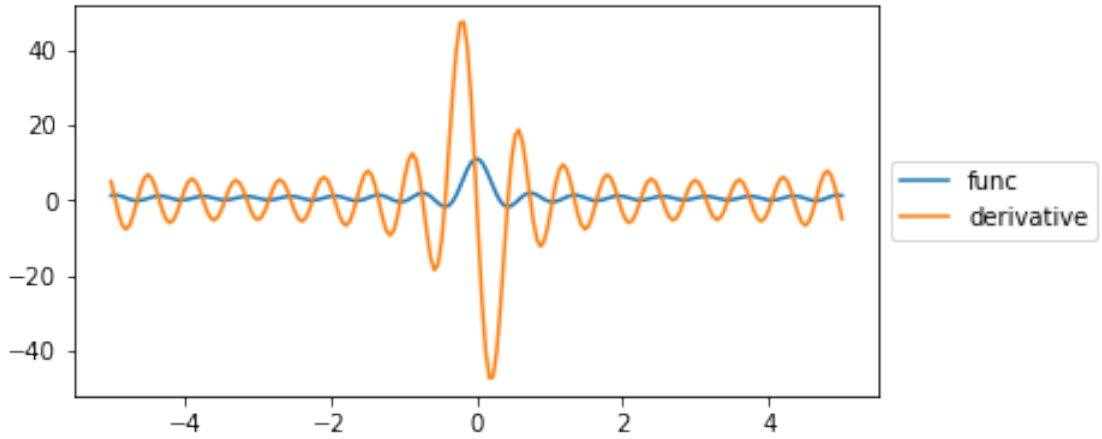
Zhisheng_Lin_HW1

January 31, 2020

Exercise 1. Python class refresher

[53]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

class GradViewer():
    def __init__(self,func):
        self.func = func

    def plot_it(self):
        # create space over which to evaluate function and gradient
        w_vals = np.linspace(-5,5,200)

        # get function
        g = self.func

        # calculate the gradient function
        nabla_g = grad(g)

        # evaluate gradient over input range
        g_vals = [g(v) for v in w_vals]
        grad_vals = [nabla_g(v) for v in w_vals]

        # create figure
        fig, ax = plt.subplots(1, 1, figsize=(6,3))

        # plot function and gradient values
        ax.plot(w_vals,g_vals)
        ax.plot(w_vals,grad_vals)
        ax.legend(['func','derivative'],loc='center left', bbox_to_anchor=(1, 0.
 ↪5))
        plt.show()

# test function input
def my_function(w):
    # initialize y for storing the sum
    y = 0
```

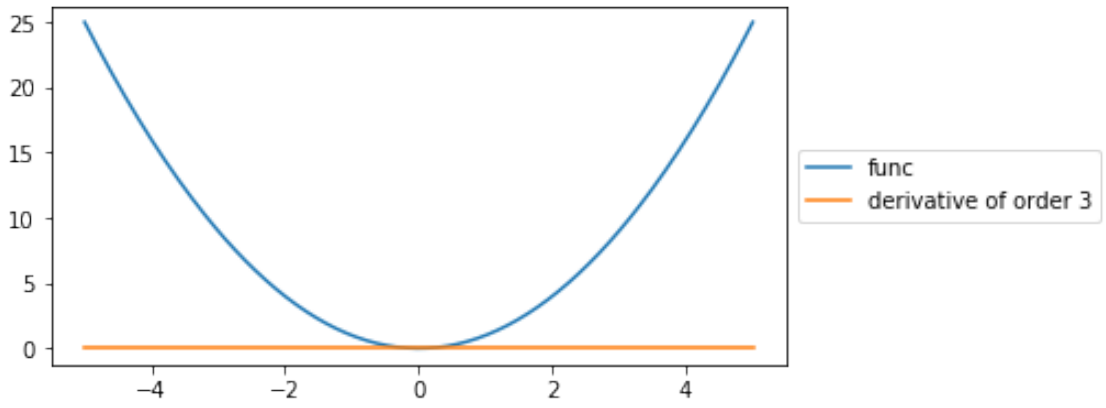
```
    # use for loop to realize summing
    for i in range(11):
        y = y+np.cos(i*w)
    return y

# create instance, inputting function on creation
test = GradViewer(my_function)

# plot this derivative and original function
test.plot_it()
```



Exercise 2. Python class refresher part 2

[66]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

class OrderViewer(GradViewer):
    def __init__(self,func):
        super().__init__(func)

    def compute_it(self,order):
        global deri
        deri = self.func
        for i in range(int(order)):
            deri = grad(deri)
        return deri

    def plot_it(self):
        # create space over which to evaluate function and gradient
        w_vals = np.linspace(-5,5,200)
```

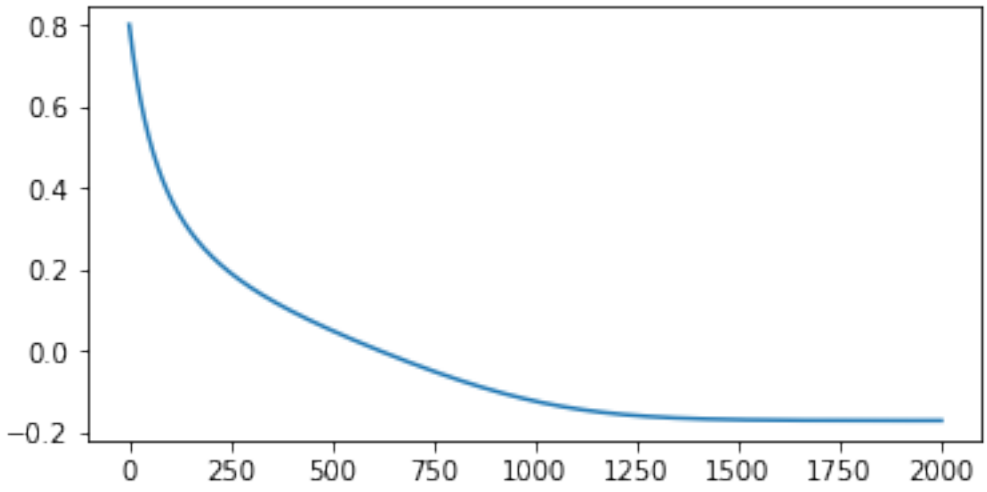
```
        # get function
        g = self.func
        # calculate the gradient function
        nabla_g = deri

        # evaluate gradient over input range
        g_vals = [g(v) for v in w_vals]
        grad_vals = [nabla_g(v) for v in w_vals]

        # create figure
        fig, ax = plt.subplots(1, 1, figsize=(6,3))

        # plot function and gradient values
        ax.plot(w_vals,g_vals)
        ax.plot(w_vals,grad_vals)
        ax.legend(['func','derivative of order 3'],loc='center left',␣
 ↪bbox_to_anchor=(1, 0.5))
        plt.show()


# test function input
def my_function(w):
    y = w**2
    return y

# create instance, inputting function on creation
test = OrderViewer(my_function)

# compute desired order derivative
test.compute_it(order = 3)

# plot this derivative and original function
test.plot_it()
```



Exercise 3. A generic gradient descent function *Input Function* 1 : $g = \frac{1}{50}(w^4 + w^2 + 10w)$

[115]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

# gradient descent function - inputs: g (input function), alpha (steplength␣
 ↪parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute the gradient of our input function - note this is a function too!
    gradient = grad(g)

    # run the gradient descent loop
    best_w = w        # weight we return, should be the one providing lowest␣
 ↪evaluation
    best_eval = g(w)       # lowest evaluation yet
    best_w_arr = []       # store weights in each step
    best_w_arr.append(best_w)       # store the first weight
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # return only the weight providing the lowest evaluation
        test_eval = g(w)
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
        best_w_arr.append(best_w)

    return best_w_arr

# create the input function
g = lambda w: 1/float(50)*(w**4 + w**2 + 10*w)    # try other functions too! ␣
 ↪Like g = lambda w: np.cos(2*w) , g = lambda w: np.sin(5*w) + 0.1*w**2, g =␣
 ↪lambda w: np.cos(5*w)*np.sin(w)

# run gradient descent
weight_history = gradient_descent(g = g,alpha = 10**-2,max_its = 2000,w = 2.0)

# cost function history plotter
def cost_history(weight_history,g):
```

```
    # loop over weight history and compute associated cost function history at␣
↪each step
    weight_history = np.asarray(weight_history)
    cost_history = g(weight_history)

    # plot cost function history
    # create figure
    fig, ax = plt.subplots(1, 1, figsize=(6,3))
    # plot function and gradient values
    ax.plot(np.linspace(0,2000,len(cost_history)),cost_history)
    plt.show()

    # compare the weight corresponding to the smallest cost value with the␣
↪expected weight
    small_costval_arg = np.argmin(cost_history)
    small_weightval = weight_history[small_costval_arg]
    w_exp = ((2031**(1/2)-45)**(1/3))/(6**(2/3))-1/(((2031**(1/2)-45)*6)**(1/3))
    print("The weight corresponding to the smallest cost function value is␣
↪",end='')
    print(small_weightval)
    print("The expected weight is ",end='')
    print(w_exp)
    print("So, they are close.")

# use cost_history plotter
cost_history(weight_history,g)
```



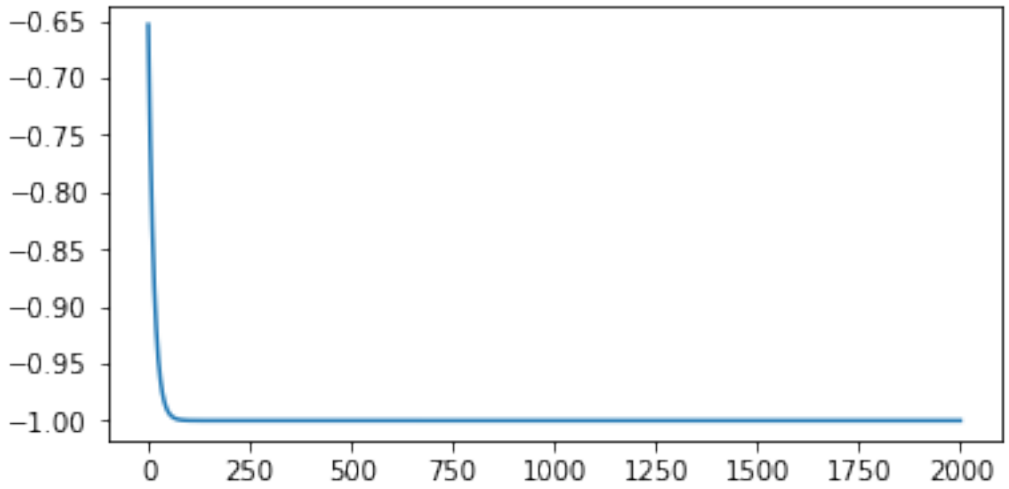

```
The weight corresponding to the smallest cost function value is
-1.2198339861245202
The expected weight is -1.2347728250533112
```

So, they are close.

Input Function 2 : $g = cos(2w)$

[112]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

# gradient descent function - inputs: g (input function), alpha (steplength␣
 ↪parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute the gradient of our input function - note this is a function too!
    gradient = grad(g)

    # run the gradient descent loop
    best_w = w        # weight we return, should be the one providing lowest␣
 ↪evaluation
    best_eval = g(w)       # lowest evaluation yet
    best_w_arr = []       # store weights in each step
    best_w_arr.append(best_w)       # store the first weight
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # return only the weight providing the lowest evaluation
        test_eval = g(w)
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
        best_w_arr.append(best_w)

    return best_w_arr

# create the input function
g = lambda w: np.cos(2*w)    # try other functions too!  Like g = lambda w: np.
 ↪cos(2*w) , g = lambda w: np.sin(5*w) + 0.1*w**2, g = lambda w: np.cos(5*w)*np.
 ↪sin(w)

# run gradient descent
weight_history = gradient_descent(g = g,alpha = 10**-2,max_its = 2000,w = 2.0)

# cost function history plotter
def cost_history(weight_history,g):
    # loop over weight history and compute associated cost function history at␣
 ↪each step
```

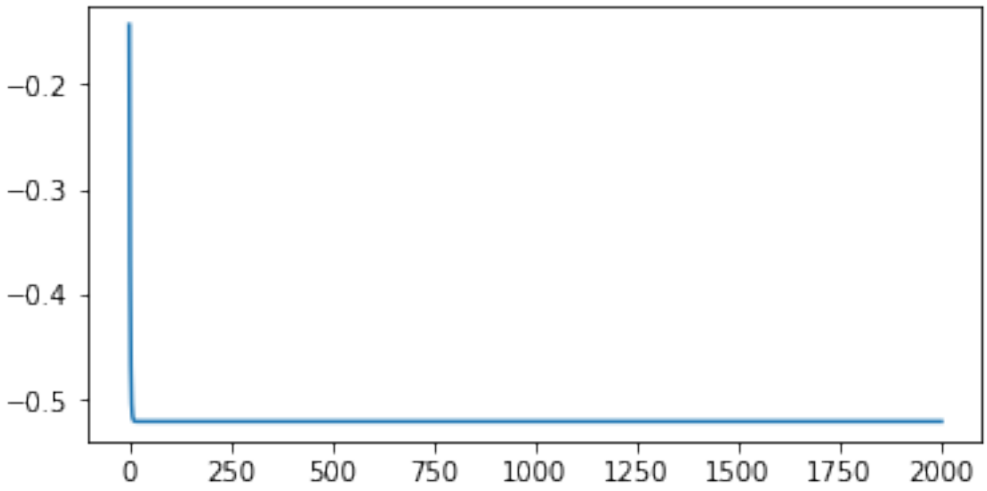
```
    weight_history = np.asarray(weight_history)
    cost_history = g(weight_history)

    # plot cost function history
    # create figure
    fig, ax = plt.subplots(1, 1, figsize=(6,3))
    # plot function and gradient values
    ax.plot(np.linspace(0,2000,len(cost_history)),cost_history)
    plt.show()

# use cost_history plotter
cost_history(weight_history,g)
```



Input Function 3 : $g = sin(5w) + 0.1w^2$

[113]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

# gradient descent function - inputs: g (input function), alpha (steplength␣
 ↪parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute the gradient of our input function - note this is a function too!
    gradient = grad(g)

    # run the gradient descent loop
    best_w = w        # weight we return, should be the one providing lowest␣
 ↪evaluation
    best_eval = g(w)       # lowest evaluation yet
    best_w_arr = []       # store weights in each step
```

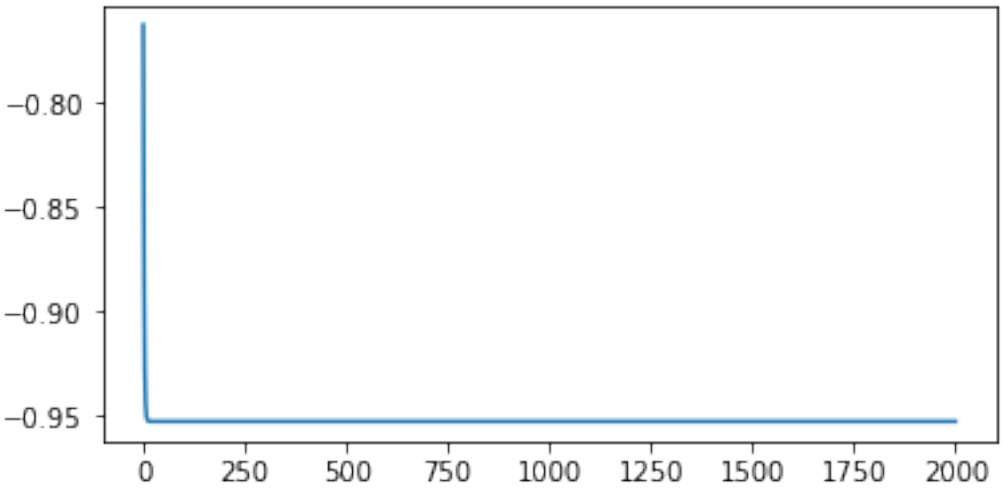
```
    best_w_arr.append(best_w)       # store the first weight
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # return only the weight providing the lowest evaluation
        test_eval = g(w)
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
        best_w_arr.append(best_w)

    return best_w_arr

# create the input function
g = lambda w: np.sin(5*w) + 0.1*w**2    # try other functions too!  Like g =␣
 ↪lambda w: np.cos(2*w) , g = lambda w: np.sin(5*w) + 0.1*w**2, g = lambda w: np.
 ↪cos(5*w)*np.sin(w)

# run gradient descent
weight_history = gradient_descent(g = g,alpha = 10**-2,max_its = 2000,w = 2.0)

# cost function history plotter
def cost_history(weight_history,g):
    # loop over weight history and compute associated cost function history at␣
 ↪each step
    weight_history = np.asarray(weight_history)
    cost_history = g(weight_history)

    # plot cost function history
    # create figure
    fig, ax = plt.subplots(1, 1, figsize=(6,3))
    # plot function and gradient values
    ax.plot(np.linspace(0,2000,len(cost_history)),cost_history)
    plt.show()

# use cost_history plotter
cost_history(weight_history,g)
```



Input Function 4 : $g = cos(5w)sin(w)$

[146]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

# gradient descent function - inputs: g (input function), alpha (steplength␣
 ↪parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute the gradient of our input function - note this is a function too!
    gradient = grad(g)

    # run the gradient descent loop
    best_w = w        # weight we return, should be the one providing lowest␣
 ↪evaluation
    best_eval = g(w)       # lowest evaluation yet
    best_w_arr = []       # store weights in each step
    best_w_arr.append(best_w)       # store the first weight
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # return only the weight providing the lowest evaluation
        test_eval = g(w)
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
```

```
        best_w_arr.append(best_w)

    return best_w_arr

# create the input function
g = lambda w: np.cos(5*w)*np.sin(w)    # try other functions too!  Like g =␣
 ↪lambda w: np.cos(2*w) , g = lambda w: np.sin(5*w) + 0.1*w**2, g = lambda w: np.
 ↪cos(5*w)*np.sin(w)

# run gradient descent
weight_history = gradient_descent(g = g,alpha = 10**-2,max_its = 2000,w = 2.0)

# cost function history plotter
def cost_history(weight_history,g):
    # loop over weight history and compute associated cost function history at␣
 ↪each step
    weight_history = np.asarray(weight_history)
    cost_history = g(weight_history)

    # plot cost function history
    # create figure
    fig, ax = plt.subplots(1, 1, figsize=(6,3))
    # plot function and gradient values
    ax.plot(np.linspace(0,2000,len(cost_history)),cost_history)
    plt.show()

# use cost_history plotter
cost_history(weight_history,g)
```

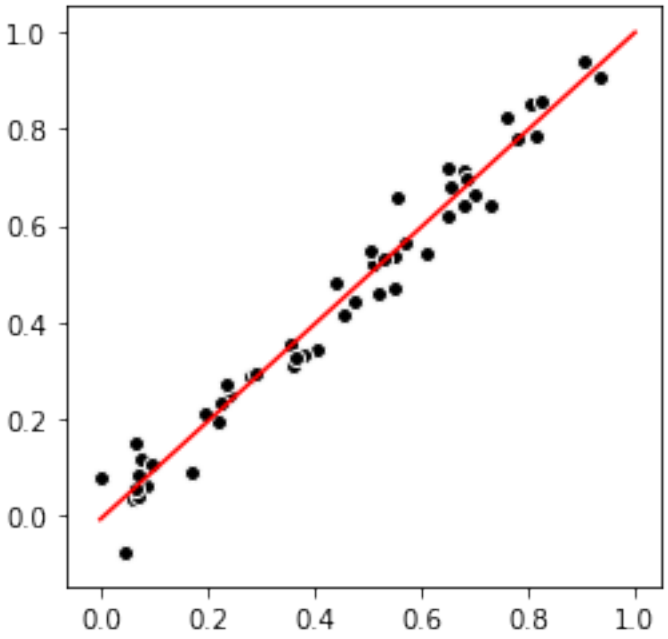


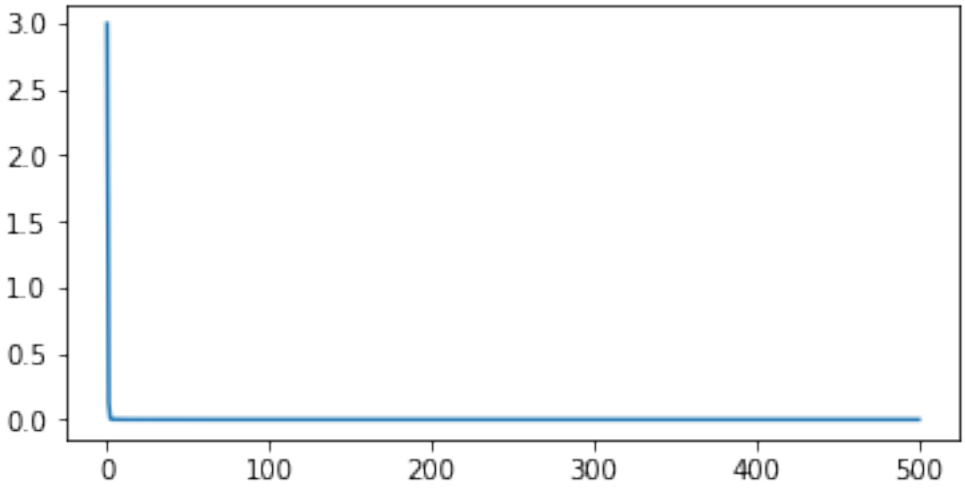
Exercise 4. Apply gradient descent to minimize the Least Squares cost for linear regression on a low dimensional dataset 1. $\alpha_1 = 0.01$

[161]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

# data input
csvname = '2d_linregress_data.csv'
data = np.loadtxt(csvname,delimiter = ',')

# form the input/output data vectors
x = data[:,:-1]
y = data[:,-1]

# least squares cost function for linear regression
def least_squares(w):
    cost = 0
    for p in range(len(y)):
        # get pth input/output pair
        x_p = x[p]
        y_p = y[p]

        # form linear combination
        c_p = w[0] + w[1]*x_p

        # add least squares for this datapoint
        cost += (c_p - y_p)**2

    return cost


# gradient descent function - inputs: g (input function), alpha (steplength␣
 ↪parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute the gradient of our input function - note this is a function too!
    gradient = grad(g)

    # run the gradient descent loop
    best_w = w        # weight we return, should be the one providing lowest␣
 ↪evaluation
    best_eval = g(w)       # lowest evaluation yet

    best_w_arr = []       # store weights in each step
    best_w_arr.append(best_w)       # store the first weight
    for k in range(max_its):
        # evaluate the gradient
```

```
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # return only the weight providing the lowest evaluation
        test_eval = g(w)
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
        best_w_arr.append(best_w)

    return best_w_arr

# run gradient descent
w = np.asarray([1.5,1.5])
weight_history = gradient_descent(g = least_squares,alpha = 10**-2,max_its =␣
 ↪500,w = w)
weight_history = np.asarray(weight_history)

# MSE history plotter
def MSE(weight_history,g):
    MSE_arr = []
    # loop over weight history and compute the MSE at each step o gradient␣
 ↪descent
    for i in range(len(weight_history)):
        MSE = g(weight_history[i])/len(y)
        MSE_arr.append(MSE)

    # plot MSE
    # create figure
    fig, ax = plt.subplots(1, 1, figsize=(6,3))
    # plot function and gradient values
    ax.plot(np.linspace(0,500,len(MSE_arr)),MSE_arr)
    plt.show()

# scatter plot the input data
fig, ax = plt.subplots(1, 1, figsize=(4,4))
ax.scatter(data[:,0],data[:,1],color = 'k',edgecolor = 'w')

# fit a trend line
x_vals = np.linspace(0,1,200)
y_vals = weight_history[-1,0] + weight_history[-1,1]*x_vals
ax.plot(x_vals,y_vals,color = 'r')
plt.show()

# plot MSE
```

```
MSE(weight_history,least_squares)
```



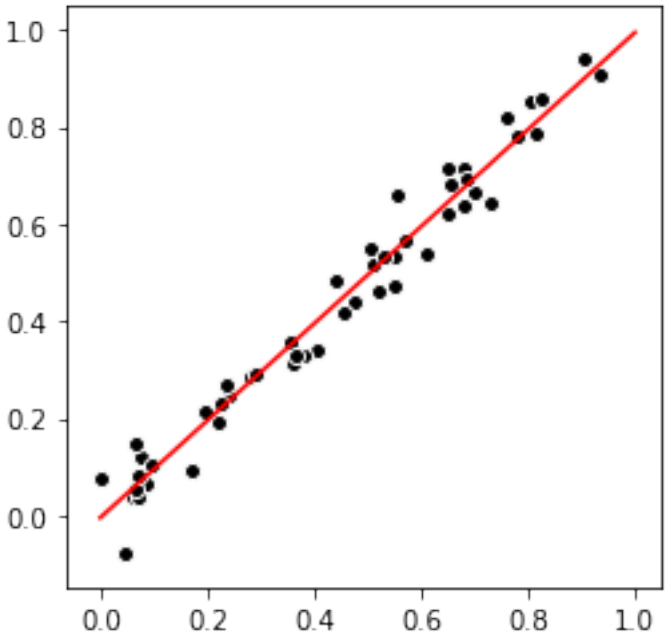


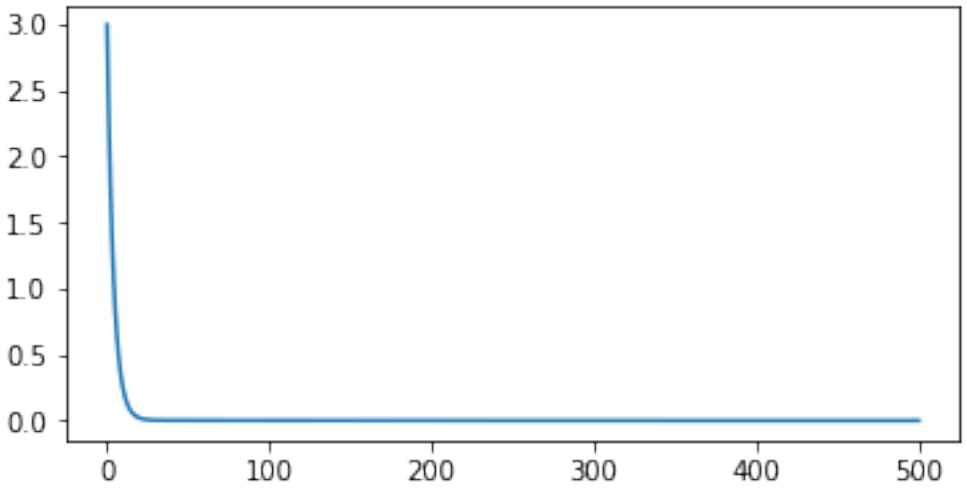
2. $\alpha_2 = 0.001$

[2]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad
```

```
# data input
csvname = '2d_linregress_data.csv'
data = np.loadtxt(csvname,delimiter = ',')

# form the input/output data vectors
x = data[:,:-1]
y = data[:,-1]

# least squares cost function for linear regression
def least_squares(w):
    cost = 0
    for p in range(len(y)):
        # get pth input/output pair
        x_p = x[p]
        y_p = y[p]

        # form linear combination
        c_p = w[0] + w[1]*x_p

        # add least squares for this datapoint
        cost += (c_p - y_p)**2

    return cost


# gradient descent function - inputs: g (input function), alpha (steplength␣
 ↪parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute the gradient of our input function - note this is a function too!
    gradient = grad(g)

    # run the gradient descent loop
    best_w = w        # weight we return, should be the one providing lowest␣
 ↪evaluation
    best_eval = g(w)       # lowest evaluation yet

    best_w_arr = []       # store weights in each step
    best_w_arr.append(best_w)       # store the first weight
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # return only the weight providing the lowest evaluation
        test_eval = g(w)
```

```
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
        best_w_arr.append(best_w)

    return best_w_arr

# run gradient descent
w = np.asarray([1.5,1.5])
weight_history = gradient_descent(g = least_squares,alpha = 10**-3,max_its =␣
 ↪500,w = w)
weight_history = np.asarray(weight_history)

# MSE history plotter
def MSE(weight_history,g):
    MSE_arr = []
    # loop over weight history and compute the MSE at each step o gradient␣
 ↪descent
    for i in range(len(weight_history)):
        MSE = g(weight_history[i])/len(y)
        MSE_arr.append(MSE)

    # plot MSE
    # create figure
    fig, ax = plt.subplots(1, 1, figsize=(6,3))
    # plot function and gradient values
    ax.plot(np.linspace(0,500,len(MSE_arr)),MSE_arr)
    plt.show()

# scatter plot the input data
fig, ax = plt.subplots(1, 1, figsize=(4,4))
ax.scatter(data[:,0],data[:,1],color = 'k',edgecolor = 'w')

# fit a trend line
x_vals = np.linspace(0,1,200)
y_vals = weight_history[-1,0] + weight_history[-1,1]*x_vals
ax.plot(x_vals,y_vals,color = 'r')
plt.show()

# plot MSE
MSE(weight_history,least_squares)
```



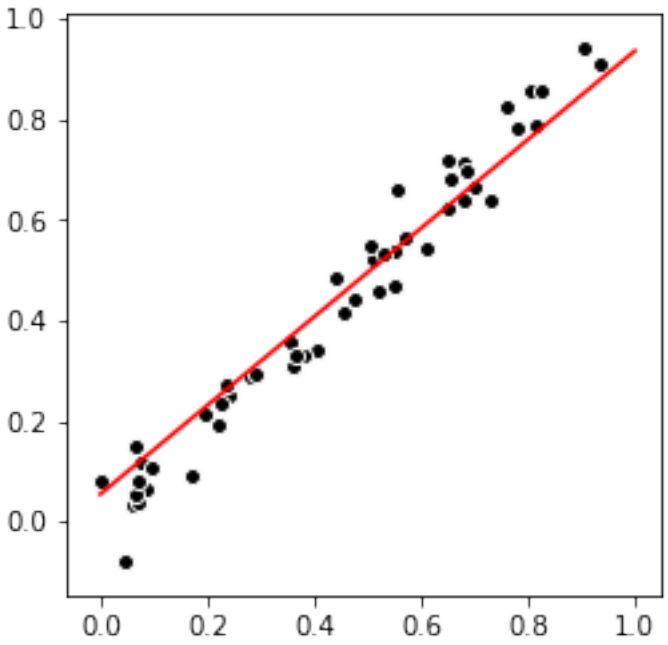


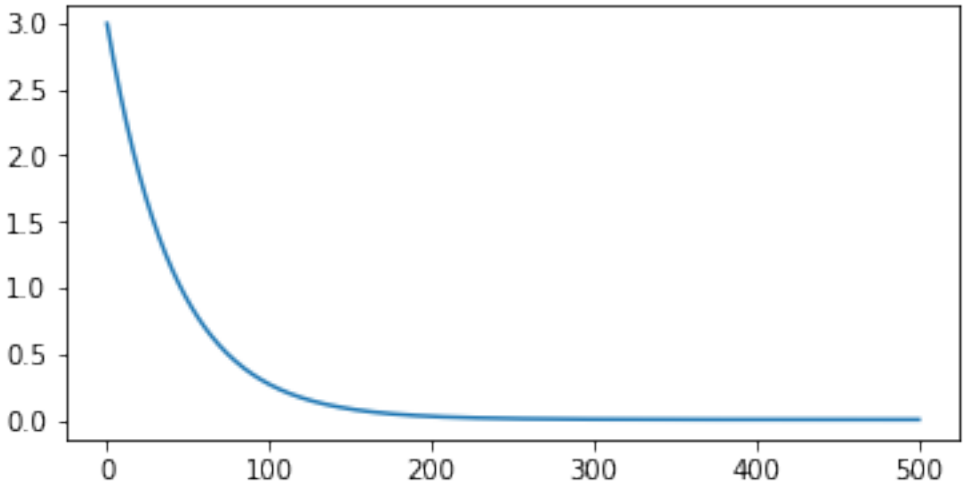
3. $\alpha_3 = 0.0001$

[3]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

# data input
csvname = '2d_linregress_data.csv'
data = np.loadtxt(csvname,delimiter = ',')
```

```
# form the input/output data vectors
x = data[:,:-1]
y = data[:,-1]

# least squares cost function for linear regression
def least_squares(w):
    cost = 0
    for p in range(len(y)):
        # get pth input/output pair
        x_p = x[p]
        y_p = y[p]

        # form linear combination
        c_p = w[0] + w[1]*x_p

        # add least squares for this datapoint
        cost += (c_p - y_p)**2

    return cost


# gradient descent function - inputs: g (input function), alpha (steplength␣
 ↪parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute the gradient of our input function - note this is a function too!
    gradient = grad(g)

    # run the gradient descent loop
    best_w = w        # weight we return, should be the one providing lowest␣
 ↪evaluation
    best_eval = g(w)       # lowest evaluation yet

    best_w_arr = []       # store weights in each step
    best_w_arr.append(best_w)       # store the first weight
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # return only the weight providing the lowest evaluation
        test_eval = g(w)
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
```

```
        best_w_arr.append(best_w)

    return best_w_arr

# run gradient descent
w = np.asarray([1.5,1.5])
weight_history = gradient_descent(g = least_squares,alpha = 10**-4,max_its =␣
 ↪500,w = w)
weight_history = np.asarray(weight_history)

# MSE history plotter
def MSE(weight_history,g):
    MSE_arr = []
    # loop over weight history and compute the MSE at each step o gradient␣
 ↪descent
    for i in range(len(weight_history)):
        MSE = g(weight_history[i])/len(y)
        MSE_arr.append(MSE)

    # plot MSE
    # create figure
    fig, ax = plt.subplots(1, 1, figsize=(6,3))
    # plot function and gradient values
    ax.plot(np.linspace(0,500,len(MSE_arr)),MSE_arr)
    plt.show()

# scatter plot the input data
fig, ax = plt.subplots(1, 1, figsize=(4,4))
ax.scatter(data[:,0],data[:,1],color = 'k',edgecolor = 'w')

# fit a trend line
x_vals = np.linspace(0,1,200)
y_vals = weight_history[-1,0] + weight_history[-1,1]*x_vals
ax.plot(x_vals,y_vals,color = 'r')
plt.show()

# plot MSE
MSE(weight_history,least_squares)
```



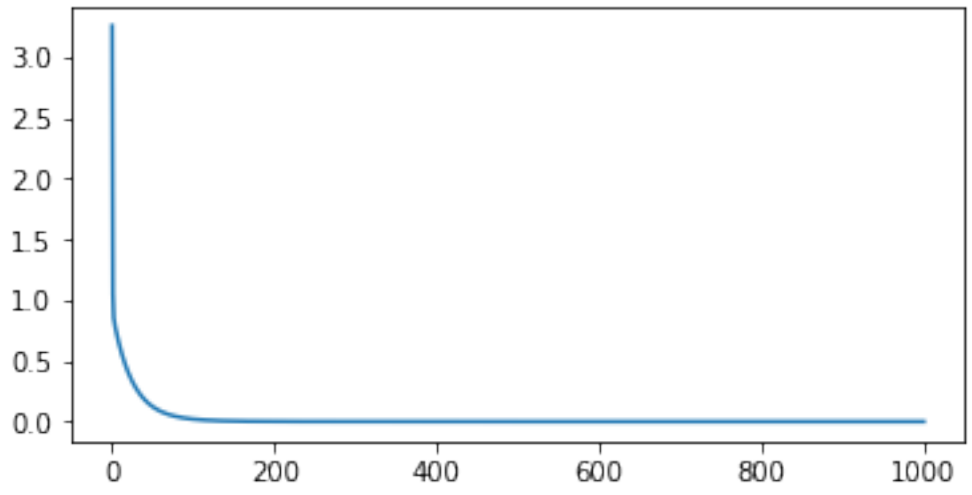


Exercise 5. Use linear regression to fit to a high dimensional dataset 1. $\alpha = 0.001$, *iterations* = 1000

[2]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

# load in dataset
```

```
data = np.loadtxt('highdim_linregress_data.csv',delimiter = ',')

# form the input/output data vectors
x = data[:,:-1]
y = data[:,-1]

# change y to a array
y = np.asarray(y).reshape(100,1)

# concatenate 11 by 1 all one vector on the left of input x
o = np.ones((np.shape(x)[0],1))
x = np.concatenate((o,x),axis=1)

# least squares cost function for linear regression
def least_squares(w):
    # get the calculation value
    c_p = x.dot(w)
    # get the real value
    y_p = y

    # get the cost
    cost = np.sum((c_p-y_p)**2)

    return cost

# gradient descent function - inputs: g (input function), alpha (steplength␣
 ↪parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute the gradient of our input function - note this is a function too!
    gradient = grad(g)

    # run the gradient descent loop
    best_w = w        # weight we return, should be the one providing lowest␣
 ↪evaluation
    best_eval = g(w)       # lowest evaluation yet

    best_w_arr = []       # store weights in each step
    best_w_arr.append(best_w)       # store the first weight
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # return only the weight providing the lowest evaluation
        test_eval = g(w)
```

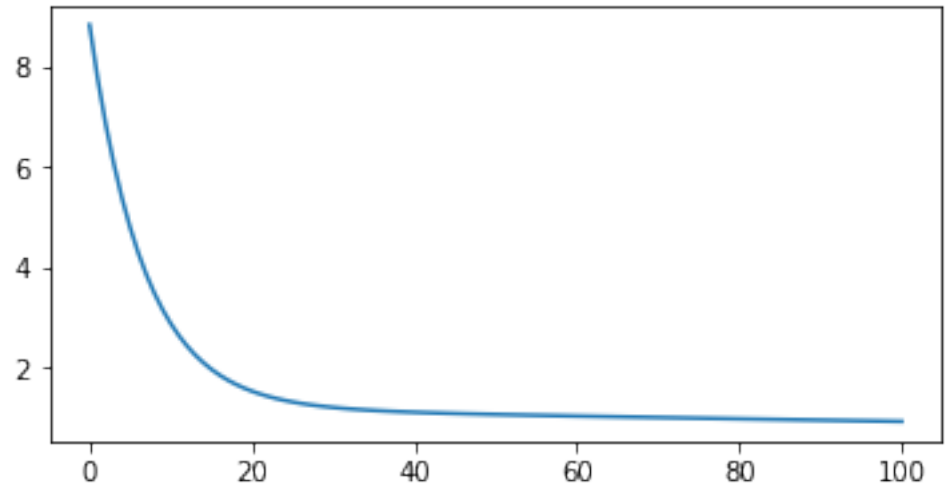
```
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
        best_w_arr.append(best_w)

    return best_w_arr

# run gradient descent
w = np.random.randn(np.shape(x)[1],1)
weight_history = gradient_descent(g = least_squares,alpha = 10**-3,max_its =␣
 ↪1000,w = w)
weight_history = np.asarray(weight_history)

# MSE history plotter
def MSE(weight_history,g):
    MSE_arr = []
    # loop over weight history and compute the MSE at each step o gradient␣
 ↪descent
    for i in range(len(weight_history)):
        MSE = g(weight_history[i])/len(y)
        MSE_arr.append(MSE)

    # plot MSE
    # create figure
    fig, ax = plt.subplots(1, 1, figsize=(6,3))
    # plot function and gradient values
    ax.plot(np.linspace(0,1000,len(MSE_arr)),MSE_arr)
    plt.show()

# plot MSE
MSE(weight_history,least_squares)
```



2. $\alpha = 0.0001$, $iterations = 100$

[189]:
```
import autograd.numpy as np
import matplotlib.pyplot as plt
from autograd import grad

# load in dataset
data = np.loadtxt('highdim_linregress_data.csv',delimiter = ',')

# form the input/output data vectors
x = data[:,:-1]
y = data[:,-1]

# change y to a array
y = np.asarray(y).reshape(100,1)

# concatenate 11 by 1 all one vector on the left of input x
o = np.ones((np.shape(x)[0],1))
x = np.concatenate((o,x),axis=1)

# least squares cost function for linear regression
def least_squares(w):
    # get the calculation value
    c_p = x.dot(w)
    # get the real value
    y_p = y

    # get the cost
    cost = np.sum((c_p-y_p)**2)

    return cost

# gradient descent function - inputs: g (input function), alpha (steplength␣
 ↪parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha,max_its,w):
    # compute the gradient of our input function - note this is a function too!
    gradient = grad(g)

    # run the gradient descent loop
    best_w = w        # weight we return, should be the one providing lowest␣
 ↪evaluation
    best_eval = g(w)       # lowest evaluation yet

    best_w_arr = []       # store weights in each step
    best_w_arr.append(best_w)       # store the first weight
```

```
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # return only the weight providing the lowest evaluation
        test_eval = g(w)
        if test_eval < best_eval:
            best_eval = test_eval
            best_w = w
        best_w_arr.append(best_w)

    return best_w_arr

# run gradient descent
w = np.random.randn(np.shape(x)[1],1)
weight_history = gradient_descent(g = least_squares,alpha = 10**-4,max_its =␣
 ↪100,w = w)
weight_history = np.asarray(weight_history)

# MSE history plotter
def MSE(weight_history,g):
    MSE_arr = []
    # loop over weight history and compute the MSE at each step o gradient␣
 ↪descent
    for i in range(len(weight_history)):
        MSE = g(weight_history[i])/len(y)
        MSE_arr.append(MSE)

    # plot MSE
    # create figure
    fig, ax = plt.subplots(1, 1, figsize=(6,3))
    # plot function and gradient values
    ax.plot(np.linspace(0,100,len(MSE_arr)),MSE_arr)
    plt.show()

# plot MSE
MSE(weight_history,least_squares)
```


8
6
4
2
0
20
40
60
80
100