde)
xhat_hist = np.zeros((3, N_ens, N_obs))  # History of updated x's (x hat)
Ptld_hist = np.zeros((3, 3, N_sys))
muw = np.zeros(3)   # the mean of process noise
mvw = np.zeros(3)   # the mean of sensor noise
Q_ens = Q.copy()    # Process noise for the EnKF (tuned to give flat rank histogram
Q_ens = 0.1*Q_ens
Qh0 = Q_ens         # initlal Q
w0 = np.random.multivariate_normal(muw, Qh0, size=N_ens).T  # Perturbations for the
x0_pure = np.array([[y[0,0], y[1,0], y[2,0]]]).T   # Unperturbed IC
xh0 = x0_pure + w0 # Initial position of ensemble members
xtld_hist[:,:,0] = xh0
xhat_hist[:,:,0] = xh0

# #### Run the EnKF
# ## Initialization
# ## Current values
for j in range(1,dyn2meas+1):
    curr_state = xtld_hist[:,:,j-1].copy()
    w = np.random.multivariate_normal(muw, Q_ens, size=N_ens).T
    xtld = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy forecast
    Ptld = np.cov(xtld)      # Sample covariance calculation
    xtld_hist[:,:,j] = xtld.copy()
    Ptld_hist[:,:,j] = Ptld

## Move through time
for i in range(1,N_obs):
    ## Update
    Ct = np.cov(xtld)    # Calculate sample covariance
    Kt = Ct @ H.T @ np.linalg.inv(H @ Ct @ H + R)  # Calculate Kalman gain
    vt = np.random.multivariate_normal(mvw, R, size=N_ens).T  # Draw measurement no
    ytld = H@xtld - vt    # Calculate simulated observations
    yact = y[:,i][:,np.newaxis]  # Actual observation
    xhat = xtld + Kt @ (yact - ytld)
    xhat_hist[:,:,i] = xhat
```

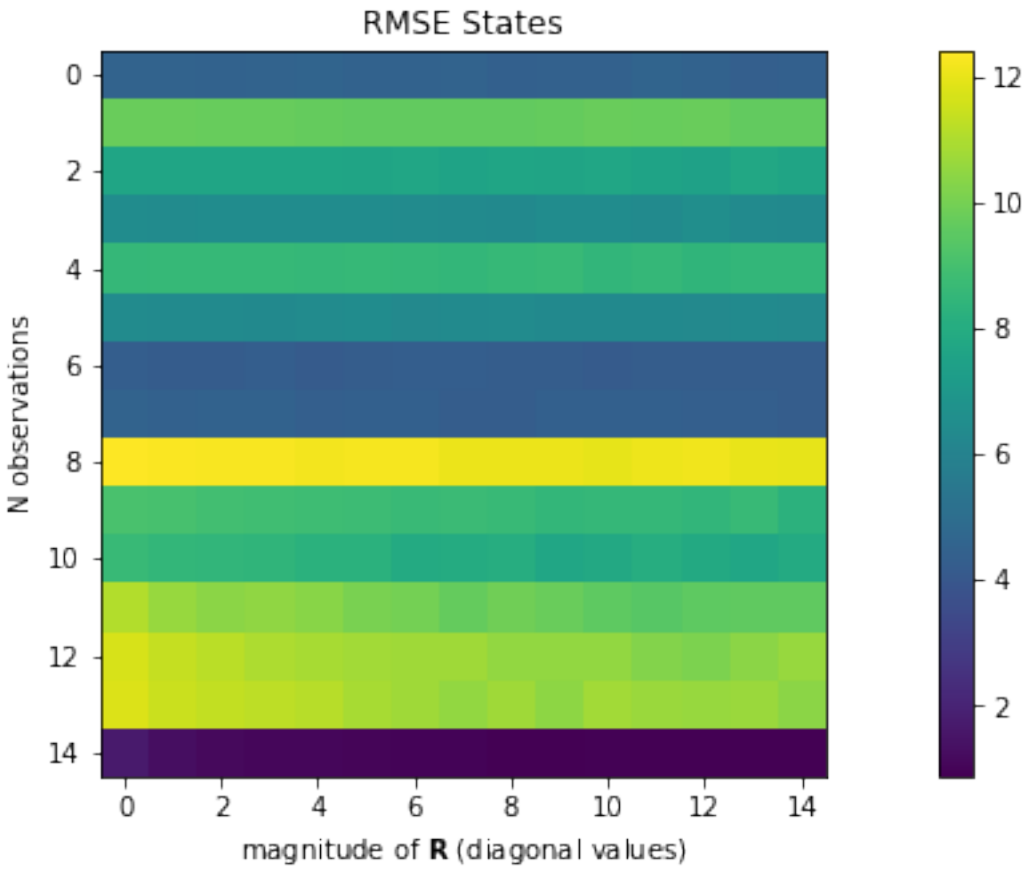
```
            ## Forecast
            for j in range(1,dyn2meas+1):
                ## Special case: end condition
                if i*dyn2meas+j >= N_sys:
                    break

                ## Special case: First forecast after observation
                if j == 1:  # For first forecast after observation, forecast from the obtai
                    curr_state = xhat_hist[:,:,i].copy()
                    w = np.random.multivariate_normal(muw, Q_ens, size=N_ens).T
                    xtld = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy fore
                    Ptld = np.cov(xtld)      # Sample covariance calculation
                    xtld_hist[:,:,i*dyn2meas+j] = xtld.copy()
                    Ptld_hist[:,:,i*dyn2meas+j] = Ptld
                else:
                    curr_state = xtld_hist[:,:,i*dyn2meas+j-1].copy()
                    w = np.random.multivariate_normal(muw, Q_ens, size=N_ens).T
                    xtld = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy fore
                    Ptld = np.cov(xtld)      # Sample covariance calculation
                    xtld_hist[:,:,i*dyn2meas+j] = xtld.copy()
                    Ptld_hist[:,:,i*dyn2meas+j] = Ptld

        obs_times = np.arange(0,N_sys,dyn2meas)
        Xa, Xtrue =  np.mean(xhat_hist,axis=1) , x_true[:,obs_times]
        RMSE = np.sqrt(np.mean((Xa - Xtrue)**2 , axis=0))
        mean_RMSE = np.mean(RMSE)
        return mean_RMSE
```

In [17]:
```
plt.figure(figsize=[15,5])
for i in range(1,100):
    mean_RMSE = sensitivity_ens(B_var=i,R_var=1,dyn2meas=10,N_ens=25)
    plt.scatter(i,mean_RMSE)

plt.xlabel('magnitude of $\mathbf{B}$ (diagonal values)')
plt.ylabel('RMSE States')
plt.show()
```



Examine the sensitivity of the filter performance to the errors in the background. (Compare at least 10 cases) We find the RMSE in the states (again, $\sqrt{(x_{i,a} - x_{i,true})^2}$) is mainly invariant to an increasing background covariance **B**. In these plots above, we increase our background diagonal elements from 1 to 100 over 100x times and have a **R** magnitude of 1 and there is an observations taken every 10 steps out of 300 (30 observations). For each state, we take the average of 25 ensemble update states.

```
In [18]: RMSE_mat = np.zeros((15,15))

         for i,forecast_steps in enumerate(np.arange(150,0,-10)):
             for rvar in range(15):
                 RMSE_mat[i,rvar] = sensitivity_ens(B_var=1.,R_var=rvar*0.1,dyn2meas=forecast_st

         plt.figure(figsize=[15,5])
         plt.imshow(RMSE_mat)#,vmin=0, vmax=25)
         plt.ylabel('N observations');plt.xlabel('magnitude of $\mathbf{R}$ (diagonal values)');
         plt.colorbar();plt.show()
```



Heats maps show the that as you increase observations into a 25-ensemble Stochastic EnKF, the observational covariance does not affect RMSE calculated in the update states (again, $\sqrt{(x_{i,a} - x_{i,true})^2}$). This result is concerning as we expect **R** to affect RMSE in each update step.

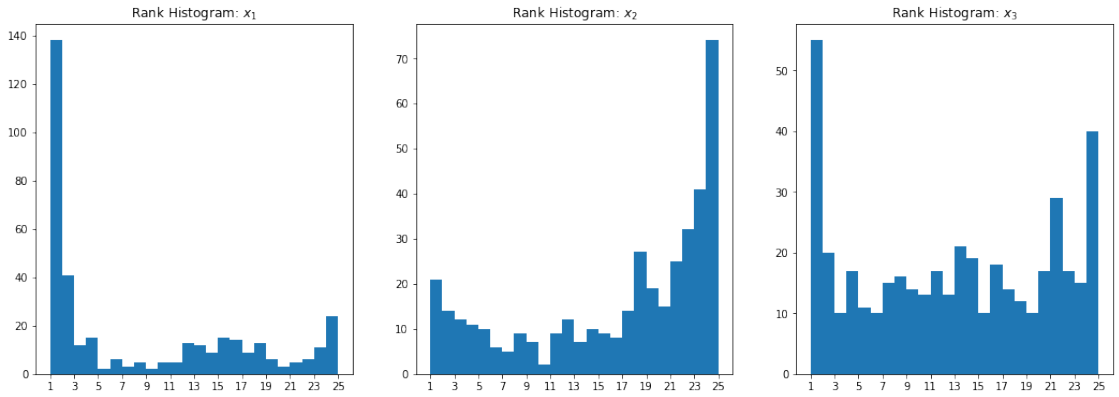
1.1.6 Generate Rank Histograms

```
In [19]: ## Rank lists
         rank1 = []
         rank2 = []
         rank3 = []
         N_rank_runs = 15

         Q_ens = Q.copy()
         Q_ens = 0.1*Q_ens

         for rank_run in range(N_rank_runs):
             ## Generate initial ensemble
             N_ens = 25   # number of ensemble members
             xtld_hist = np.zeros((3, N_ens, N_sys))  # History of forecasts (x tilde)
             xhat_hist = np.zeros((3, N_ens, N_obs))  # History of updated x's (x hat)
```

```
Ptld_hist = np.zeros((3, 3, N_sys))
muw = np.zeros(3)    # the mean of process noise
mvw = np.zeros(3)    # the mean of sensor noise
Qh0 = Q_ens          # initlal Q
w0 = np.random.multivariate_normal(muw, Qh0, size=N_ens).T  # Perturbations for the
x0_pure = np.array([[y[0,0], y[1,0], y[2,0]]]).T   # Unperturbed IC
xh0 = x0_pure + w0 # Initial position of ensemble members
xtld_hist[:,:,0] = xh0
xhat_hist[:,:,0] = xh0

# #### Run the EnKF
# ## Initialization
# ## Current values
for j in range(1,dyn2meas+1):
    curr_state = xtld_hist[:,:,j-1].copy()
    w = np.random.multivariate_normal(muw, Q_ens, size=N_ens).T
    xtld = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy forecast
    Ptld = np.cov(xtld)      # Sample covariance calculation
    xtld_hist[:,:,j] = xtld.copy()
    Ptld_hist[:,:,j] = Ptld

## Move through time
for i in range(1,N_obs):
    ## Update
    Ct = np.cov(xtld)    # Calculate sample covariance
    Kt = Ct @ H.T @ np.linalg.inv(H @ Ct @ H + R)  # Calculate Kalman gain
    vt = np.random.multivariate_normal(mvw, R, size=N_ens).T  # Draw measurement no
    ytld = H@xtld - vt    # Calculate simulated observations
    yact = y[:,i][:,np.newaxis]  # Actual observation
    xhat = xtld + Kt @ (yact - ytld)
    xhat_hist[:,:,i] = xhat

    ## Forecast
    for j in range(1,dyn2meas+1):
        ## Special case: end condition
        if i*dyn2meas+j >= N_sys:
            break

        ## Special case: First forecast after observation
        if j == 1:  # For first forecast after observation, forecast from the obtai
            curr_state = xhat_hist[:,:,i].copy()
            w = np.random.multivariate_normal(muw, Q_ens, size=N_ens).T
            xtld = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy fore
            Ptld = np.cov(xtld)      # Sample covariance calculation
            xtld_hist[:,:,i*dyn2meas+j] = xtld.copy()
            Ptld_hist[:,:,i*dyn2meas+j] = Ptld
        else:
            curr_state = xtld_hist[:,:,i*dyn2meas+j-1].copy()
```

```
                    w = np.random.multivariate_normal(muw, Q_ens, size=N_ens).T
                    xtld = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy fore
                    Ptld = np.cov(xtld)      # Sample covariance calculation
                    xtld_hist[:,:,i*dyn2meas+j] = xtld.copy()
                    Ptld_hist[:,:,i*dyn2meas+j] = Ptld

            ## Store ranks of observations before re-initializing the simulation
            for q in range(N_obs):
                sorted_ens1 = np.sort(xhat_hist[0,:,q])
                sorted_ens2 = np.sort(xhat_hist[1,:,q])
                sorted_ens3 = np.sort(xhat_hist[2,:,q])

                rank1.append(np.sum(y[0,q] > sorted_ens1) + 1)
                rank2.append(np.sum(y[1,q] > sorted_ens2) + 1)
                rank3.append(np.sum(y[2,q] > sorted_ens3) + 1)
```

In [20]:
```
plt.figure(figsize=(18,6))

plt.subplot(1,3,1)
plt.hist(rank1, bins=np.arange(1,N_ens+1,1))
plt.xticks(np.arange(1,N_ens+1,2))
plt.title("Rank Histogram: $x_1$")


plt.subplot(1,3,2)
plt.hist(rank2, bins=np.arange(1,N_ens+1,1))
plt.xticks(np.arange(1,N_ens+1,2))
plt.title("Rank Histogram: $x_2$")


plt.subplot(1,3,3)
plt.hist(rank3, bins=np.arange(1,N_ens+1,1))
plt.xticks(np.arange(1,N_ens+1,2))
plt.title("Rank Histogram: $x_3$")


plt.show()
```



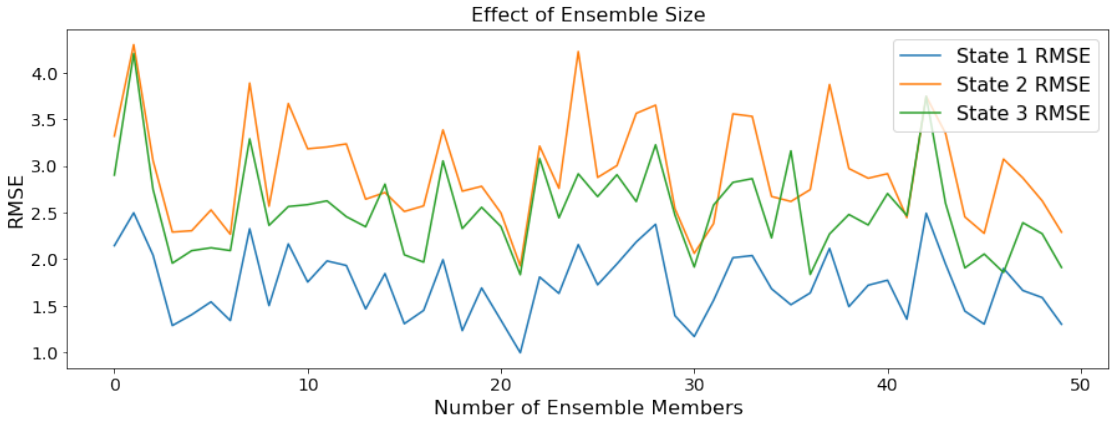
The rank histograms show that we do not have an ideal ensemble. State 2 and state 3 are somewhat flat, which is the desired behavior. However, the Rank 1 and Rank 25 bins in State 1 are highly populated, suggesting overconfidence in State 1's estimate.

1.1.7 Effect of Ensemble Size

```
In [21]: rmse1_list = []
         rmse2_list = []
         rmse3_list = []
         max_ens_membs = 50

         for foo in range(max_ens_membs):
             ## Generate initial ensemble
             N_ens     = 3                              # Number of ensemble members
             xtld_hist = np.zeros((3, N_ens, N_sys))  # History of forecasts (x tilde)
             xhat_hist = np.zeros((3, N_ens, N_obs))  # History of updated x's (x hat)
             Ptld_hist = np.zeros((3, 3, N_sys))      # History of forecasted uncertainty
             muw       = np.zeros(3)                  # The mean of process noise
             mvw       = np.zeros(3)                  # the mean of sensor noise
             Qh0       = Q                            # initlal Q
             w0        = np.random.multivariate_normal(muw, Qh0, size=N_ens).T  # Perturbations
             x0_pure   = np.array([[y[0,0], y[1,0], y[2,0]]]).T   # Unperturbed IC
             xh0       = x0_pure + w0                 # Initial position of ensemble members
             xtld_hist[:,:,0] = xh0
             xhat_hist[:,:,0] = xh0

             # #### Run the EnKF
             # ## Initialization
             # ## Current values
             for j in range(1,dyn2meas+1):
                 curr_state       = xtld_hist[:,:,j-1].copy()
                 w                = np.random.multivariate_normal(muw, Q, size=N_ens).T
                 xtld             = advance_state(curr_state.copy(), coeffs, h_sys) + w # Noisy
                 Ptld             = np.cov(xtld)      # Sample covariance calculation
```

```
    xtld_hist[:,:,j] = xtld.copy()
    Ptld_hist[:,:,j] = Ptld

## Move through time
for i in range(1,N_obs):
    ## Update
    Ct               = np.cov(xtld)    # Calculate sample covariance
    Kt               = Ct @ H.T @ np.linalg.inv(H @ Ct @ H + R)  # Calculate Kalman
    vt               = np.random.multivariate_normal(mvw, R, size=N_ens).T  # Draw
    ytld             = H@xtld - vt    # Calculate simulated observations
    yact             = y[:,i][:,np.newaxis]  # Actual observation
    xhat             = xtld + Kt @ (yact - ytld)
    xhat_hist[:,:,i] = xhat

    ## Forecast
    for j in range(1,dyn2meas+1):
        ## Special case: end condition
        if i*dyn2meas+j >= N_sys:
            break

        ## Special case: First forecast after observation
        if j == 1:  # For first forecast after observation, forecast from the obtai
            curr_state = xhat_hist[:,:,i].copy()
            w          = np.random.multivariate_normal(muw, Q, size=N_ens).T
            xtld       = advance_state(curr_state.copy(), coeffs, h_sys) + w # Nois
            Ptld       = np.cov(xtld)      # Sample covariance calculation
            xtld_hist[:,:,i*dyn2meas+j] = xtld.copy()
            Ptld_hist[:,:,i*dyn2meas+j] = Ptld
        else:
            curr_state = xtld_hist[:,:,i*dyn2meas+j-1].copy()
            w          = np.random.multivariate_normal(muw, Q, size=N_ens).T
            xtld       = advance_state(curr_state.copy(), coeffs, h_sys) + w # Nois
            Ptld       = np.cov(xtld)      # Sample covariance calculation
            xtld_hist[:,:,i*dyn2meas+j] = xtld.copy()
            Ptld_hist[:,:,i*dyn2meas+j] = Ptld

## Calculate rmse
mean1 = np.mean(xtld_hist, axis=1)[0,:]
rmse1 = np.sqrt(np.mean((mean1 - x_true[0,:])**2))
mean2 = np.mean(xtld_hist, axis=1)[1,:]
rmse2 = np.sqrt(np.mean((mean2 - x_true[1,:])**2))
mean3 = np.mean(xtld_hist, axis=1)[2,:]
rmse3 = np.sqrt(np.mean((mean3 - x_true[2,:])**2))

rmse1_list.append(rmse1)
rmse2_list.append(rmse2)
rmse3_list.append(rmse3)
```

```
In [23]: plt.figure(figsize=(15,5))
         plt.title("Effect of Ensemble Size", fontsize=16)
         plt.plot(range(max_ens_membs), rmse1_list, label='State 1 RMSE')
         plt.plot(range(max_ens_membs), rmse2_list, label='State 2 RMSE')
         plt.plot(range(max_ens_membs), rmse3_list, label='State 3 RMSE')
         plt.ylabel("RMSE", fontsize=16)
         plt.xlabel("Number of Ensemble Members", fontsize=16)
         plt.tick_params(labelsize=14)
         plt.legend(fontsize=16)
         plt.show()
```



The RMSE of any particular state **does not change** as the number of ensemble members increases. We expect the RMSE to decrease as more members are added, so this behavior suggests a **bug in the EnKF** code.