⇒ at the posterior loss expectation for adopting action g. given x, is

$$\rho(\hat{9}|\mathbf{z}) = E_{P}(\hat{9}|\mathbf{z})[L(\hat{9},\hat{9})]$$

$$= P_{0} \cdot L(\hat{9},0) + P_{1} \cdot L(\hat{9},1)$$

$$= P_{0} \cdot L(\hat{9},0) + (1-P_{0}) \cdot L(\hat{9},1)$$

$$= P_{0} \cdot L(\hat{9},0) + P_{0} \cdot L(\hat{9},0) - L(\hat{9},1)$$

$$= L(\hat{9},1) + P_{0} \cdot L(\hat{9},0) - L(\hat{9},1)$$

Therefore,

Because P(0/2) and P(1/2) are both linear function of Po, the unique optimal threshold is

for example, Po = Soi+Sio

> b) We need to derive the loss matrix where the threshold is 0.1.

We can execute then,

$$P_1 = \frac{310}{301 + 310} = \frac{1}{9+1} = 01$$

Predicted	true labely
Predicted block 9	0 1
0	110
1	110

と(では) これ(川文)、単二二二

calculate the Gaussian posterior's of the covariance matrix 0 $SIGMA_posterior_theta = np.linalg.inv((SIGMA_theta**-1) * np.eye(l) + (SIGMA_n**-1) * np.dot(PH_i.conj().transg(line)) + (SIGMA_n**-1) * np.dot() + (SIGMA_n**-1) * np.do$ # calculate the mean of the posterior $myu_posterior_theta = myu_prior_theta + (SIGMA_n**-1) * np.dot(np.dot(SIGMA posterior theta, PH i.conj().transgerians + (SIGMA_n**-1) * np.dot(np.dot(SIGMA posterior theta, PH i.conj()).transgerians + (SIGMA_n**-1) * np.dot(np.dot()).transgerians + (SIGMA_n**-1) * np.dot() * np.$ # 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() The graph of ture function 1.6 1.4 1.2 1.0 0.8 0.6 0.4 0.0 0.25 0.00 0.75 1.00 1.25 1.50 1.75 2.00 Part ii: Repeating the previous stage with random values np.random.seed(5) # true signal curve x = 0 = np.array(list(frange(0, 2, 0.0001))) $x_0 = \text{np.reshape}(x_0, \text{newshape}(x_0.\text{shape}[0], 1))$ y = 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**5N = 20 # examples of training (20 or 500)# [a b] sample interval a = 0b = 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 = 0.1myu 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.c # 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.linalq 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() The graph of ture function 1.75 1.50 1.25 1.00 0.75 0.50 0.25 0.00 1.75 0.25 0.75 1.00 1.25 1.50 0.00 0.50 2.00 The graph of ture function 2.5 2.0 1.5 1.0 0.5 0.0 -0.51.25 1.50 1.75 0.25 0.50 0.75 1.00 2.00 0.00 The graph of ture function 3.0 2.5 2.0 1.5 1.0 0.5 0.0 -0.50.25 0.75 1.00 1.25 1.50 1.75 0.00 0.50 2.00 The graph of ture function 2.5 2.0 1.5 0.5 -0.50.25 1.25 1.50 1.75 0.50 0.75 1.00 2.00 0.00 The graph of ture function 2.00 1.75 1.50 1.25 > 1.00 0.75 0.50 0.25 0.00 1.75 0.25 0.50 0.75 1.00 1.25 1.50 2.00 The graph of ture function 1.75 1.50 1.25 1.00 0.75 0.50 0.25 0.00 -0.250.50 0.75 1.25 0.25 1.00 1.50 1.75 2.00 0.00 # 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 = 0b = 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]]) 1 = 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 $SIGMA_posterior_theta = np.linalg.inv((SIGMA_theta**-1) * np.eye(l) + (SIGMA_n**-1) * np.dot(PH_i.conj().transgraphically for the conj().transgraphically for the conj() and the conj() are conj().transgraphically for the conj() are conj().transgraphically for the conj() are conj().transgraphically for the conj() are conj() are conj() are conj() are conj().transgraphically for the conj() are c$ # calculate the mean of the posterior # 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() The graph of ture function 1.75 1.50 1.25 1.00

0.75

0.50

0.25

0.00

0.00

Comment

0.25

the error bars become larger.¶

does not adequately fit the data.

0.75

1.00

value will not be particularly accurate, regardless of the value of sigma.

1.25

* The figures above show that when the number of training samples is limited, the sigma

* According to the graph it is clear that, regarding noise variences when they are increased,

*When the mean of theta prior is picked at random to match the data samples, the curve

1.50

1.75

2.00

Assignment C: 3.3

import matplotlib.pyplot as plt

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

 $y_0 = 0.2 * np.ones(shape=(x_0.shape[0],1)) - x_0 + 0.9 * x_0**2 + 0.7 * x_0**3 - 0.2 * x_0**5$

def frange(x_0, y_0, jump):
 while x_0 < y_0:
 yield x_0
 x 0 += jump</pre>

Gaussian distribution

x = 0 = np.array(list(frange(0, 2, 0.0001)))

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

x 01 = np.array(list(frange(a, b, b/N)))

x = 0 = np.reshape(x = 0, newshape(x = 0.shape[0], 1))

x 01 = np.reshape(x 01, newshape=(x 01.shape[0], 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)

THETA_tr = np.array([[0.2], [-1], [0.9], [0.7], [-0.2]])

myu_prior_theta = THETA_tr # or myu_prior_theta = THETA_dst;

Using the linear model, produce noisy observations

THETA dst = np.array([[-0.005], [-10.60], [0.470], [0.097], [-0.083]])

n = math.sqrt(SIGMA_n) * np.random.randn(N,1)

np.random.seed(5)

true signal curve

[a b] sample interval

assemble samples

assembling noise
SIGMA n = 0.05

1 = THETA tr.shape[0]

SIGMA theta = 0.1

matrix 0 measurement

PH i = np.ones(shape=(N, 1))

make use of the true theta

 $y_01 = np.dot(PH_i, THETA_tr) + n$

select the Gaussian prior parameters

a = 0b = 2

import numpy as np

import math

In [19]:

In [2]:	<pre>part np.r N = w0 = sigm</pre>	= 2 * 10**(-3) * np.pi na = np.sqrt(0.0025)
In [3]:	a1 = a2 = S = Part	<pre>te = sigma * np.random.randn(N,1) to 0.8 to 0.75 np.cos(w0 * np.arange(N)).reshape(-1,1) to ii: Creating data samples of the AR process v1(n) = a1*v1(n-1) + Eta_n to np.zeros((N,1)) #initilalize AR procedures to np.zeros((N,1))</pre>
In [4]:	Part	<pre>in range(1,N): v1[i] = a1 * v1[i-1] + noise[i] tiii: Generating Contaminated signal s + v1 tis = np.arange(0,N,1) figure(1)</pre>
	plt.	plot(x_axis,S+v1, 'g',lw=0.5) title(r'Contaminated Signal', fontsize=16) show() Contaminated Signal
	0.0 - -0.5 - -1.0 -	0 1000 2000 3000 4000 5000
In [5]:	_{for}	tiv: Creating data samples of the AR process v1(n) = a2*v2(n-1) + Eta_n i in range(1,N): v2[i] = a2 * v2[i-1] + noise[i] tv: Creating the sequence of the restored signal
	a = X = w1 = d_es # 1i # va # va # va # va	<pre>np.array([np.roll(a,1)]).T np.array([a]).T np.concatenate((a,X),axis=1) np.linalg.inv(np.dot(X.T,X)).dot(X.T).dot(d) nt = X.dot(w1) nst_A = [] nr = np.array([0,1,2,3]) nr1 = np.hstack((var[-2:],var)) nr2 = var1[3-2:3] nr3 = list_A.append(var2[::-1]) nr2 = var1[4-2:4]</pre>
	<pre># va # va # np x_ax plt. plt. plt.</pre>	<pre>ir3 = list_A.append(var2[::-1]) ir2 = var1[5-2:5] ir3 = list_A.append(var2[::-1]) i.asarray(list_A).squeeze() iis = np.arange(0,N,1) figure(1) plot(x_axis,S+v1-d_est, 'g',lw=0.5) title(r'Signal \$s_n+v_1(n)-\hat d(n)\$', fontsize=16) figure(2) plot(y_avis_S+v1-d_est)</pre>
	plt. plt.	plot(x_axis , $S+v1$, 'g', $1w=0.5$) title($r'Signal $d_n=s_n+v_1(n)$'$, fontsize=16) show() show() Signal $s_n + v_1(n) - \hat{d}(n)$
	0.25 0.00 -0.25 -0.50 -0.75 -1.00	0 1000 2000 3000 4000 5000
	1.0 - 0.5 - 0.0 -	Signal $d_n = s_n + v_1(n)$
In [7]:	a1 =	vi: Repeating the step ii-v using a_2 value 0.8 inp.array([0.9,0.8,0.7,0.6,0.5,0.3])
		<pre>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)</pre>
		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 Signal $s_n+v_1(n)-\hat{d}(n)$
	0.0 - -0.5 - -1.0 -	0 1000 2000 3000 4000 5000
	1.00 0.75 0.50 0.25 0.00 -0.25	Signal $s_n + v_1(n) - \hat{d}(n)$
	-0.75 -1.00	$\frac{1}{0} \frac{1000}{1000} \frac{2000}{3000} \frac{3000}{4000} \frac{4000}{5000}$ Signal $s_n + v_1(n) - \hat{d}(n)$
	0.0 - -0.5 - -1.0 -	0 1000 2000 3000 4000 5000
	0.5 - 0.0 -	Signal $s_n + v_1(n) - \hat{d}(n)$
	-1.0 { 1.0 { 0.5 {	Signal $s_n + v_1(n) - \hat{d}(n)$
	0.0 - -0.5 - -1.0 -	0 1000 2000 3000 4000 5000
	0.5 - 0.0 -	Signal $s_n + v_1(n) - \hat{d}(n)$
		ording to the graph, As the value of a_2 is decreasing, the noise is not celling properly t vii: Repeating the step ii-v using (Sigma)^2 and a_2 values
In [8]:	sigm a1 = a2 = for	<pre>ma = np.sqrt(np.array([0.01,0.05,0.1,0.2,0.5])) s 0.8 s np.array([0.9,0.8,0.7,0.6,0.5,0.3]) 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</pre>
		<pre>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="\mu = {:3.2f}\n\sigma = {:3.2f}\".format(sigma[k],a2[j]), alpha=0)</pre>
	plt.	
	0.5 - 0.0 - -0.5 -	0 1000 2000 3000 4000 5000
	1.0 - 0.5 - 0.0 -	Signal $s_n + v_1(n) - \hat{d}(n)$
	-0.5 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	0.5 - 0.0 - -0.5 -	
	1.0 - 0.5 -	$\frac{1}{2}$ Signal $s_n + v_1(n) - \hat{d}(n)$
	-0.5 - -1.0 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	0.5 - 0.0 - -0.5 - -1.0 -	
	1.0 - 0.5 - 0.0 -	0 1000 2000 3000 4000 5000 Signal $s_n + v_1(n) - \hat{d}(n)$
	-0.5 - -1.0 -	0 1000 2000 3000 4000 5000 Signal $s_n + v_1(n) - \hat{d}(n)$
	0.5 - 0.0 - -0.5 - -1.0 -	0 1000 2000 3000 4000 5000
	1.5 - 1.0 - 0.5 - 0.0 -	Signal $s_n + v_1(n) - \hat{d}(n)$
	-1.0 - -1.5 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	0.0 - -0.5 - -1.0 - -1.5 - -2.0 -	0 1000 2000 3000 4000 5000
	1.5 - 1.0 - 0.5 0.5 1.0	Signal $s_n + v_1(n) - \hat{d}(n)$
	-1.5 - -2.0 - 1.0 - 0.5 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	0.0 - -0.5 - -1.0 - -1.5 - -2.0 -	0 1000 2000 3000 4000 5000 Signal $s_n + v_1(n) - \hat{d}(n)$
	1.0 - 0.5 - 0.0 - -0.5 - -1.0 -	Signal $S_n + V_1(n) - d(n)$
	2 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	0 -	$\frac{1}{0}$ $\frac{1}{1000}$ $\frac{2}{000}$ $\frac{3}{000}$ $\frac{4}{000}$ $\frac{5}{000}$ Signal $s_n + v_1(n) - \hat{d}(n)$
	2 - 1 - 0 1 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2	
	2 - 1 - 0 -	$\begin{array}{c ccccccccccccccccccccccccccccccccccc$
	-1 · -2 ·	$\frac{1}{1000}$ $\frac{1}{2000}$ $\frac{3}{3000}$ $\frac{4}{3000}$ $\frac{5}{3000}$ Signal $\frac{1}{3000}$ $\frac{1}{3000$
	1 - 012 -	
		Signal $s_n + v_1(n) - \hat{d}(n)$
	-1 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	2 - 1 - 0 - -1 -	
	3 - 2 - 1 - 0 -	Signal $s_n + v_1(n) - \hat{d}(n)$
	-1 - -2 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	1 - 0 - -1 - -2 -	0 1000 2000 3000 4000 5000
	2 - 1 - 01 -	$\frac{1}{2}$ $\frac{1}$
	-2 - -3 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	1 - 0 1 2 3	0 1000 2000 3000 4000 5000
	2 - 1 - 0 -	Signal $s_n + v_1(n) - \hat{d}(n)$
	-2 - -3 - 2 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	1 - 0 - -1 - -2 -	0 1000 2000 3000 4000 5000
	3 - 2 - 1 - 0 -	Signal $s_n + v_1(n) - \hat{d}(n)$
	-2 - -3 - -4 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	1 - 0 1 2 3 4 4	0 1000 2000 3000 4000 5000
	2 - 1 - 0 - -1 - -2 - -3 -	Signal $s_n + v_1(n) - \hat{d}(n)$
	-4 - -5 -	0 1000 2000 3000 4000 5000 Signal $s_n + v_1(n) - \hat{d}(n)$
	0 - -1 - -2 - -3 - -4 - -5 -	0 1000 2000 3000 4000 5000
	3 - 2 - 1 - 0 - -1 - -2 -	Signal $s_n + v_1(n) - \hat{d}(n)$
	-3 - -4 - 3 - 2 -	$0 1000 2000 3000 4000 5000$ Signal $s_n + v_1(n) - \hat{d}(n)$
	1 - 0 1 2 3 4 1	

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

Assignment C: 3.4

import matplotlib.pyplot as plt

import numpy as np