Importing libraries and Defining Visualizer

In [1]:
```python
import autograd.numpy as np
from autograd import grad
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
from math import tanh
```

In [24]:
```python
# import standard plotting and animation
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import gridspec
from IPython.display import clear_output
from mpl_toolkits.mplot3d import proj3d
from matplotlib.patches import FancyArrowPatch
from matplotlib.text import Annotation
from mpl_toolkits.mplot3d.proj3d import proj_transform

import math
import time
import copy


class Visualizer:
    '''
    Illustrate a run of your preferred optimization algorithm on a one or two-input function.  Run
    the algorithm first, and input the resulting weight history into this wrapper.
    '''
    ##### draw picture of function and run for two-input function ####
    def two_input_surface_contour_plot(self,g,w_hist,**kwargs):
        ### input arguments ###
        num_contours = 10
        if 'num_contours' in kwargs:
            num_contours = kwargs['num_contours']

        view = [20,20]
        if 'view' in kwargs:
            view = kwargs['view']

        ##### construct figure with panels #####
        # construct figure
        fig = plt.figure(figsize = (11,5))
        self.edgecolor = 'k'

        # create subplot with 3 panels, plot input function in center plot
        # this seems to be the best option for whitespace management when using
        # both a surface and contour plot in the same figure
        gs = gridspec.GridSpec(1, 3, width_ratios=[1,5,10])
        ax1 = plt.subplot(gs[1],projection='3d');
        ax2 = plt.subplot(gs[2],aspect='equal');

        # remove whitespace from figure
        fig.subplots_adjust(left=0, right=1, bottom=0, top=1) # remove whitespace
        fig.subplots_adjust(wspace=0.01,hspace=0.01)

        # plot 3d surface and path in left panel
        self.draw_surface(g,ax1,**kwargs)
        self.show_inputspace_path(w_hist,ax1)
        ax1.view_init(view[0],view[1])

        ### make contour right plot - as well as horizontal and vertical axes ###
        self.contour_plot_setup(g,ax2,**kwargs)  # draw contour plot
        self.draw_weight_path(ax2,w_hist)              # draw path on contour plot

        # plot
        plt.show()



    ##### draw picture of function and run for two-input function ####
    def compare_runs_contour_plots(self,g,weight_histories,titles,**kwargs):
        ##### construct figure with panels #####
        # construct figure
        fig = plt.figure(figsize = (10,4.5))
        self.edgecolor = 'k'
        fig.suptitle('Contours for cost function g and weight history. ', fontsize=16)
        # create figure with single plot for contour
        gs = gridspec.GridSpec(1, 3)
        ax1 = plt.subplot(gs[0],aspect='equal');
        ax1.title.set_text(titles[0])

        ax2 = plt.subplot(gs[1],aspect='equal');
        ax2.title.set_text(titles[1])

        ax3 = plt.subplot(gs[2],aspect='equal');
        ax3.title.set_text(titles[2])

        # remove whitespace from figure
        fig.subplots_adjust(left=0, right=1, bottom=0, top=1) # remove whitespace
        fig.subplots_adjust(wspace=0.01,hspace=0.01)

        ### make contour right plot - as well as horizontal and vertical axes ###
        self.contour_plot_setup(g,ax1,**kwargs)  # draw contour plot
        w_hist = weight_histories[0]
        self.draw_weight_path(ax1,w_hist)        # draw path on contour plot

        self.contour_plot_setup(g,ax2,**kwargs)  # draw contour plot
        w_hist = weight_histories[1]
        self.draw_weight_path(ax2,w_hist)        # draw path on contour plot
        ax2.set_yticks([])
        ax2.set_ylabel('')

        self.contour_plot_setup(g,ax3,**kwargs)  # draw contour plot
        w_hist = weight_histories[2]
        self.draw_weight_path(ax3,w_hist)
        ax3.set_yticks([])
        ax3.set_ylabel('')

        # plot
        plt.show()


    # compare cost histories from multiple runs
    def plot_cost_histories(self,histories,start,**kwargs):
        # plotting colors
        colors = ['k','magenta','aqua','blueviolet','chocolate']

        # initialize figure
        fig = plt.figure(figsize = (10,3))

        # create subplot with 1 panel
        gs = gridspec.GridSpec(1, 1)
        ax = plt.subplot(gs[0]);

        # any labels to add?
        labels = [' ',' ']
        if 'labels' in kwargs:
            labels = kwargs['labels']

        # plot points on cost function plot too?
        points = False
        if 'points' in kwargs:
            points = kwargs['points']

        # run through input histories, plotting each beginning at 'start' iteration
        for c in range(len(histories)):
            history = histories[c]
            label = 0
            if c == 0:
                label = labels[0]
            else:
                label = labels[1]

            # check if a label exists, if so add it to the plot
            if np.size(label) == 0:
                ax.plot(np.arange(start,len(history),1),history[start:],linewidth = 3*(0.8)**(c),color = colors
            else:
                ax.plot(np.arange(start,len(history),1),history[start:],linewidth = 3*(0.8)**(c),color = colors

            # check if points should be plotted for visualization purposes
            if points == True:
                ax.scatter(np.arange(start,len(history),1),history[start:],s = 90,color = colors[c],edgecolor =


        # clean up panel
        xlabel = 'step $k$'
        if 'xlabel' in kwargs:
            xlabel = kwargs['xlabel']
        ylabel = r'$g\left(\mathbf{w}^k\right)$'
        if 'ylabel' in kwargs:
            ylabel = kwargs['ylabel']
        ax.set_xlabel(xlabel,fontsize = 14)
        ax.set_ylabel(ylabel,fontsize = 14,rotation = 0,labelpad = 25)
        if np.size(label) > 0:
            anchor = (1,1)
            if 'anchor' in kwargs:
                anchor = kwargs['anchor']
            plt.legend(loc='upper right', bbox_to_anchor=anchor)
            #leg = ax.legend(loc='upper left', bbox_to_anchor=(1.02, 1), borderaxespad=0)

        ax.set_xlim([start - 0.5,len(history) - 0.5])

       # fig.tight_layout()
        plt.show()


    ######################################################################################
    #### utility functions - for setting up / making contour plots, 3d surface plots, etc., ####
    # show contour plot of input function
    def contour_plot_setup(self,g,ax,**kwargs):
        xmin = -3.1
        xmax = 3.1
        ymin = -3.1
        ymax = 3.1
        if 'xmin' in kwargs:
            xmin = kwargs['xmin']
        if 'xmax' in kwargs:
            xmax = kwargs['xmax']
        if 'ymin' in kwargs:
            ymin = kwargs['ymin']
        if 'ymax' in kwargs:
            ymax = kwargs['ymax']
        num_contours = 20
        if 'num_contours' in kwargs:
            num_contours = kwargs['num_contours']

        # choose viewing range using weight history?
        if 'view_by_weights' in kwargs:
            view_by_weights = True
            weight_history = kwargs['weight_history']
            if view_by_weights == True:
                xmin = min([v[0] for v in weight_history])[0]
                xmax = max([v[0] for v in weight_history])[0]
                xgap = (xmax - xmin)*0.25
                xmin -= xgap
                xmax += xgap

                ymin = min([v[1] for v in weight_history])[0]
                ymax = max([v[1] for v in weight_history])[0]
                ygap = (ymax - ymin)*0.25
                ymin -= ygap
                ymax += ygap

        ### plot function as contours ###
        self.draw_contour_plot(g,ax,num_contours,xmin,xmax,ymin,ymax)

        ### cleanup panel ###
        ax.set_xlabel('$w_0$',fontsize = 14)
        ax.set_ylabel('$w_1$',fontsize = 14,labelpad = 15,rotation = 0)
        ax.axhline(y=0, color='k',zorder = 0,linewidth = 0.5)
        ax.axvline(x=0, color='k',zorder = 0,linewidth = 0.5)
        # ax.set_xticks(np.arange(round(xmin),round(xmax)+1))
        # ax.set_yticks(np.arange(round(ymin),round(ymax)+1))

        # set viewing limits
        ax.set_xlim(xmin,xmax)
        ax.set_ylim(ymin,ymax)

    ### function for creating contour plot
    def draw_contour_plot(self,g,ax,num_contours,xmin,xmax,ymin,ymax):
        #### define input space for function and evaluate ####
        w1 = np.linspace(xmin,xmax,400)
        w2 = np.linspace(ymin,ymax,400)
        w1_vals, w2_vals = np.meshgrid(w1,w2)
        w1_vals.shape = (len(w1)**2,1)
        w2_vals.shape = (len(w2)**2,1)
        h = np.concatenate((w1_vals,w2_vals),axis=1)
        func_vals = np.asarray([ g(np.reshape(s,(2,1))) for s in h])

        w1_vals.shape = (len(w1),len(w1))
        w2_vals.shape = (len(w2),len(w2))
        func_vals.shape = (len(w1),len(w2))

        ### make contour right plot - as well as horizontal and vertical axes ###
        # set level ridges
        levelmin = min(func_vals.flatten())
        levelmax = max(func_vals.flatten())
        cutoff = 1
        cutoff = (levelmax - levelmin)*cutoff
        numper = 4
        levels1 = np.linspace(cutoff,levelmax,numper)
        num_contours -= numper

        # produce generic contours
        levels2 = np.linspace(levelmin,cutoff,min(num_contours,numper))
        levels = np.unique(np.append(levels1,levels2))
        num_contours -= numper
        while num_contours > 0:
            cutoff = levels[1]
            levels2 = np.linspace(levelmin,cutoff,min(num_contours,numper))
            levels = np.unique(np.append(levels2,levels))
            num_contours -= numper

        # plot the contours
        ax.contour(w1_vals, w2_vals, func_vals,levels = levels[1:],colors = 'k')
        ax.contourf(w1_vals, w2_vals, func_vals,levels = levels,cmap = 'Blues')

        ###### clean up plot ######
        ax.set_xlabel('$w_0$',fontsize = 12)
        ax.set_ylabel('$w_1$',fontsize = 12,rotation = 0)
        ax.axhline(y=0, color='k',zorder = 0,linewidth = 0.5)
        ax.axvline(x=0, color='k',zorder = 0,linewidth = 0.5)


    ### makes color spectrum for plotted run points - from green (start) to red (stop)
    def make_colorspec(self,w_hist):
        # make color range for path
        s = np.linspace(0,1,len(w_hist[:round(len(w_hist)/2)]))
        s.shape = (len(s),1)
        t = np.ones(len(w_hist[round(len(w_hist)/2):]))
        t.shape = (len(t),1)
        s = np.vstack((s,t))
        colorspec = []
        colorspec = np.concatenate((s,np.flipud(s)),1)
        colorspec = np.concatenate((colorspec,np.zeros((len(s),1))),1)
        return colorspec


    ### function for drawing weight history path
    def draw_grads(self,ax,directions,**kwargs):
        # make colors for plot
        colorspec = self.make_colorspec(directions)

        arrows = True
        if 'arrows' in kwargs:
            arrows = kwargs['arrows']

        # plot axes
        ax.axhline(y=0, color='k',zorder = 0,linewidth = 0.5)
        ax.axvline(x=0, color='k',zorder = 0,linewidth = 0.5)

        ### plot function decrease plot in right panel
        for j in range(len(directions)):
            # get current direction
            direction = directions[j]

            # draw arrows connecting pairwise points
            head_length = 0.1
            head_width = 0.1
            ax.arrow(0,0,direction[0],direction[1], head_width=head_width, head_length=head_length, fc='k', ec=
            ax.arrow(0,0,direction[0],direction[1], head_width=0.1, head_length=head_length, fc=colorspec[j], e

    ### function for drawing weight history path
    def draw_grads_v2(self,ax,directions,**kwargs):
        arrows = True
        if 'arrows' in kwargs:
            arrows = kwargs['arrows']

        # plot axes
        ax.axhline(y=0, color='k',zorder = 0,linewidth = 0.5)
        ax.axvline(x=0, color='k',zorder = 0,linewidth = 0.5)

        ### plot function decrease plot in right panel
        head_length = 0.1
        head_width = 0.1
        alpha = 0.1
        for j in range(len(directions)-1):
            # get current direction
            direction = directions[j]

            # draw arrows connecting pairwise points
            ax.arrow(0,0,direction[0],direction[1], head_width=head_width, head_length=head_length, fc='k', ec=
            ax.arrow(0,0,direction[0],direction[1], head_width=0.1, head_length=head_length, fc=self.colorspec[

        # plot most recent direction
        direction = directions[-1]
        num_dirs = len(directions)

        # draw arrows connecting pairwise points
        ax.arrow(0,0,direction[0],direction[1], head_width=head_width, head_length=head_length, fc='k', ec='k',
        ax.arrow(0,0,direction[0],direction[1], head_width=0.1, head_length=head_length, fc=self.colorspec[num_

    ### function for drawing weight history path
    def draw_weight_path(self,ax,w_hist,**kwargs):
        # make colors for plot
        colorspec = self.make_colorspec(w_hist)

        arrows = True
        if 'arrows' in kwargs:
            arrows = kwargs['arrows']

        ### plot function decrease plot in right panel
        for j in range(len(w_hist)):
            w_val = w_hist[j]

            # plot each weight set as a point
            ax.scatter(w_val[0],w_val[1],s = 80,c = colorspec[j],edgecolor = self.edgecolor,linewidth = 2*math.

            # plot connector between points for visualization purposes
            if j > 0:
                pt1 = w_hist[j-1]
                pt2 = w_hist[j]

                # produce scalar for arrow head length
                pt_length = np.linalg.norm(pt1 - pt2)
                head_length = 0.1
                alpha = (head_length - 0.35)/pt_length + 1

                # if points are different draw error
                if np.linalg.norm(pt1 - pt2) > head_length and arrows == True:
                    if np.ndim(pt1) > 1:
                        pt1 = pt1.flatten()
                        pt2 = pt2.flatten()


                    # draw color connectors for visualization
                    w_old = pt1
                    w_new = pt2
                    ax.plot([w_old[0],w_new[0]],[w_old[1],w_new[1]],color = colorspec[j],linewidth = 2,alpha =
                    ax.plot([w_old[0],w_new[0]],[w_old[1],w_new[1]],color = 'k',linewidth = 3,alpha = 1,zorder


                    # draw arrows connecting pairwise points
                    #ax.arrow(pt1[0],pt1[1],(pt2[0] - pt1[0])*alpha,(pt2[1] - pt1[1])*alpha, head_width=0.1, he
                    #ax.arrow(pt1[0],pt1[1],(pt2[0] - pt1[0])*alpha,(pt2[1] - pt1[1])*alpha, head_width=0.1, he

    ### draw surface plot
    def draw_surface(self,g,ax,**kwargs):
        xmin = -3.1
        xmax = 3.1
        ymin = -3.1
        ymax = 3.1
        if 'xmin' in kwargs:
            xmin = kwargs['xmin']
        if 'xmax' in kwargs:
            xmax = kwargs['xmax']
        if 'ymin' in kwargs:
            ymin = kwargs['ymin']
        if 'ymax' in kwargs:
            ymax = kwargs['ymax']

        #### define input space for function and evaluate ####
        w1 = np.linspace(xmin,xmax,200)
        w2 = np.linspace(ymin,ymax,200)
        w1_vals, w2_vals = np.meshgrid(w1,w2)
        w1_vals.shape = (len(w1)**2,1)
        w2_vals.shape = (len(w2)**2,1)
        h = np.concatenate((w1_vals,w2_vals),axis=1)
        func_vals = np.asarray([g(np.reshape(s,(2,1))) for s in h])

        ### plot function as surface ###
        w1_vals.shape = (len(w1),len(w2))
        w2_vals.shape = (len(w1),len(w2))
        func_vals.shape = (len(w1),len(w2))
        ax.plot_surface(w1_vals, w2_vals, func_vals, alpha = 0.1,color = 'w',rstride=25, cstride=25,linewidth=

        # plot z=0 plane
        ax.plot_surface(w1_vals, w2_vals, func_vals*0, alpha = 0.1,color = 'w',zorder = 1,rstride=25, cstride=

        # clean up axis
        ax.xaxis.pane.fill = False
        ax.yaxis.pane.fill = False
        ax.zaxis.pane.fill = False

        ax.xaxis.pane.set_edgecolor('white')
        ax.yaxis.pane.set_edgecolor('white')
        ax.zaxis.pane.set_edgecolor('white')

        ax.xaxis._axinfo["grid"]['color'] =  (1,1,1,0)
        ax.yaxis._axinfo["grid"]['color'] =  (1,1,1,0)
        ax.zaxis._axinfo["grid"]['color'] =  (1,1,1,0)

        ax.set_xlabel('$w_0$',fontsize = 14)
        ax.set_ylabel('$w_1$',fontsize = 14,rotation = 0)
        ax.set_title('$g(w_0,w_1)$',fontsize = 14)


    ### plot points and connectors in input space in 3d plot
    def show_inputspace_path(self,w_hist,ax):
        # make colors for plot
        colorspec = self.make_colorspec(w_hist)

        for k in range(len(w_hist)):
            pt1 = w_hist[k]
            ax.scatter(pt1[0],pt1[1],0,s = 60,c = colorspec[k],edgecolor = 'k',linewidth = 0.5*math.sqrt((1/(fl
            if k < len(w_hist)-1:
                pt2 = w_hist[k+1]
                if np.linalg.norm(pt1 - pt2) > 10**(-3):
                    # draw arrow in left plot
                    a = Arrow3D([pt1[0],pt2[0]], [pt1[1],pt2[1]], [0, 0], mutation_scale=10, lw=2, arrowstyle='
                    ax.add_artist(a)

#### custom 3d arrow and annotator functions ###
# nice arrow maker from https://stackoverflow.com/questions/11140163/python-matplotlib-plotting-a-3d-cube-a-sph
class Arrow3D(FancyArrowPatch):

    def __init__(self, xs, ys, zs, *args, **kwargs):
        FancyArrowPatch.__init__(self, (0, 0), (0, 0), *args, **kwargs)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
        self.set_positions((xs[0], ys[0]), (xs[1], ys[1]))
        FancyArrowPatch.draw(self, renderer)
```

Exercise 1

In [3]:
```python
import autograd.numpy as np
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 gradient module using autograd
    gradient = grad(g)

    # run the gradient descent 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, store current weights and cost function value
        grad_eval = gradient(w)

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

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

1.1

In [4]:
```python
def momentum_accelerated_gradient_descent(g,alpha,max_its,w,beta):
    # compute gradient module using autograd
    gradient = grad(g)

    # run the gradient descent loop
    weight_history = [w]     # container for weight history
    cost_history = [g(w)]
    # container for corresponding cost function history
    for k in range(max_its):
        if(k==0):
        #If we are on the first step, set d to -gradient(w) and take step
            d= -gradient(w)
            w = w + alpha*d
            weight_history.append(w)
            cost_history.append(g(w))
            continue

        #Evalueate previous momentum step
        d = beta*d + (1-beta)*(-gradient(w))
        # take gradient descent step
        w = w + alpha*d

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

1.2

In [27]:
```python
C = np.array([[0.5,0],[0,9.75]])
g = lambda w: (np.dot(np.dot(w.T,C),w))
```

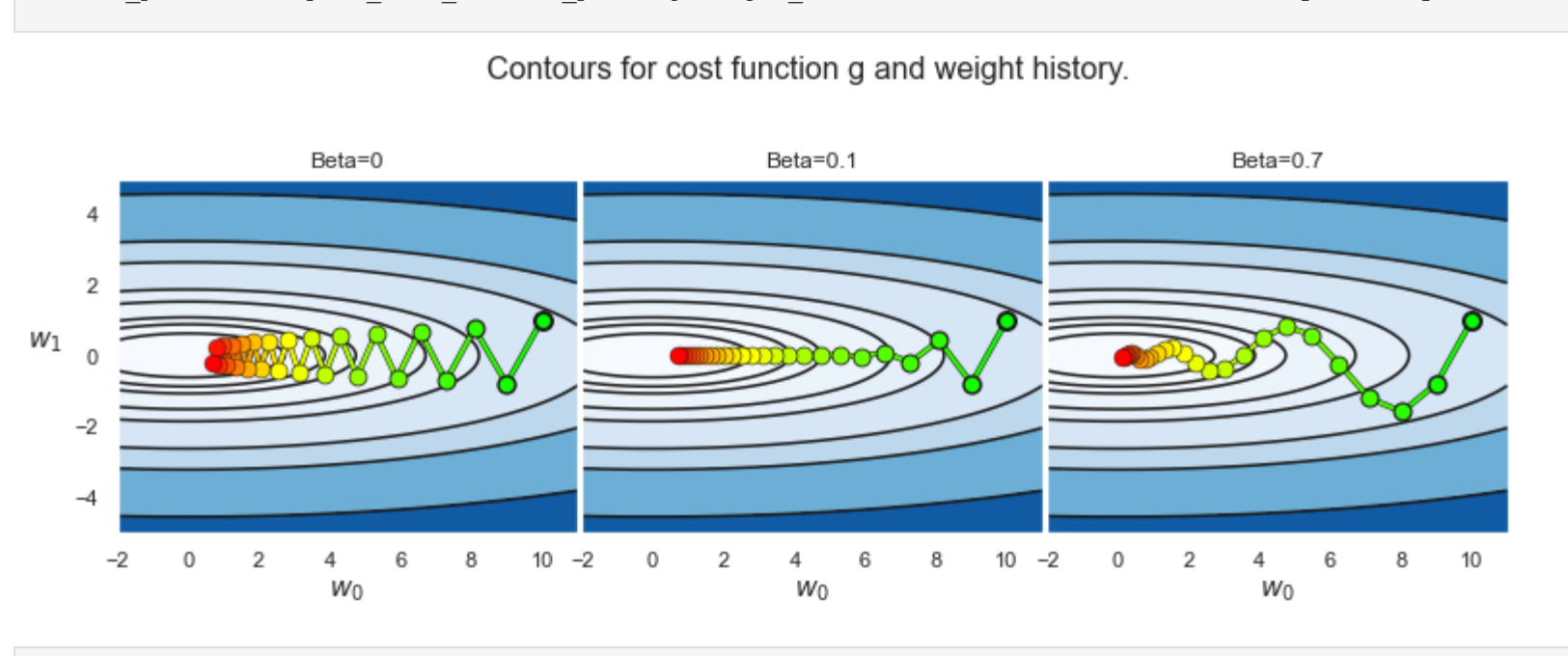
In [28]:
```python
static_plotter=Visualizer();
```

In [29]:
```python
w=np.array([10,1])
alpha=0.1
max_its=25
beta1=0
beta2=0.1
beta3=0.7
```

In [30]:
```python
weight_history_0,cost_history_0=momentum_accelerated_gradient_descent(g,alpha,max_its,w,beta1)
weight_history_0_1,cost_history_0_1=momentum_accelerated_gradient_descent(g,alpha,max_its,w,beta2)
weight_history_0_7,cost_history_0_7=momentum_accelerated_gradient_descent(g,alpha,max_its,w,beta3)
weight_histories=[weight_history_0,weight_history_0_1,weight_history_0_7]
titles=['Beta=0','Beta=0.1','Beta=0.7']
```

In [31]:
```python
#Setting to hide some matplotlib errors generated by plotting
from matplotlib.axes._axes import _log as matplotlib_axes_logger
matplotlib_axes_logger.setLevel('ERROR')
```

In [32]:
```python
#Contour plot using different values for Beta
static_plotter.compare_runs_contour_plots(g,weight_histories,titles,xmin=-2,xmax=11,ymin=-5,ymax=5)
```



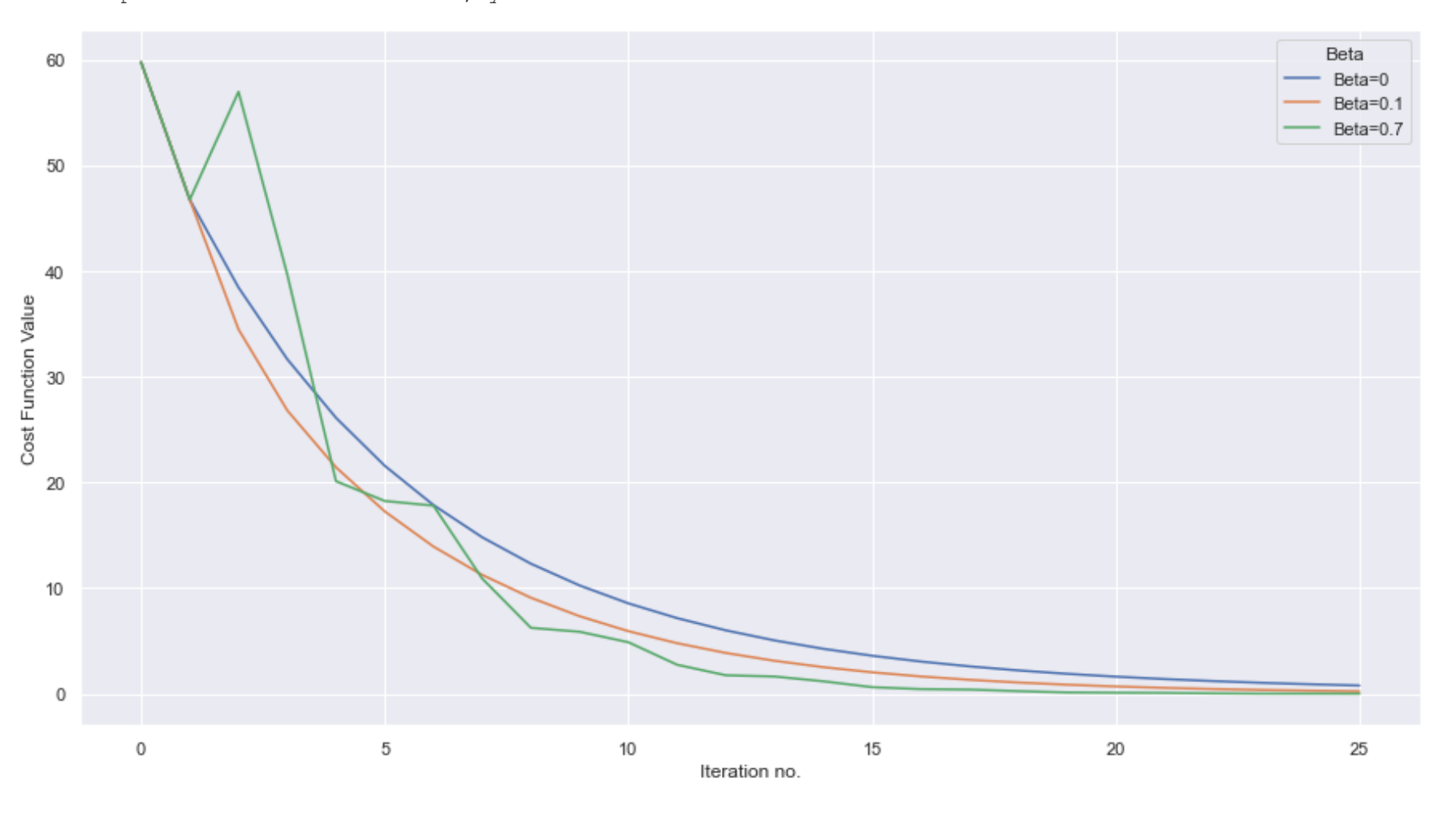
In [33]:
```python
cost_histories=np.append(cost_history_0,[cost_history_0_1,cost_history_0_7])
beta_values=np.repeat(titles,len(cost_history_0))
iter_no=[i for i in range(len(cost_history_0))]*3
cost_history_df=pd.DataFrame([cost_histories,beta_values,iter_no]).T
```

In [34]:
```python
cost_history_df.columns=['Cost Function Value','Beta','Iteration no.']
cost_history_df["Cost Function Value"]=pd.to_numeric(cost_history_df["Cost Function Value"], downcast="float")
cost_history_df["Iteration no."]=pd.to_numeric(cost_history_df["Iteration no."], downcast="float")
```

In [35]:
```python
%matplotlib inline
sns.set(rc = {'figure.figsize':(15,8)}) #Seaborn settings for plotting
sns.set_style("darkgrid")
```

In [14]:
```python
sns.lineplot(data=cost_history_df, x="Iteration no.", y="Cost Function Value", hue="Beta")
```

Out[14]:
```
<AxesSubplot:xlabel='Iteration no.', ylabel='Cost Function Value'>
```



Exercise 2

In [58]:
```python
def slow_crawling_GD(g,alpha,max_its,w,epsilon=1e-4):
    gradient = grad(g)
    # run the gradient descent loop
    weight_history = [w]     # container for weight history
    cost_history = [g(w)]
    # container for corresponding cost function history
    for k in range(max_its):
        #calculate gradient, euclidan norm and step direction
        grad_eval=gradient(w)
        grad_norm=np.linalg.norm(grad_eval)
        step = (alpha/(grad_norm+epsilon))*grad_eval
        w = w-step

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

In [74]:
```python
max(1,0.4*(w[0]**2))
```

Out[74]:
```
40.0
```

In [75]:
```python
np.tanh(4*w[0]+4*w[1])
```

Out[75]:
```
1.0
```

In [69]:
```python
g = lambda w: np.tanh(4*w[0]+4*w[1])+max(1,0.4*(w[0]**2))+1.0
gradient = grad(g)
# run the gradient descent loop
```

In [80]:
```python
w0=np.array([2.0,2.0])
alpha=0.1
max_its=1000
```

In [81]:
```python
gradient(weight_history_standard[-1])
```

Out[81]:
```
array([6.55013204e-12, 6.55013204e-12])
```

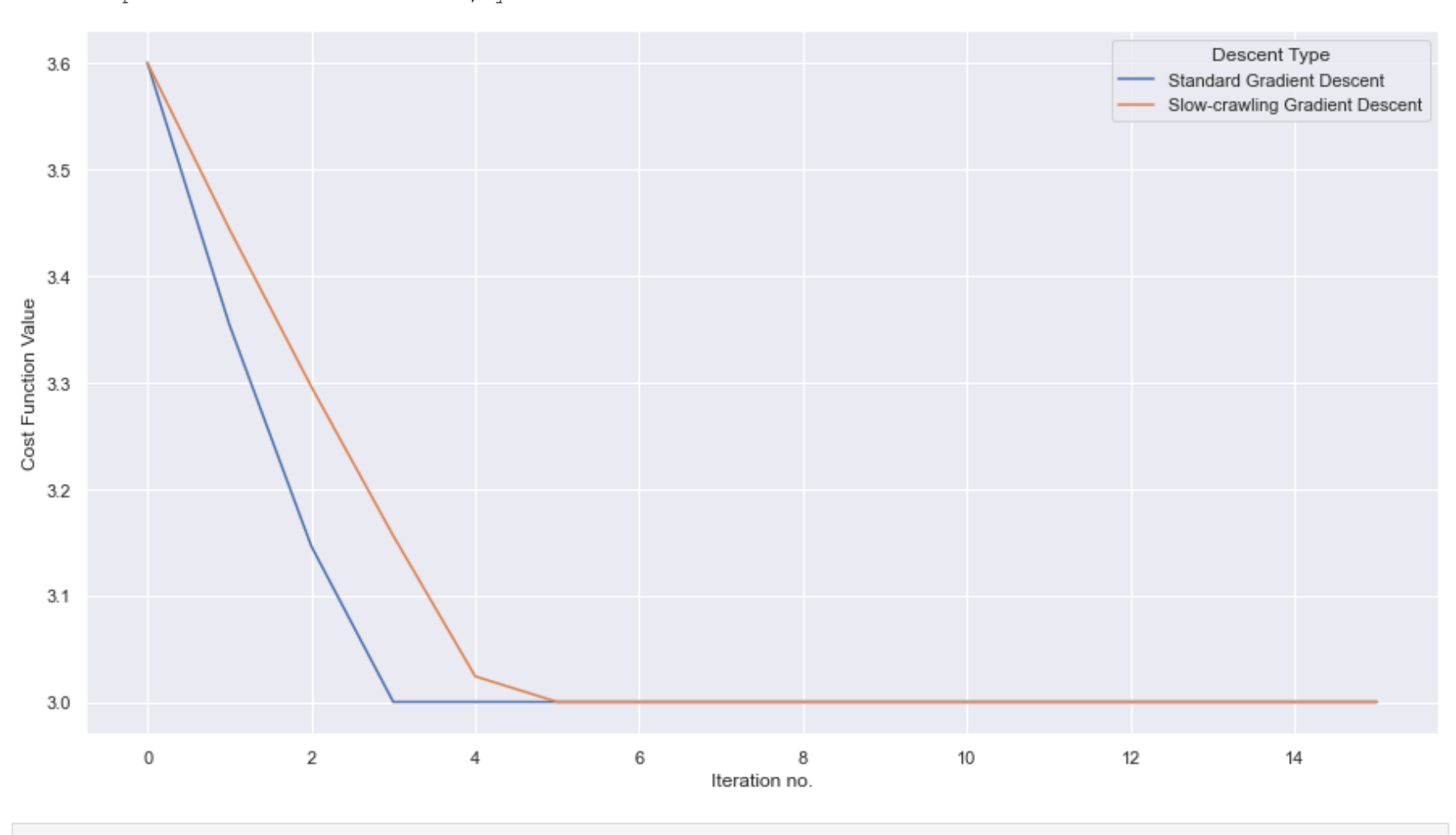
In [82]:
```python
weight_history_standard,cost_history_standard=gradient_descent(g,alpha,max_its,w0)
weight_history_slow,cost_history_slow=slow_crawling_GD(g,alpha,max_its,w0)
```

In [88]:
```python
cost_histories=np.append(cost_history_standard,cost_history_slow)
titles=['Standard Gradient Descent','Slow-crawling Gradient Descent']
beta_values=np.repeat(titles,len(cost_history_standard))
iter_no=[i for i in range(len(cost_history_standard))]*2
cost_history_df=pd.DataFrame([cost_histories,beta_values,iter_no]).T
```

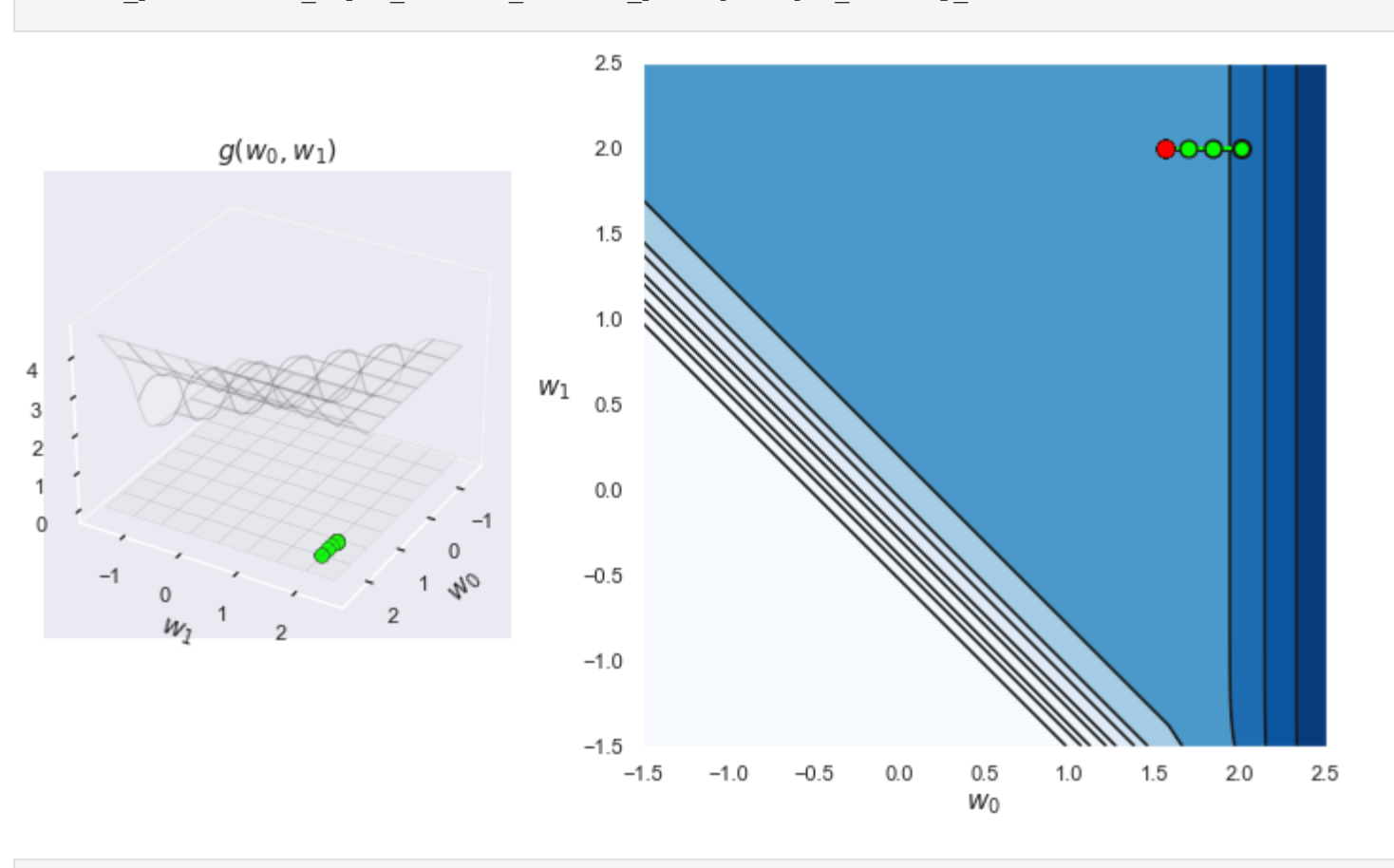
In [93]:
```python
cost_history_df.columns=['Cost Function Value','Descent Type','Iteration no.']
cost_history_df["Cost Function Value"]=pd.to_numeric(cost_history_df["Cost Function Value"], downcast="float")
cost_history_df["Iteration no."]=pd.to_numeric(cost_history_df["Iteration no."], downcast="float")
```

In [97]:
```python
sns.lineplot(data=cost_history_df[cost_history_df['Iteration no.']<16], x="Iteration no.", y="Cost Function Val
```

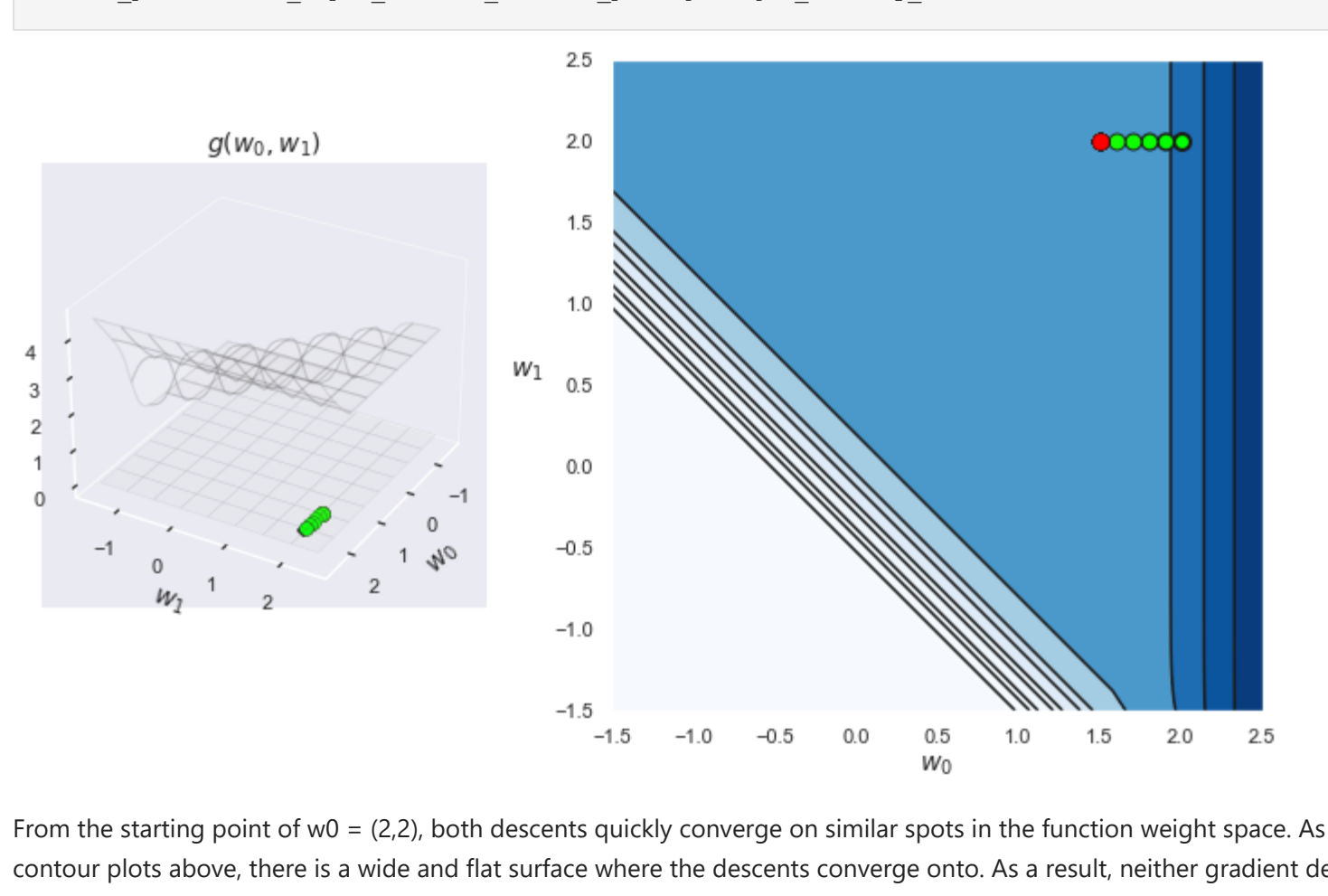
Out[97]:
```
<AxesSubplot:xlabel='Iteration no.', ylabel='Cost Function Value'>
```



In [84]:
```python
static_plotter.two_input_surface_contour_plot(g,weight_history_standard[0:100],view=[30,30],xmin=-1.5,xmax=2.5,
```



In [98]:
```python
static_plotter.two_input_surface_contour_plot(g,weight_history_slow[0:100],view=[30,30],xmin=-1.5,xmax=2.5,ymir
```



From the starting point of w0 = (2,2), both descents quickly converge on similar spots in the function weight space. As shown in the two contour plots above, there is a wide and flat surface where the descents converge onto. As a result, neither gradient descent nor slow-crawling descent can descend onto the valley seen at (-0.5,-0.5). Gradient descent does converge in slightly fewer steps, but with respect to the entire function space and the number of iterations this difference is minimal.

Exercise 3

In [102…
```python
import pandas as pd
# import the dataset
csvname = 'student_debt_data.csv'
data_df=pd.read_csv(csvname,header=None)
data = np.asarray(data_df)
```

In [103…
```python
# extract input
x = data[:,0]
x.shape = (len(x),1)
y = data[:,1]
o = np.ones((len(x),1))
x_new = np.concatenate((o,x),axis = 1)
```

The matrix A should be a 2x2 matrix

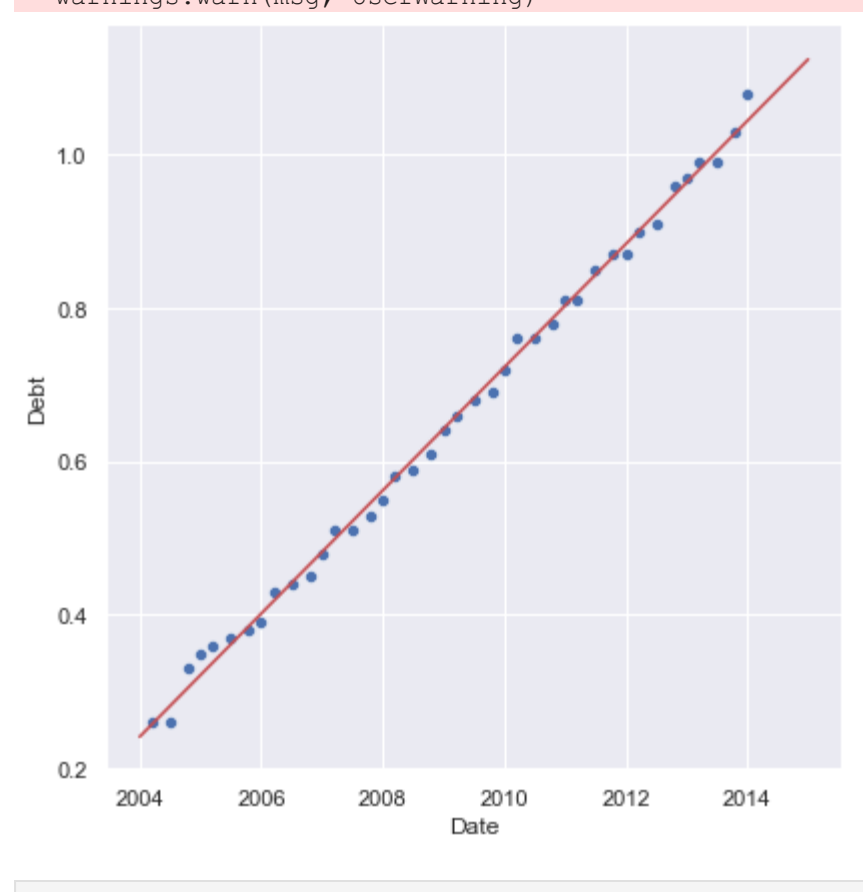
In [104…
```python
A = np.sum([np.matmul(i.reshape(-1,1),i.reshape(1,-1)) for i in x_new],axis=0)
#Compute vector b
b = np.sum(x_new*np.repeat(y.reshape(-1,1),2,axis=1),axis=0)
#Compute w and turn into column vector
w=np.matmul(np.linalg.pinv(A),b)
```

In [105…
```python
data_df.columns=['Date','Debt']
```

In [106…
```python
#Plot points
g = sns.FacetGrid(data_df, size = 6)
g.map(sns.scatterplot, "Date", "Debt")

#Generate line from sample X values
X_plot = np.linspace(2004, 2015, 100)
Y_plot = w[1]*X_plot+w[0]
#Plot line
plt.plot(X_plot, Y_plot, color='r')
plt.show()
```

```
G:\anaconda\lib\site-packages\seaborn\axisgrid.py:316: UserWarning: The `size` parameter has been renamed to `h
eight`; please update your code.
  warnings.warn(msg, UserWarning)
```



In [107…
```python
x_future=np.array([[1.0],[2050.0]])
future_debt=np.matmul(w,x_future)
```

In [108…
```python
print('The predicted debt in 2050 is: '+str(round(future_debt[0],4)))
```

```
The predicted debt in 2050 is: 3.936
```

Exercise 4

In [109…
```python
# import automatic differentiator to compute gradient module
from autograd import grad

# gradient descent function
def gradient_descent(x,y,function,alpha,max_its,w):

    # compute gradient module using autograd
    gradient = grad(g)

    # run the gradient descent loop
    weight_history = [w] # weight history container
    cost_history = [g(w)] # cost function history container
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)
        # take gradient descent step
        w = w - alpha*grad_eval

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

In [110…
```python
# import the dataset. Note you need to take the log of the imported data
csvname = 'kleibers_law_data.csv'
data = np.loadtxt(csvname,delimiter=',')
x = data[0]
x = np.log(x).reshape(1,-1)
x_new=np.append(np.ones(x.shape[1]).reshape(1,-1),x,axis=0)
y = np.log(data[1])
```

In [111…
```python
data_df=pd.read_csv(csvname,header=None).T
data_df=np.log(data_df)
data_df.columns=['Body mass','Metabolic Rate']
```

In [112…
```python
def model(x,w):
    y = np.matmul(w,x)
    return y
```

In [113…
```python
def least_squares(w,x,y):
    y_test=model(x,w)
    cost = np.sum((y_test-y)**2)
    return cost/x.shape[1]
```

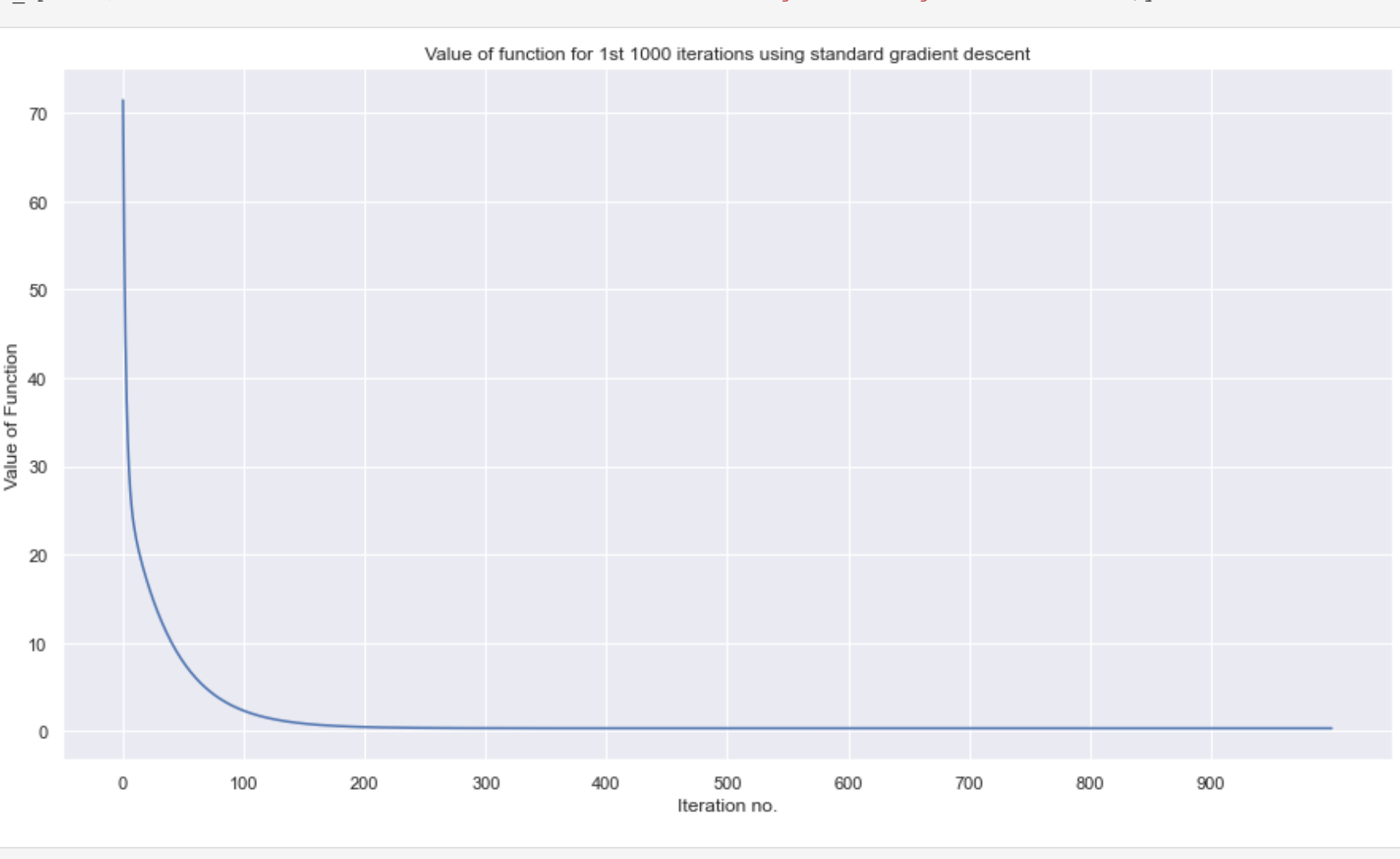
In [114…
```python
g = lambda w: least_squares(w,x_new,y)
```

In [115…
```python
w=np.random.randn(2)
alpha=0.01
max_its=1000
w_history,cost_history=gradient_descent(x_new,y,least_squares,alpha,max_its,w)
```

In [116…
```python
len(cost_history)
```

Out[116… 1001

In [117…
```python
%matplotlib inline
sns.set(rc = {'figure.figsize':(15,8)}) #Seaborn settings for plotting
sns.set_style("darkgrid")
p=sns.lineplot(y=cost_history,x=[i for i in range(1001)])
_=p.set(title="Value of function for 1st 1000 iterations using standard gradient descent",ylabel="Value of Func
```



In [118…
```python
w_final=w_history[-1].reshape(1,-1)
x_test=np.array([[1],[np.log(10)]])
num_kJ=np.exp(model(x_test,w_final))[0][0]
num_calories=num_kJ/4.18
```

In [119…
```python
print('The number of calories needed for an animal weighing 10 kg is about '+str(round(num_calories,2))+' calor
```

The number of calories needed for an animal weighing 10 kg is about 980.01 calories.