Assignment 3

Date ....

*3.1

⇒ a] The posterior loss expectation for adopting action $\hat{y}$, given x, is

$$\rho(\hat{y}|x) = E_{p(y|x)}[L(y,\hat{y})]$$
$$= p_0 \cdot L(\hat{y},0) + p_1 \cdot L(\hat{y},1)$$
$$= p_0 \cdot L(\hat{y},0) + (1-p_0) \cdot L(\hat{y},1)$$
$$= L(\hat{y},1) + p_0 (L(\hat{y},0) - L(\hat{y},1))$$

Therefore,

$$\rho(0|x) = \lambda_{01} - p_0 \cdot \lambda_{01}$$
$$\rho(1|x) = p_0 \cdot \lambda_{10}$$

Because $\rho(0|x)$ and $\rho(1|x)$ are both linear function of p_0, the unique optimal threshold is $\rho(0|x) = \rho(1|x)$.

for example, $p_0 = \dfrac{\lambda_{01}}{\lambda_{01}+\lambda_{10}}$

$$p_1 = \frac{\lambda_{10}}{\lambda_{01}+\lambda_{10}}$$

Date ....

⇒ b] We need to derive the loss matrix where the threshold is 0.1.

Let, $\lambda_{10} = 1$

$\lambda_{01} = 9$

We can execute then,

$$p_0 = \frac{\lambda_{01}}{\lambda_{01}+\lambda_{10}} = \frac{9}{9+1} = 0.9$$

$$p_1 = \frac{\lambda_{10}}{\lambda_{01}+\lambda_{10}} = \frac{1}{9+1} = 0.1$$

$0.1\lambda_{01} + 0.1\lambda_{10} = \lambda_{10}$ .... (I)

$\lambda_{01} = 9\lambda_{10}$

$0.9\lambda_{01} + 0.9\lambda_{10} = \lambda_{01}$ ..... (II)

$\lambda_{10} = \lambda_{01}/9$

$\therefore 9\lambda_{10} = \lambda_{01}$

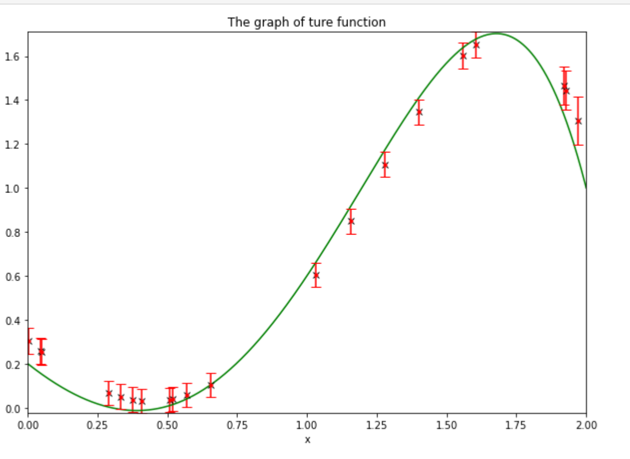
Predicted label $\hat{y}$	true label y	
	0	1
0	0	9
1	1	0

Assignment C: 3.3

```python
import numpy as np
import math
import matplotlib.pyplot as plt

def frange(x_0, y_0, jump):
    while x_0 < y_0:
        yield x_0
        x_0 += jump
```

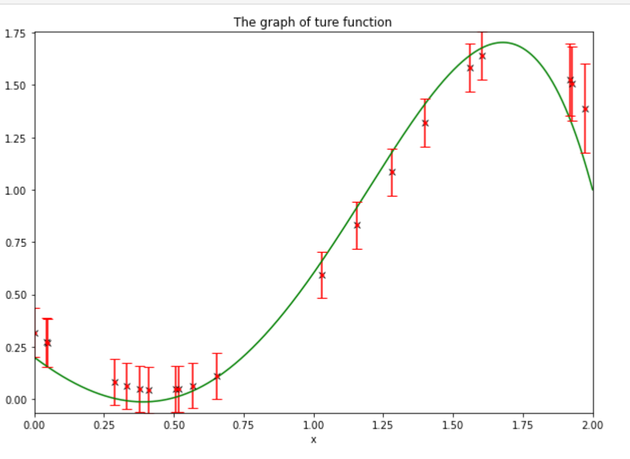
Part i: Computing the covariance matrix and the mean of the posterior Gaussian distribution

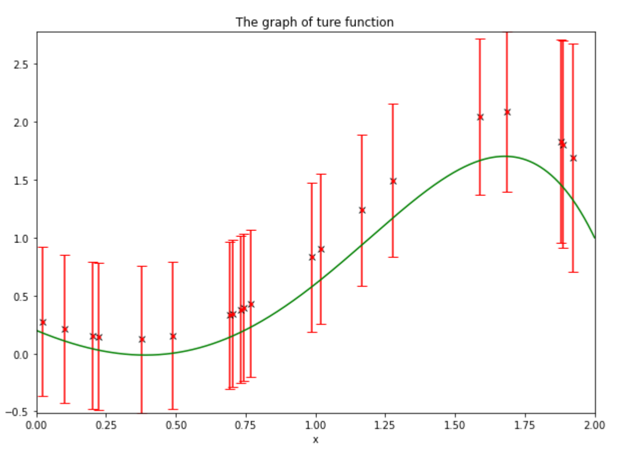
```python
np.random.seed(5)

# true signal curve
x_0 = np.array(list(frange(0, 2, 0.0001)))
x_0 = np.reshape(x_0, newshape=(x_0.shape[0], 1))
y_0 = 0.2 * np.ones(shape=(x_0.shape[0],1)) - x_0 + 0.9 * x_0**2 + 0.7 * x_0**3 - 0.2 * x_0**5

N = 20 # examples of training (20 or 500)

# [a b] sample interval
a = 0
b = 2

# assemble samples
x_01 = np.array(list(frange(a, b, b/N)))
x_01 = np.reshape(x_01, newshape=(x_01.shape[0], 1))
# matrix_0 measurement
PH_i = np.ones(shape=(N, 1))
PH_i = np.concatenate((PH_i, np.array(x_01)), axis=1)
PH_i = np.concatenate((PH_i, np.array(x_01**2)), axis=1)
PH_i = np.concatenate((PH_i, np.array(x_01**3)), axis=1)
PH_i = np.concatenate((PH_i, np.array(x_01**5)), axis=1)
# assembling noise
SIGMA_n = 0.05
n = math.sqrt(SIGMA_n) * np.random.randn(N,1)

# make use of the true theta
THETA_tr = np.array([[0.2], [-1], [0.9], [0.7], [-0.2]])

THETA_dst = np.array([[-0.005], [-10.60], [0.470], [0.097], [-0.083]])
l = THETA_tr.shape[0]
# Using the linear model, produce noisy observations
y_01 = np.dot(PH_i, THETA_tr) + n

# select the Gaussian prior parameters
SIGMA_theta = 0.1
myu_prior_theta = THETA_tr # or myu_prior_theta = THETA_dst;

# calculate the Gaussian posterior's of the covariance matrix_0
SIGMA_posterior_theta = np.linalg.inv((SIGMA_theta**-1) * np.eye(l) + (SIGMA_n**-1) * np.dot(PH_i.conj().transp
# calculate the mean of the posterior
myu_posterior_theta = myu_prior_theta + (SIGMA_n**-1) * np.dot(np.dot(SIGMA_posterior_theta, PH_i.conj().transp

# linear prediction
L_p = 20

# create samples for prediction
x_02 = (b-a) * np.random.rand(L_p, 1)

# calculate the measurement matrix_0 for prediction
PH_ip = np.ones(shape=(L_p, 1))
PH_ip = np.concatenate((PH_ip, np.array(x_02)), axis=1)
PH_ip = np.concatenate((PH_ip, np.array(x_02 ** 2)), axis=1)
PH_ip = np.concatenate((PH_ip, np.array(x_02 ** 3)), axis=1)
PH_ip = np.concatenate((PH_ip, np.array(x_02 ** 5)), axis=1)

# compute the predicted mean and variance
mu_y_0_predicted = np.dot(PH_ip, myu_posterior_theta).flatten()
sigma_y_0_predicted = np.diag(SIGMA_n + SIGMA_n * SIGMA_theta * np.dot(np.dot(PH_ip,np.linalg.inv(SIGMA_n * np.
                      PH_ip.conj().transpose()))
sigma_y_0_predicted = np.reshape(sigma_y_0_predicted, newshape=(sigma_y_0_predicted.shape[0],1)).flatten()

plt.figure(figsize = (10, 7))
plt.autoscale(enable=True, axis='x', tight=True)
plt.autoscale(enable=True, axis='y', tight=True)

plt.plot(x_0, y_0, 'g')
plt.plot(x_02, mu_y_0_predicted, 'kx')
plt.errorbar(x_02, mu_y_0_predicted, sigma_y_0_predicted, fmt='r.', capsize=5)
plt.title("The graph of ture function")
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

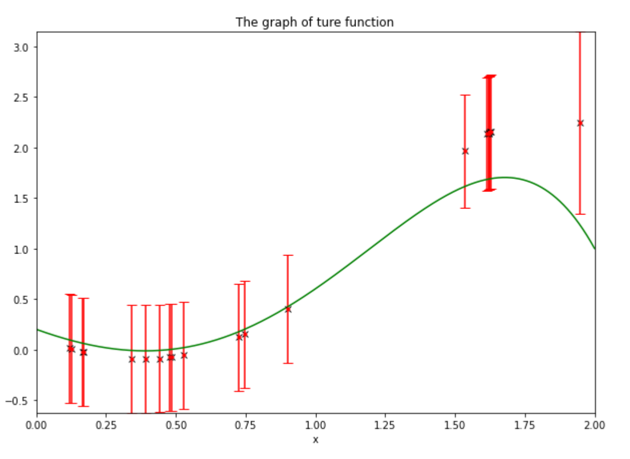


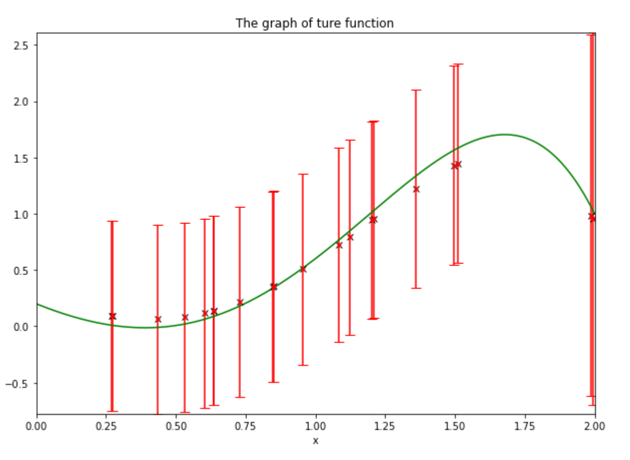
Part ii: Repeating the previous stage with random values

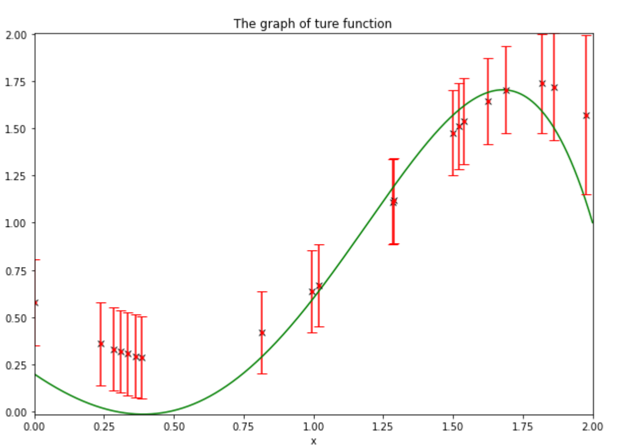
```python
np.random.seed(5)

# true signal curve
x_0 = np.array(list(frange(0, 2, 0.0001)))
x_0 = np.reshape(x_0, newshape=(x_0.shape[0], 1))
y_0 = 0.2 * np.ones(shape=(x_0.shape[0],1)) - x_0 + 0.9 * x_0**2 + 0.7 * x_0**3 - 0.2 * x_0**5

N = 20 # examples of training (20 or 500)

# [a b] sample interval
a = 0
b = 2

# assemble samples
x_01 = np.array(list(frange(a, b, b/N)))
x_01 = np.reshape(x_01, newshape=(x_01.shape[0], 1))

# assembling noise
SIGMA_array = np.array([0.1,0.6,0.5,0.8,0.2,0.3])

for i in range(6):
    n = math.sqrt(SIGMA_array[i]) * np.random.randn(N,1)

    # make use of the true theta
    THETA_tr = np.array([[0.2], [-1], [0.9], [0.7], [-0.2]])

    THETA_dst = np.array([[-0.005], [-10.60], [0.470], [0.097], [-0.083]])
    l = THETA_tr.shape[0]

    # matrix_0 measurement
    PH_i = np.ones(shape=(N, 1))
    PH_i = np.concatenate((PH_i, np.array(x_01)), axis=1)
    PH_i = np.concatenate((PH_i, np.array(x_01**2)), axis=1)
    PH_i = np.concatenate((PH_i, np.array(x_01**3)), axis=1)
    PH_i = np.concatenate((PH_i, np.array(x_01**5)), axis=1)

    # Using the linear model, produce noisy observations
    y_01 = np.dot(PH_i, THETA_tr) + n

    # select the Gaussian prior parameters
    SIGMA_theta = 0.1
    myu_prior_theta = THETA_tr # or myu_prior_theta = THETA_dst;

    # calculate the Gaussian posterior's of the covariance matrix_0
    SIGMA_posterior_theta = np.linalg.inv((SIGMA_theta**-1) * np.eye(l) + (SIGMA_array[i]**-1) * np.dot(PH_i.co
    # calculate the mean of the posterior
    myu_posterior_theta = myu_prior_theta + (SIGMA_array[i]**-1) * np.dot(np.dot(SIGMA_posterior_theta, PH_i.co

    # linear prediction
    L_p = 20

    # create samples for prediction
    x_02 = (b-a) * np.random.rand(L_p, 1)

    # calculate the measurement matrix_0 for prediction
    PH_ip = np.ones(shape=(L_p, 1))
    PH_ip = np.concatenate((PH_ip, np.array(x_02)), axis=1)
    PH_ip = np.concatenate((PH_ip, np.array(x_02 ** 2)), axis=1)
    PH_ip = np.concatenate((PH_ip, np.array(x_02 ** 3)), axis=1)
    PH_ip = np.concatenate((PH_ip, np.array(x_02 ** 5)), axis=1)

    # compute the predicted mean and variance
    mu_y_0_predicted = np.dot(PH_ip, myu_posterior_theta).flatten()
    sigma_y_0_predicted = np.diag(SIGMA_array[i] + SIGMA_array[i] * SIGMA_theta * np.dot(np.dot(PH_ip,np.linalg
                          PH_ip.conj().transpose()))
    sigma_y_0_predicted = np.reshape(sigma_y_0_predicted, newshape=(sigma_y_0_predicted.shape[0],1)).flatten()

    plt.figure(figsize = (10, 7))
    plt.autoscale(enable=True, axis='x', tight=True)
    plt.autoscale(enable=True, axis='y', tight=True)

    plt.plot(x_0, y_0, 'g')
    plt.plot(x_02, mu_y_0_predicted, 'kx')
    plt.errorbar(x_02, mu_y_0_predicted, sigma_y_0_predicted, fmt='r.', capsize=5)
    plt.title("The graph of ture function")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

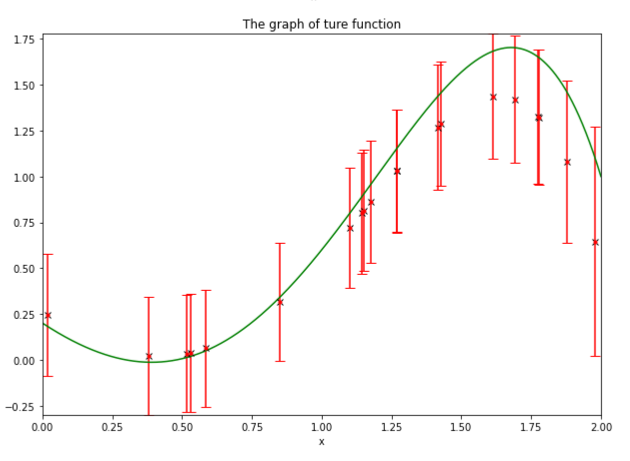













```python
np.random.seed(5)

# true signal curve
x_0 = np.array(list(frange(0, 2, 0.0001)))
x_0 = np.reshape(x_0, newshape=(x_0.shape[0], 1))
y_0 = 0.2 * np.ones(shape=(x_0.shape[0],1)) - x_0 + 0.9 * x_0**2 + 0.7 * x_0**3 - 0.2 * x_0**5

N = 20 # examples of training (20 or 500)

# [a b] sample interval
a = 0
b = 2

# assemble samples
x_01 = np.array(list(frange(a, b, b/N)))
x_01 = np.reshape(x_01, newshape=(x_01.shape[0], 1))
# matrix_0 measurement
PH_i = np.ones(shape=(N, 1))
PH_i = np.concatenate((PH_i, np.array(x_01)), axis=1)
PH_i = np.concatenate((PH_i, np.array(x_01**2)), axis=1)
PH_i = np.concatenate((PH_i, np.array(x_01**3)), axis=1)
PH_i = np.concatenate((PH_i, np.array(x_01**5)), axis=1)

# assembling noise
SIGMA_n = 0.15
n = math.sqrt(SIGMA_n) * np.random.randn(N,1)

# make use of the true theta
THETA_tr = np.array([[0.2], [-1], [0.9], [0.7], [-0.2]])

THETA_dst = np.array([[-0.006], [-10.80], [0.480], [0.098], [-0.085]])
l = THETA_tr.shape[0]
# Using the linear model, produce noisy observations
y_01 = np.dot(PH_i, THETA_tr) + n

# select the Gaussian prior parameters
SIGMA_theta = 2
myu_prior_theta = THETA_tr # or myu_prior_theta = THETA_dst;

# calculate the Gaussian posterior's of the covariance matrix_0
SIGMA_posterior_theta = np.linalg.inv((SIGMA_theta**-1) * np.eye(l) + (SIGMA_n**-1) * np.dot(PH_i.conj().transp
# calculate the mean of the posterior
myu_posterior_theta = myu_prior_theta + (SIGMA_n**-1) * np.dot(np.dot(SIGMA_posterior_theta, PH_i.conj().transp

# linear prediction
L_p = 20

# create samples for prediction
x_02 = (b-a) * np.random.rand(L_p, 1)

# calculate the measurement matrix_0 for prediction
PH_ip = np.ones(shape=(L_p, 1))
PH_ip = np.concatenate((PH_ip, np.array(x_02)), axis=1)
PH_ip = np.concatenate((PH_ip, np.array(x_02 ** 2)), axis=1)
PH_ip = np.concatenate((PH_ip, np.array(x_02 ** 3)), axis=1)
PH_ip = np.concatenate((PH_ip, np.array(x_02 ** 5)), axis=1)

# compute the predicted mean and variance
mu_y_0_predicted = np.dot(PH_ip, myu_posterior_theta).flatten()
sigma_y_0_predicted = np.diag(SIGMA_n + SIGMA_n * SIGMA_theta * np.dot(np.dot(PH_ip,np.linalg.inv(SIGMA_n * np.
                      PH_ip.conj().transpose()))
sigma_y_0_predicted = np.reshape(sigma_y_0_predicted, newshape=(sigma_y_0_predicted.shape[0],1)).flatten()

plt.figure(figsize = (10, 7))
plt.autoscale(enable=True, axis='x', tight=True)
plt.autoscale(enable=True, axis='y', tight=True)

plt.plot(x_0, y_0, 'g')
plt.plot(x_02, mu_y_0_predicted, 'kx')
plt.errorbar(x_02, mu_y_0_predicted, sigma_y_0_predicted, fmt='r.', capsize=5)
plt.title("The graph of ture function")
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Comment

* The figures above show that when the number of training samples is limited, the sigma value will not be particularly accurate, regardless of the value of sigma.

* According to the graph it is clear that, regarding noise variences when they are increased, the error bars become larger.¶

*When the mean of theta prior is picked at random to match the data samples, the curve does not adequately fit the data.

Assignment C: 3.4

```
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)
```

Part i: Create data samples of the signal

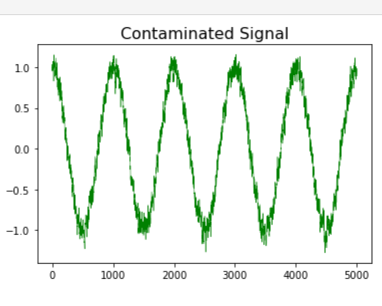
```
np.random.seed(10)
N = 5000
w0 = 2 * 10**(-3) * np.pi
sigma = np.sqrt(0.0025)
noise = sigma * np.random.randn(N,1)
a1 = 0.8
a2 = 0.75
S = np.cos(w0 * np.arange(N)).reshape(-1,1)
```

Part ii: Creating data samples of the AR process v1(n) = a1*v1(n-1) + Eta_n

```
v1 = np.zeros((N,1)) #initilalize AR procedures
v2 = np.zeros((N,1))
for i in range(1,N):
    v1[i] = a1 * v1[i-1] + noise[i]
```

Part iii: Generating Contaminated signal

```
d = S + v1
x_axis = np.arange(0,N,1)
plt.figure(1)
plt.plot(x_axis,S+v1, 'g',lw=0.5)
plt.title(r'Contaminated Signal', fontsize=16)
plt.show()
```

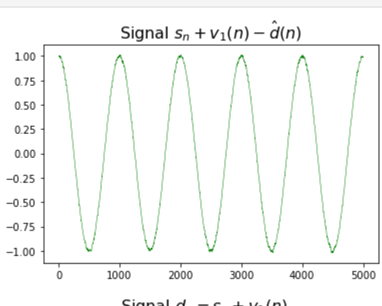


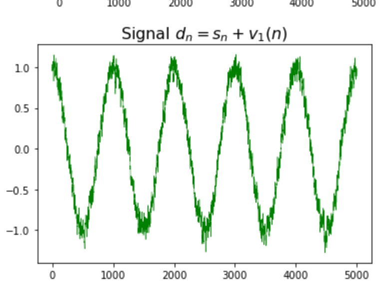
Part iv: Creating data samples of the AR process v1(n) = a2*v2(n-1) + Eta_n

```
for i in range(1,N):
     v2[i] = a2 * v2[i-1] + noise[i]
```

Part v: Creating the sequence of the restored signal

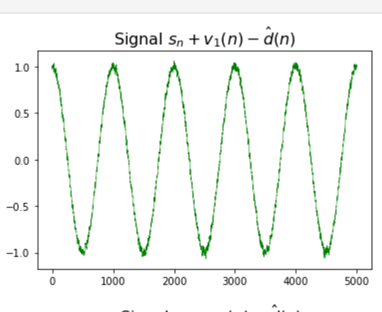
```
a=v2.ravel()
X = np.array([np.roll(a,1)]).T
a = np.array([a]).T
X = np.concatenate((a,X),axis=1)
w1 = np.linalg.inv(np.dot(X.T,X)).dot(X.T).dot(d)
d_est = X.dot(w1)
# list_A = []
# var = np.array([0,1,2,3])
# var1 = np.hstack((var[-2:],var))
# var2 = var1[3-2:3]
# var3 = list_A.append(var2[::-1])
# var2 = var1[4-2:4]
# var3 = list_A.append(var2[::-1])
# var2 = var1[5-2:5]
# var3 = list_A.append(var2[::-1])
# np.asarray(list_A).squeeze()

x_axis = np.arange(0,N,1)
plt.figure(1)
plt.plot(x_axis,S+v1-d_est, 'g',lw=0.5)
plt.title(r'Signal $s_n+v_1(n)-\hat d(n)$', fontsize=16)

plt.figure(2)
plt.plot(x_axis,S+v1, 'g',lw=0.5)
plt.title(r'Signal $d_n=s_n+v_1(n)$', fontsize=16)
plt.show()
plt.show()
```

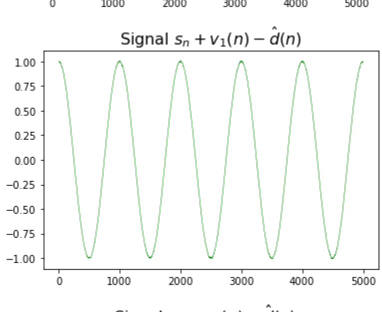


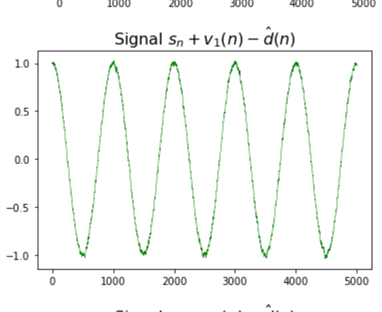


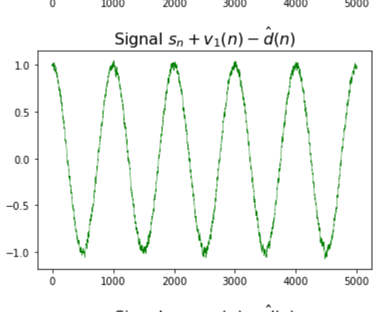
Part vi: Repeating the step ii-v using a_2 value

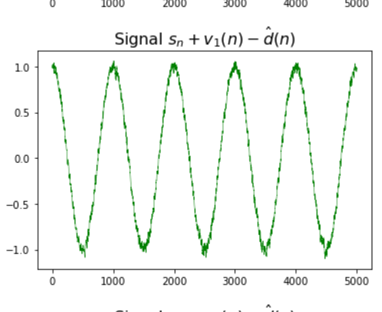
```
a1 = 0.8
a2 = np.array([0.9,0.8,0.7,0.6,0.5,0.3])
for i in range (len(a2)):
    for j in range(1,N):
        v2[j] = a2[i] * v2[j-1] + noise[j]
    a=v2.ravel()
    X = np.array([np.roll(a,1)]).T
    a = np.array([a]).T
    X = np.concatenate((a,X),axis=1)
    w1 = np.linalg.inv(np.dot(X.T,X)).dot(X.T).dot(d)
    d_est = X.dot(w1)
    x_axis = np.arange(0,N,1)
    plt.figure()
    plt.plot(x_axis,S+v1-d_est, 'g',lw=0.5)
    plt.title(r'Signal $s_n+v_1(n)-\hat d(n)$', fontsize=16)

    plt.show()
    # Optimum value of a2 is between 0.8 and 0.7
```

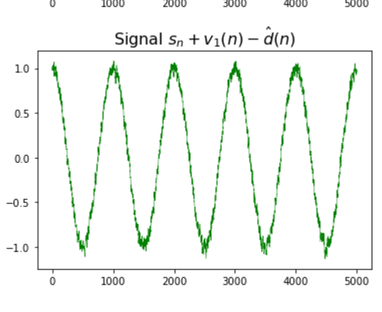








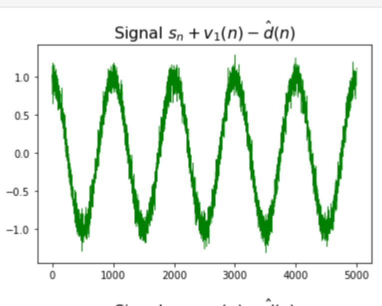


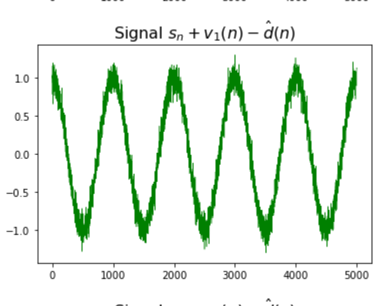


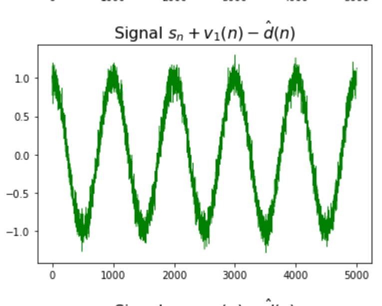
According to the graph, As the value of a_2 is decreasing, the noise is not cancelling properly

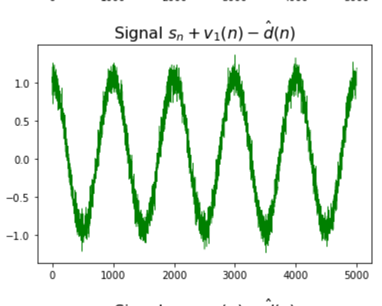
Part vii: Repeating the step ii-v using (Sigma)^2 and a_2 values

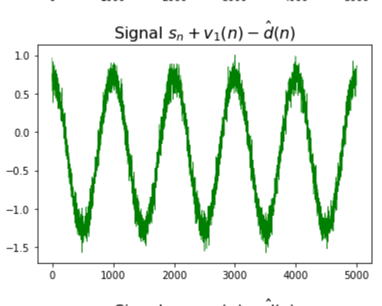
```
sigma = np.sqrt(np.array([0.01,0.05,0.1,0.2,0.5]))
a1 = 0.8
a2 = np.array([0.9,0.8,0.7,0.6,0.5,0.3])
for k in range (len(sigma)):
    noise = sigma[k] * np.random.randn(N,1)
    for j in range (len(a2)):
        for i in range(1,N):
            v2[j] = a2[j] * v2[j-1] + noise[i]
            v1[i] = a1 * v1[j-1] + noise[i]
        d = S + v1
        a=v2.ravel()
        X = np.array([np.roll(a,1)]).T
        a = np.array([a]).T
        X = np.concatenate((a,X),axis=1)
        w1 = np.linalg.inv(np.dot(X.T,X)).dot(X.T).dot(d)
        d_est = X.dot(w1)
        S_hat_n=d-d_est
        x_axis = np.arange(0,N,1)
        plt.figure()
        #ax[k,j].plot(x_axis,d_n-d_est, 'r',lw=0.5)
        #ax[k,j].plot([],label="μ = {:3.2f}\nσ = {:3.2f}".format(sigma[k],a2[j] ), alpha=0)
        #ax[k,j].title(r'Signal $s_n+v_1(n)-\hat d(n)$', fontsize=16)
        plt.plot(x_axis, S_hat_n, 'g',lw=0.5)
        plt.plot([],label="μ = {:3.2f}\nσ = {:3.2f}".format(sigma[k],a2[j] ), alpha=0)
        plt.title(r'Signal $s_n+v_1(n)-\hat d(n)$', fontsize=16)

plt.show()
```

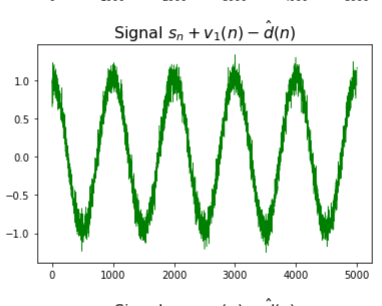


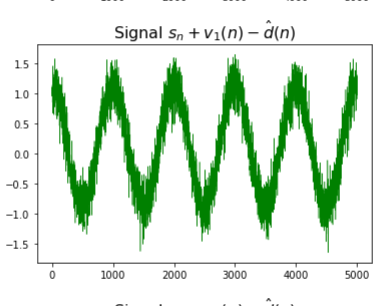


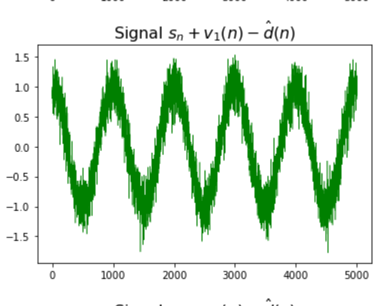


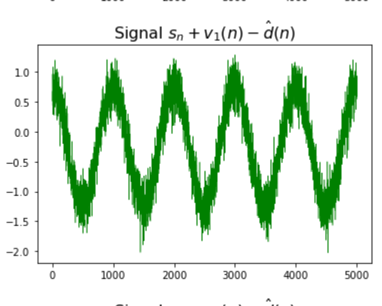


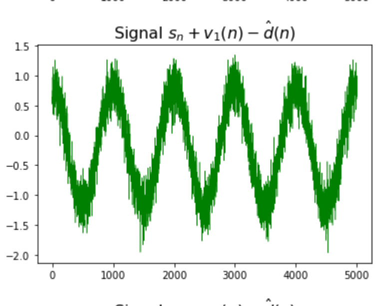


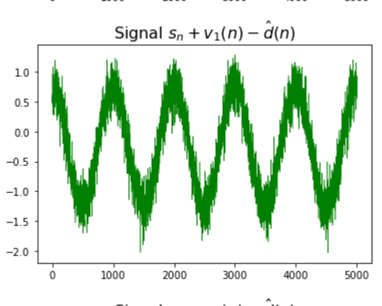


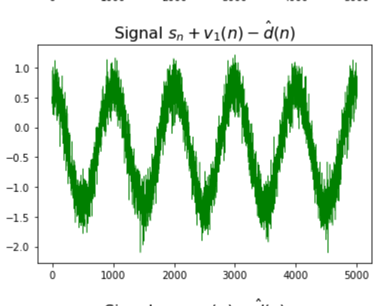


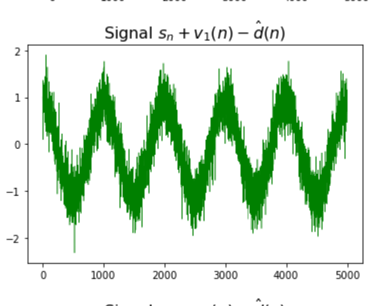


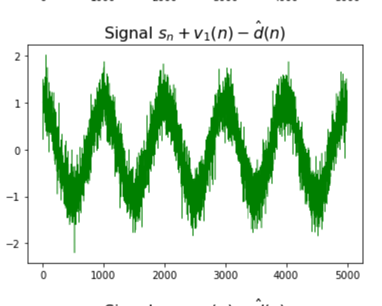


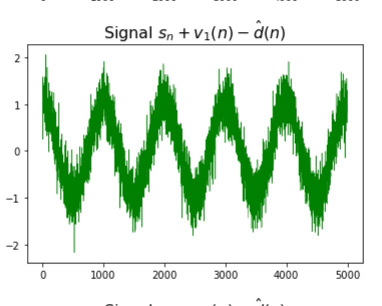


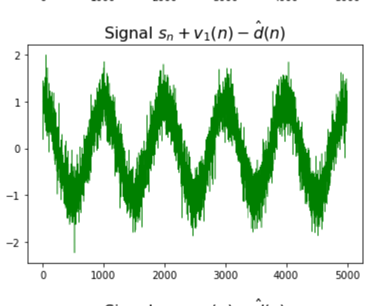


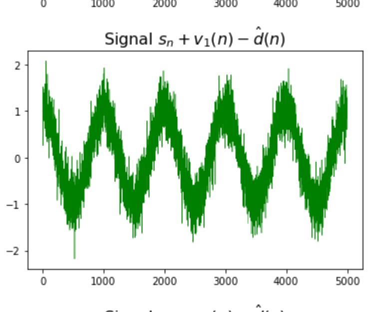


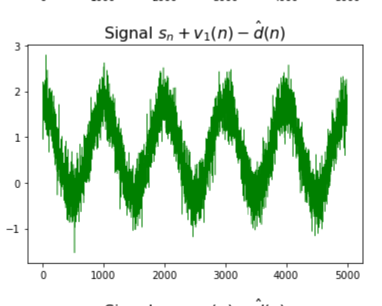


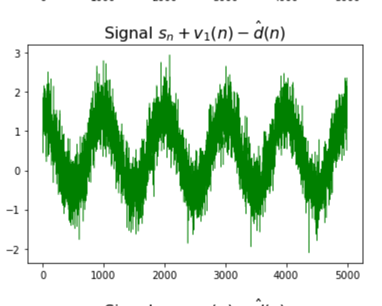


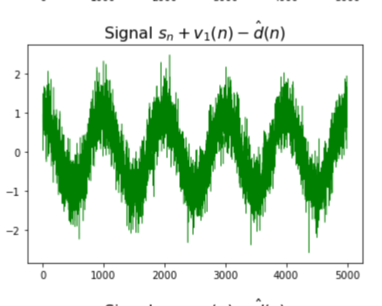


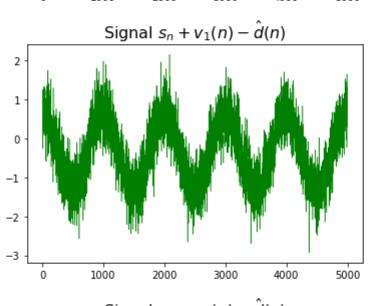


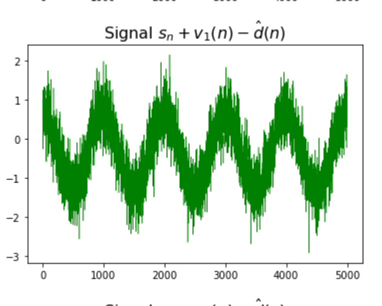


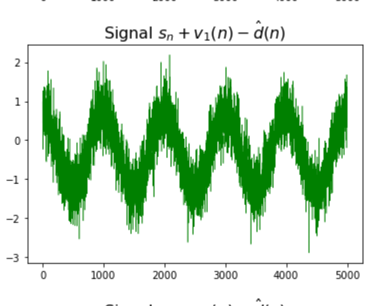


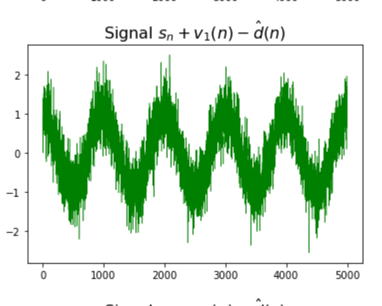


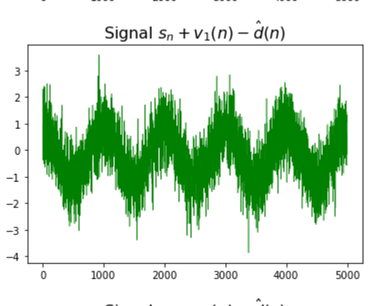


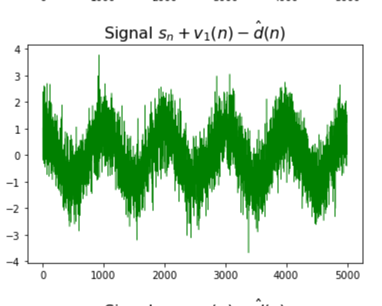


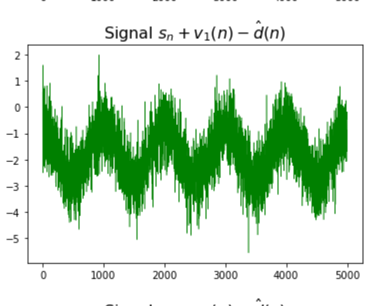


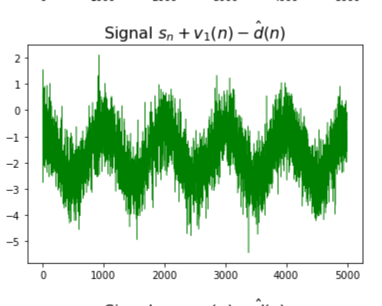


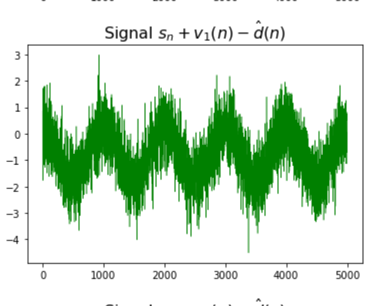


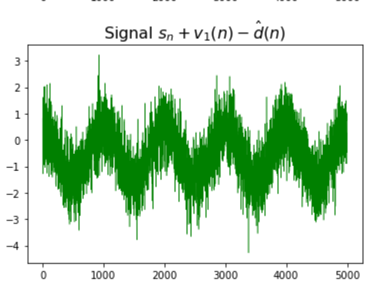












When shown in the graphs, as the variance increases, the noise in the signal increases, making it more difficult for the algorithm to solve.