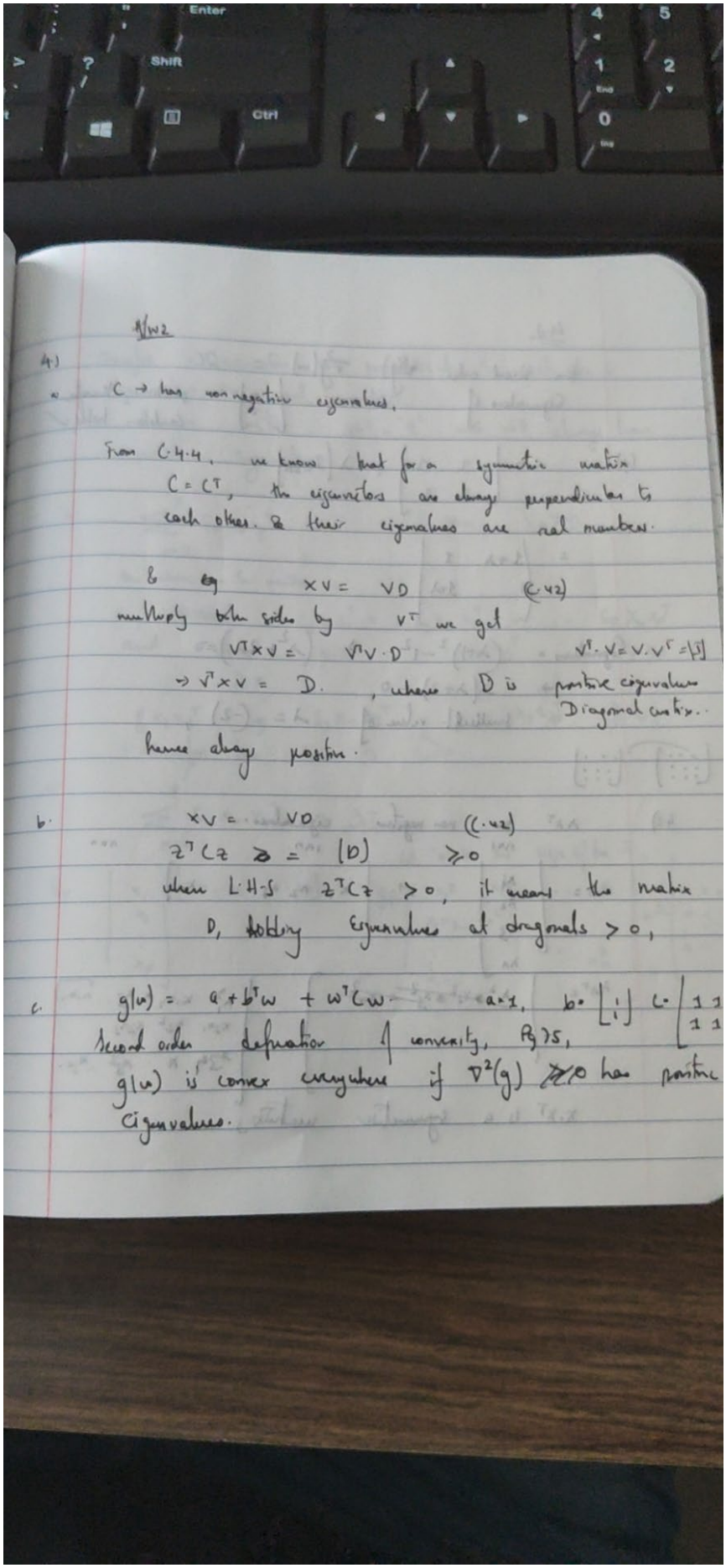


Question 4.1

HW2

4.)

a. C → has non negative eigenvalues.

From C.4.4. we know that for a symmetric matrix $C = C^T$, the eigenvectors are always perpendicular to each other. & their eigenvalues are real numbers.

& eg $XV = VD$ (C.42)

multiply both sides by V^T we get

$V^T X V = V^T V \cdot D$ $\quad V^T \cdot V = V \cdot V^T = [I]$

$\Rightarrow V^T X V = D$. , where D is positive eigenvalues Diagonal matrix..

hence always positive.

b. $XV = VD$ (C.42)

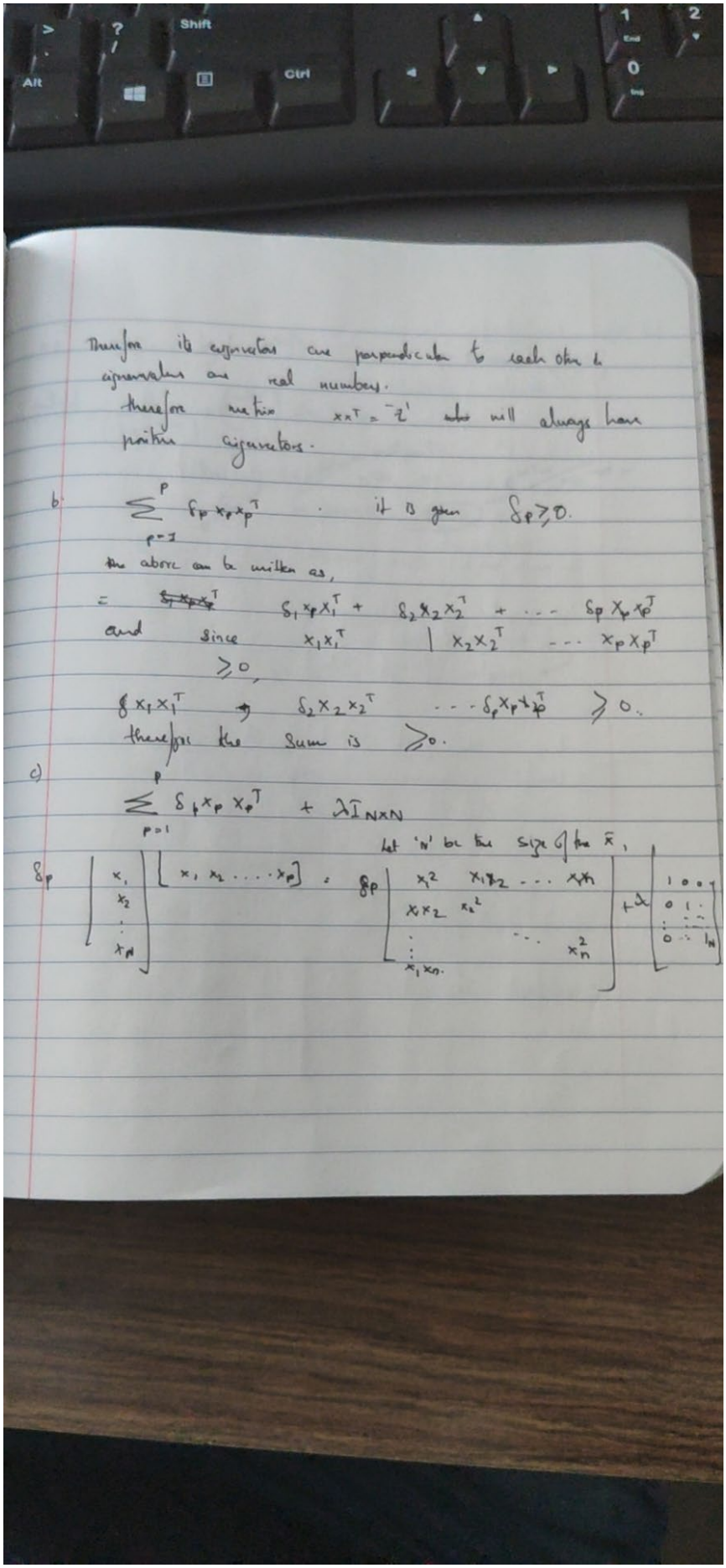
$z^T C z \geq = (D) \geq 0$

when L.H.S $z^T C z > 0$, it means the matrix D, holding Eigenvalues at diagonals > 0,

c. $g(w) = a + b^T w + w^T C w$. $a = 1$, $b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ $C = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

Second order definition ↑ convexity, Pg 75,

$g(w)$ is convex everywhere if $\nabla^2(g) \geq 0$ has positive Eigenvalues.



Therefore its eigenvectors are perpendicular to each other & eigenvalues are real numbers.

therefore matrix $xx^T = \bar{z}$ also will always have positive eigenvectors.

b) $\sum_{p=1}^{P} \delta_p x_p x_p^T$. if it is given $\delta_p \geq 0$.

the above can be written as,

$= \delta_1 x_p x_1^T + \delta_2 x_2 x_2^T + \ldots \delta_p x_p x_p^T$

and since $x_1 x_1^T \mid x_2 x_2^T \ldots x_p x_p^T \geq 0$,

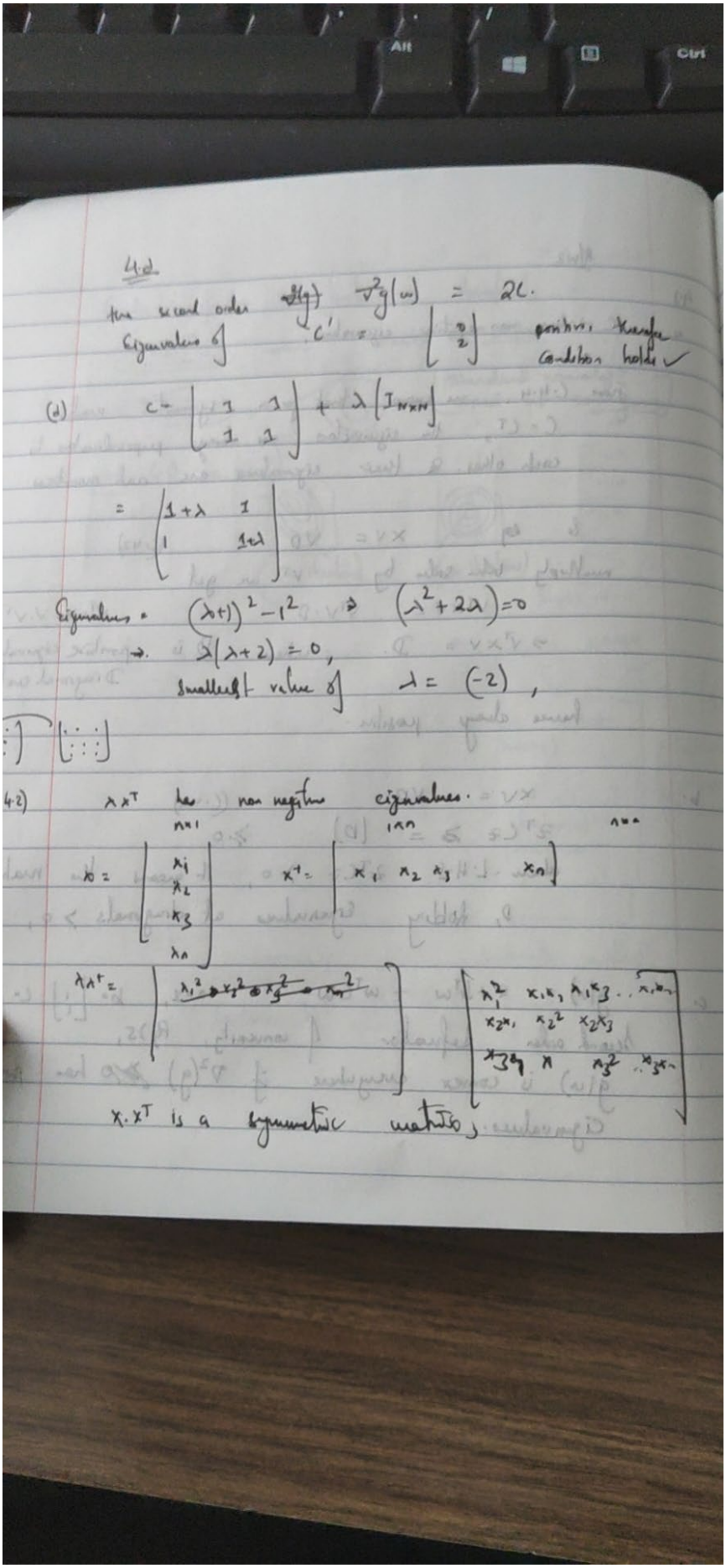
$\delta x_1 x_1^T + \delta_2 x_2 x_2^T \ldots \delta_p x_p x_p^T \geq 0$.

therefore the sum is ≥ 0.

c) $\sum_{p=1}^{P} \delta_p x_p x_p^T + \lambda I_{N \times N}$

Let 'N' be the size of the $\bar{x}$,

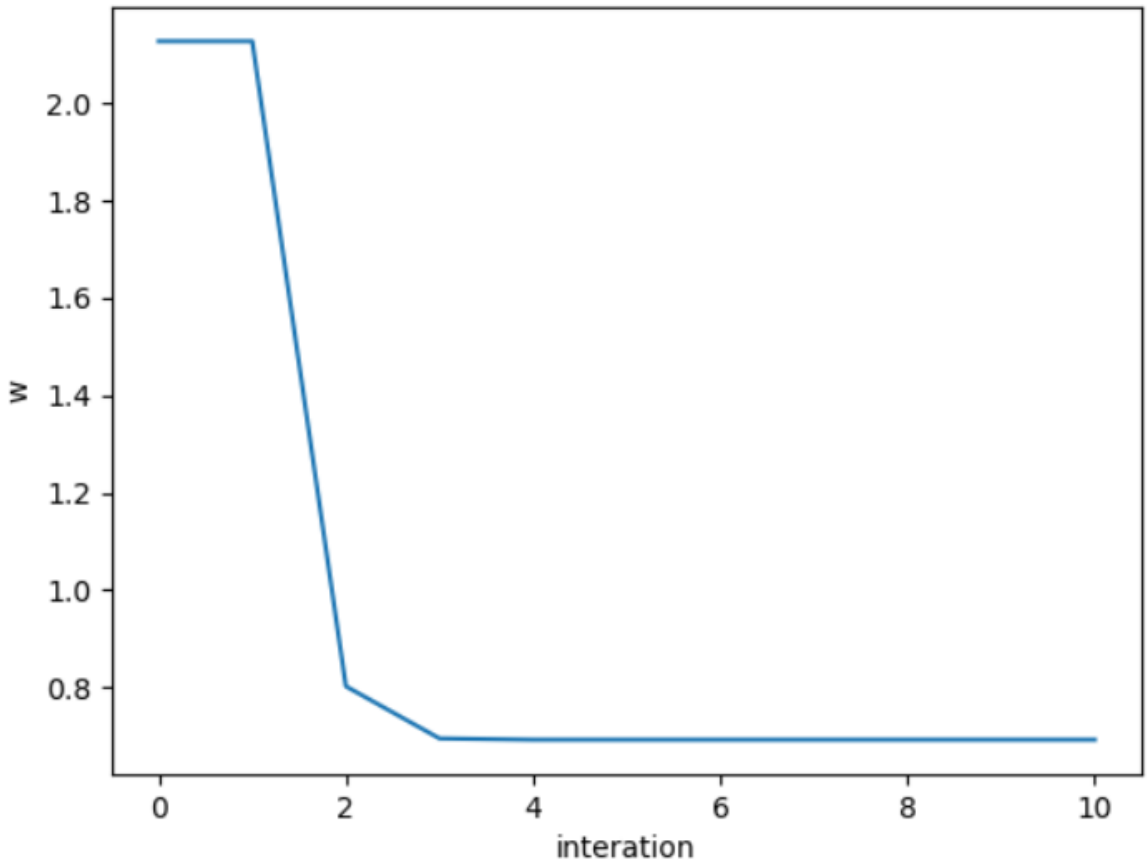
$\delta_p \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \ldots & x_p \end{bmatrix} = \delta_p \begin{bmatrix} x_1^2 & x_1 x_2 & \ldots & x_1 x_n \\ x_1 x_2 & x_2^2 & & \\ \vdots & & \ddots & x_n^2 \\ x_1 x_n & & & \end{bmatrix} + \lambda \begin{bmatrix} 1 & 0 & \ldots \\ 0 & 1 & \\ \vdots & & \ddots \\ 0 & \ldots & I_N \end{bmatrix}$


4.2.
the second order $\nabla^2 g(w) = 2C$.
Eigenvalues of $C = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$ positive definite condition holds ✓
(d) $C = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \lambda \left[I_{n \times n} \right]$
$= \begin{vmatrix} 1+\lambda & 1 \\ 1 & 1+\lambda \end{vmatrix}$
Eigenvalues $(\lambda+1)^2 - 1^2 \Rightarrow (\lambda^2 + 2\lambda) = 0$
$\Rightarrow \lambda(\lambda+2) = 0$,
Smallest value of $\lambda = (-2)$,
4.2) xx^T has non negative eigenvalues.
$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_n \end{bmatrix}$ $x^T = \begin{bmatrix} x_1 & x_2 & x_3 & x_n \end{bmatrix}$
$xx^T = \begin{bmatrix} x_1^2 & x_1x_2 & x_1x_3 & x_1x_n \\ x_2x_1 & x_2^2 & x_2x_3 & \\ x_3x_1 & x & x_3^2 & x_3x_n \end{bmatrix}$
$x \cdot x^T$ is a symmetric matrix,

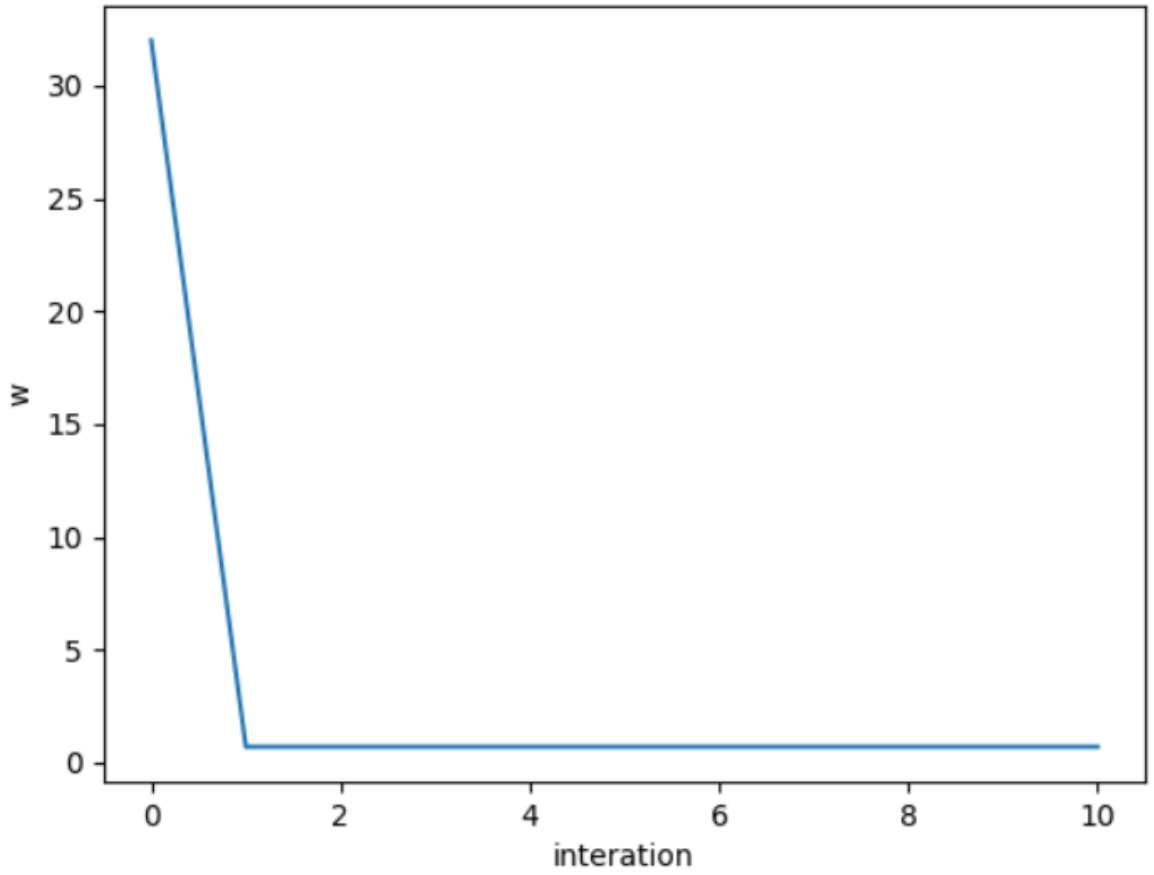
```
# using an automatic differentiator - like the one imported via the statement
below - makes coding up gradient descent a breeze
import autograd.numpy as np
from autograd import grad
from autograd import hessian
# from matplotlib import rcParams
# rcParams['figure.autolayout'] = True
import matplotlib.pyplot as plt
import math
import sympy as sy


# newtons method function - inputs: g (input function), max_its (maximum number of
iterations), w (initialization)
def newtons_method(g, max_its, w, **kwargs):
    # compute gradient module using autograd
    gradient = grad(g)
    hess = hessian(g)

    # set numericxal stability parameter / regularization parameter
    epsilon = 10 ** (-7)
    if 'epsilon' in kwargs:
        beta = kwargs['epsilon']

    # run the newtons method loop
    weight_history = [w]  # container for weight history
    cost_history = [g(w)]  # container for corresponding cost function history
    for k in range(max_its):
        # evaluate the gradient and hessian
        grad_eval = gradient(w)
        hess_eval = hess(w)

        # reshape hessian to square matrix for numpy linalg functionality
        hess_eval.shape = (int((np.size(hess_eval)) ** (0.5)),
int((np.size(hess_eval)) ** (0.5)))

        # solve second order system system for weight update
        A = hess_eval + epsilon * np.eye(w.size)
        b = grad_eval
        w = np.linalg.solve(A, np.dot(A, w) - b)

        # record weight and cost
        weight_history.append(w)
        cost_history.append(g(w))
    return weight_history, cost_history


w1 = np.array([[1], [1]])


def g(w):
    x = np.dot(np.transpose(w), w)
    return np.log(1 + np.exp(x))


max1 = 10
newtons_method(g, max1, w1)
```

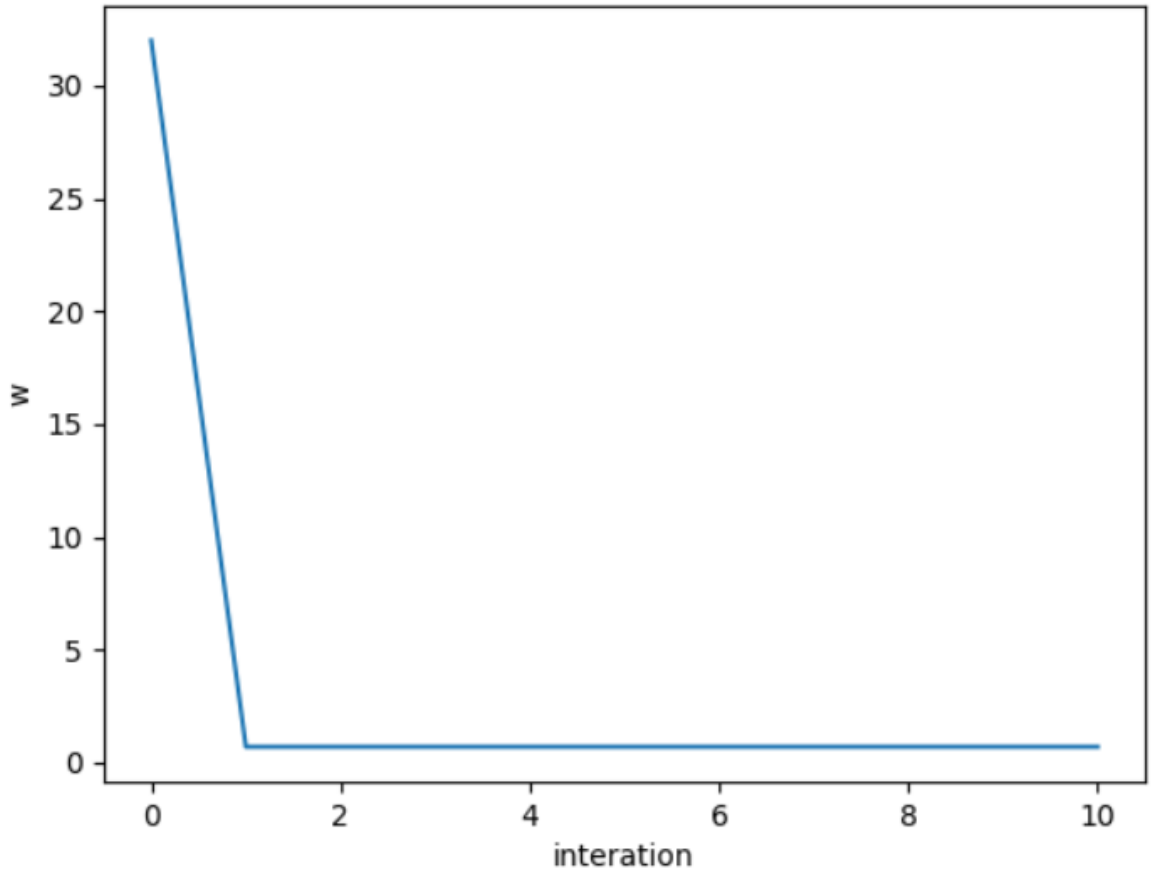
```
weights, costliest = newtons_method(g, max1, w1)
plot_list = []
for item in costliest:
    plot_list.append(item[0,0])
plt.plot(plot_list)
plt.xlabel("interation")
plt.ylabel("w")
plt.show()
```

c. Taking initial point as [1,1]



d. Taking initial point as [4,4]

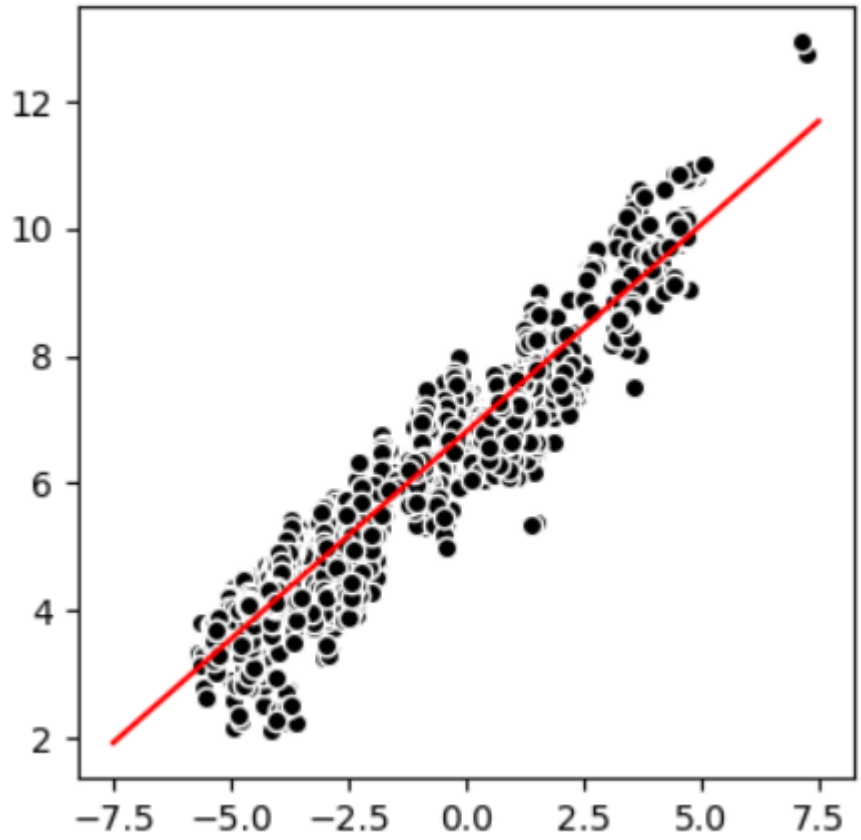




5.2

```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

csvname = 'kleibers_law_data.csv'
data = np.loadtxt(csvname,delimiter = ',')
fig, ax = plt.subplots(1, 1, figsize=(4,4))
ax.scatter(np.log(data[0,:]),np.log(data[1,:]),color = 'k',edgecolor = 'w')
plt.show()
x = np.log(data[:-1,:])
y = np.log(data[-1,:])
def model(x,w):
    a = w[0] +np.dot(x.T,w[1:])
    return a.T
def least_squares(w):
    cost = np.sum((model(x,w)-y)**2)
    return cost/float(y.size)
w = np.asarray([1.5,1.5])
w1=least_squares(w)
print(w1)
def gradient_descent(g,alpha,max_its,w):
     gradient = grad(g)
     weight_history=[]
     best_w = w
     best_eval = g(w)
     for k in range(max_its):
```

```
        grad_eval = gradient(w)
        weight_history.append(w)
        w = w - alpha*grad_eval
        test_eval = g(w)
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
    return best_w,weight_history
best_w1,weight_history1 = gradient_descent(g = least_squares,alpha = 10**-
0,max_its = 500,w = w)
best_w2,weight_history2 = gradient_descent(g = least_squares,alpha = 10**-
1,max_its = 500,w = w)
best_w3,weight_history3 = gradient_descent(g = least_squares,alpha = 10**-
2,max_its = 500,w = w)
def MSE(weight_history,g):
 # loop over weight history and compute the MSE at each step o gradient descent
 costfunchistory=[g(w) for w in weight_history]
 # plot cost function history
 plt.figure()
 plt.plot([i for i in range(500)], costfunchistory)
 plt.show()
MSE(weight_history1, g = least_squares)
MSE(weight_history2, g = least_squares)
MSE(weight_history3, g = least_squares)
w=best_w3
# scatter plot the input data
fig, ax = plt.subplots(1, 1, figsize=(4,4))
ax.scatter(np.log(data[0,:]),np.log(data[1,:]),color = 'k',edgecolor = 'w')
# fit a trend line
x_vals = np.linspace(-7.5,7.5,200)
y_vals = w[0] + w[1]*x_vals
ax.plot(x_vals,y_vals,color = 'r')
plt.show()
```



d. For mass of 10, input log(10)=1

output y(metabolic rate)= 28827592.44

w=[6.80825,

0.65156]

Question 5.9

5.9 **Housing price and Automobile Miles-per-Gallon prediction**
Verify the quality metrics given in Examples 5.5 and 5.6 for the Boston Housing and Automobile Miles-per-Gallon datasets. Because of the large variations in the input values of these datasets you should *standard normalize* the input features of each – as detailed in Section 9.3 – prior to optimization.

```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

csvname = 'boston_housing.csv'
```

```
data = np.loadtxt(csvname,delimiter = ',')

x=(data[0:12,:])
y=(data[13,:])
# distance=(data[8,:])
# Lowerclass=(data[12,:])

#fig, ax = plt.subplots(1, 1, figsize=(4,4))
#ax.scatter(np.log(data[0,:]),np.log(data[1,:]),color = 'k',edgecolor = 'w')
#ax.set_xlabel('features', fontsize=10)
#ax.set_ylabel('houses', fontsize='medium')
#plt.show()
#x = np.log(data[:-1,:])
#y = np.log(data[-1,:])
## defined from book

def model(x,w):
    a = w[0] +np.dot(x.T,w[1:])
    return a.T
#defined from book

def least_squares(w):
    cost = np.sum((model(x,w)-y)**2)
    return cost/float(y.size)
##book

w = 1*1*np.ones([13,1])

cost1=least_squares(w)
print(cost1)

##from page 85
def gradient_descent(g,alpha,max_its,w):
     gradient = grad(g)
     weight_history=[]

     best_w = w #Weight history container
     best_eval = g(w) #cost function history container
     for k in range(max_its):
         grad_eval = gradient(w)
         weight_history.append(w)
         w = w - alpha*grad_eval
         test_eval = g(w)
         if test_eval < best_eval:
             best_eval = test_eval
             best_w = w
     return best_w,weight_history
best_w1,weight_history1 = gradient_descent(g = least_squares,alpha = 10**-
0,max_its = 500,w = w)
best_w2,weight_history2 = gradient_descent(g = least_squares,alpha = 10**-
1,max_its = 500,w = w)
best_w3,weight_history3 = gradient_descent(g = least_squares,alpha = 10**-
2,max_its = 500,w = w)


def MSE(weight_history,g):
 # loop over weight history and compute the MSE at each step o gradient descent
 costfunchistory=[g(w) for w in weight_history]
 # plot cost function history
```

```
 plt.figure()
 plt.plot([i for i in range(500)], costfunchistory)
 plt.show()
MSE(weight_history1, g = least_squares)
MSE(weight_history2, g = least_squares)
MSE(weight_history3, g = least_squares)
w=best_w3
# # scatter plot the input data
# fig, ax = plt.subplots(1, 1, figsize=(4,4))
# ax.scatter(np.log(data[0,:]),np.log(data[1,:]),color = 'k',edgecolor = 'w')
# # fit a trend line
# x_vals = np.linspace(-7.5,7.5,200)
# y_vals = w[0] + w[1]*x_vals
# ax.plot(x_vals,y_vals,color = 'r')
# plt.show()
```