Table of Contents

In [1]:
```
# load in basic libraries
from autograd import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sys
sys.path.append('../')

# imports from custom library
from mlrefined_libraries import basics_library as baslib
from mlrefined_libraries import calculus_library as calib
from mlrefined_libraries import math_optimization_library as optlib
from mlrefined_libraries import superlearn_library as superlearn

# demos for this notebook
regress_plotter = superlearn.lin_regression_demos
optimizers = optlib.optimizers
static_plotter = optlib.static_plotter.Visualizer();
datapath = '../mlrefined_datasets/superlearn_datasets/'
plotter = superlearn.multi_outupt_plotters

# this is needed to compensate for matplotlib notebook's tendancy to blow up imag
es when plotted inline
%matplotlib notebook
from matplotlib import rcParams
rcParams['figure.autolayout'] = True
```

Exercise 5.1. Fitting a regression line to the student debt data

Load up the dataset.

In [2]:
```
# import the dataset
csvname = datapath + 'student_debt_data.csv'
data = np.asarray(pd.read_csv(csvname,header = None))

# extract input
x = data[:,0]
x.shape = (len(x),1)

# pad input with ones
o = np.ones((len(x),1))
x_new = np.concatenate((o,x),axis = 1)

# extract output and re-shape
y = data[:,1]
y.shape = (len(y),1)
```

Lets setup the linear system associated to minimizing the Least Squares cost function for this problem and solve it.

In [3]:
```
# solve linear system of equations for regression fit
A = np.dot(x_new.T,x_new)
b = np.dot(x_new.T,y)
w = np.dot(np.linalg.pinv(A),b)
```

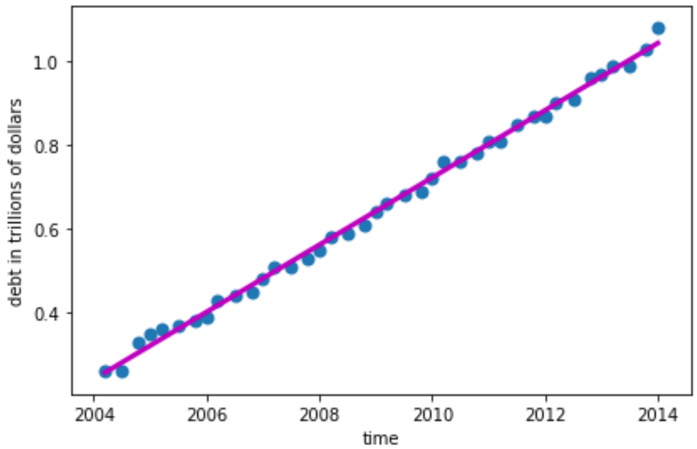
With our line fit to the data we can now predict total student debt in 2050.

In [4]:
```
# print out predicted amount of student debt in 2050
debt_in_2050 = w[0] + w[1]*2050
print ('if this linear trend continues there will be ' + str(debt_in_2050[0]) + ' trillion dollars in student debt in 2050!')
```

```
if this linear trend continues there will be 3.936011966600745 trillion dollars
in student debt in 2050!
```

Finally, lets print out the dataset and linear fit.

In [5]:
```
# plot data with linear fit - this is optional
s = np.linspace(np.min(x),np.max(x))
t = w[0] + w[1]*s

figure = plt.figure()
plt.plot(s,t,linewidth = 3,color = 'm')
plt.scatter(x,y,linewidth = 2)
plt.xlabel('time')
plt.ylabel('debt in trillions of dollars')
plt.show()
```



Predicted debt in 2020

Exercise 5.2. Kleiber's law and linear regression

In [6]:
```
# import the dataset
csvname = datapath + 'kleibers_law_data.csv'
data = np.loadtxt(csvname,delimiter=',')
x = data[:-1,:]
y = data[-1:,:]

# log-transform data
x = np.log(x)
y = np.log(y)
```

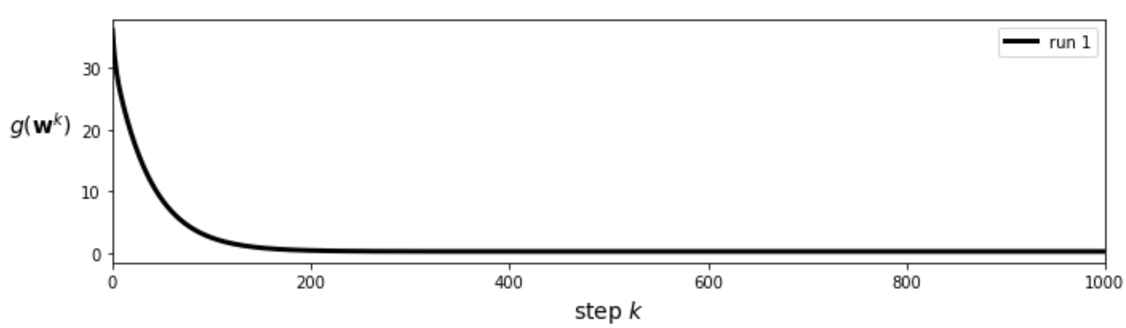
In [7]:
```
# import automatic differentiator to compute gradient module
from autograd import grad

# gradient descent function
def gradient_descent(g,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 [8]:
```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

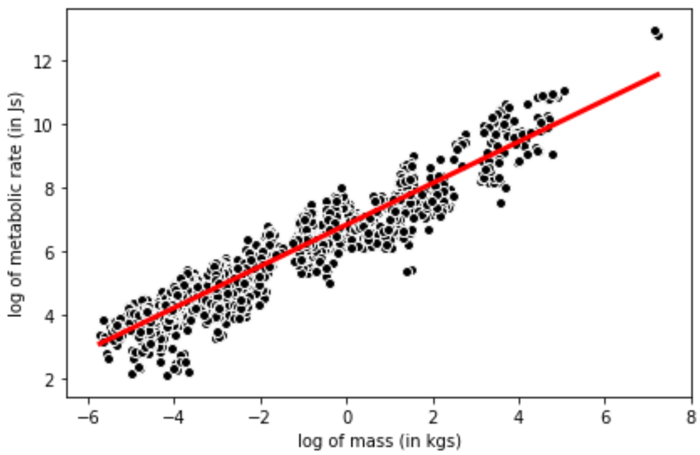
In [9]:
```
# an implementation of the least squares cost function for linear regression
def least_squares(w):
    cost = np.sum((model(x,w) - y)**2)
    return cost/float(np.size(y))
```

In [10]:
```
# run gradient descent to minimize the Least Squares cost for linear regression
g = least_squares; w = 0.1*np.random.randn(2,1); max_its = 1000; alpha_choice = 1
0**(-2);
weight_history, cost_history = gradient_descent(g,alpha_choice,max_its,w)
```

In [11]:
```
static_plotter.plot_cost_histories([cost_history],start = 0,points = False,labels
= ['run 1'])
```



In [12]:
```
# plot data with linear fit - this is optional
s = np.linspace(np.min(x),np.max(x))
w = weight_history[-1]
t = w[0] + w[1]*s

figure = plt.figure()
plt.plot(s,t,linewidth = 3,color = 'r')
plt.scatter(x,y,linewidth = 1,c='k',edgecolor='w')
plt.xlabel('log of mass (in kgs)')
plt.ylabel('log of metabolic rate (in Js)')
plt.show()
```



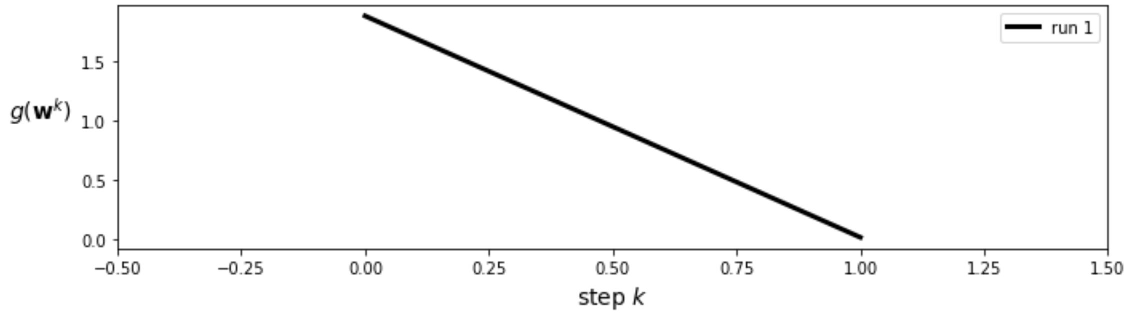
Exercise 5.3. The Least Squares cost function and a single Newton step

In [14]:
```
# using an automatic differentiator - like the one imported via the statement bel
ow - makes coding up gradient descent a breeze
from autograd import grad
from autograd import hessian

# newtons method function - inputs: g (input function), max_its (maximum number o
f 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**(-10)
    if 'epsilon' in kwargs:
        epsilon = kwargs['epsilon']

    # run the newtons method loop
    weight_history = [w]           # container for weight history
    cost_history = [g(w)]          # container for corresponding cost function hi
story
    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_eva
l))**(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
```

In [20]:
```
# load in data
csvname = datapath + '3d_linregress_data.csv'
data = np.loadtxt(csvname,delimiter=',')
x = data[:-1,:]
y = data[-1:,:]
```

In [22]:
```
# run gradient descent to minimize the Least Squares cost for linear regression
g = least_squares; w = 0.1*np.random.randn(3,1); max_its = 1; alpha_choice = 1
0**(-2);
weight_history, cost_history = newtons_method(g,max_its,w)
```

In [23]:
```
static_plotter.plot_cost_histories([cost_history],start = 0,points = False,labels
= ['run 1'])
```



Exercise 5.4. Solving the normal equations

Newton steps are computationally far more expensive than first-order steps (e.g., gradient descent), requiring the storage and computation of not just a gradient but an entire Hessian matrix of second derivative information. This can become an issue as the input dimension N increases. For very large values of N it can be more computationally efficient to take multiple gradient steps (instead of a single Newton step).

Exercise 5.5. Lipschitz constant for the Least Squares cost

Computing the Hessian of Least Squares we have

$$\nabla^2 g\left(\mathbf{w}\right) = \frac{2}{P}\sum_{p=1}^{P}\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T$$

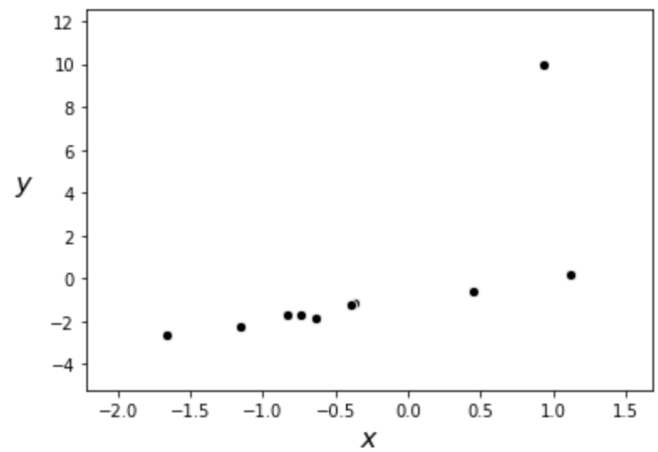
then denoting by $\|\cdot\|_2$ the maximum eigenvalue of an iput matrix, the Lipschitz constant for the Least Squares cost is

$$L = \frac{2}{P}\left\|\sum_{p=1}^{P}\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\right\|_2$$

Exercise 5.6. Compare the Least Squares and Least Absolute Deviation costs

Load in dataset.

In [26]:
```
# load in dataset
data = np.loadtxt(datapath + 'regression_outliers.csv',delimiter = ',')
x = data[:-1,:]
y = data[-1:,:]

# plot dataset
demo = regress_plotter.Visualizer(data)
demo.plot_data()
```



Define Least Squares and Least Absolute Deviations.

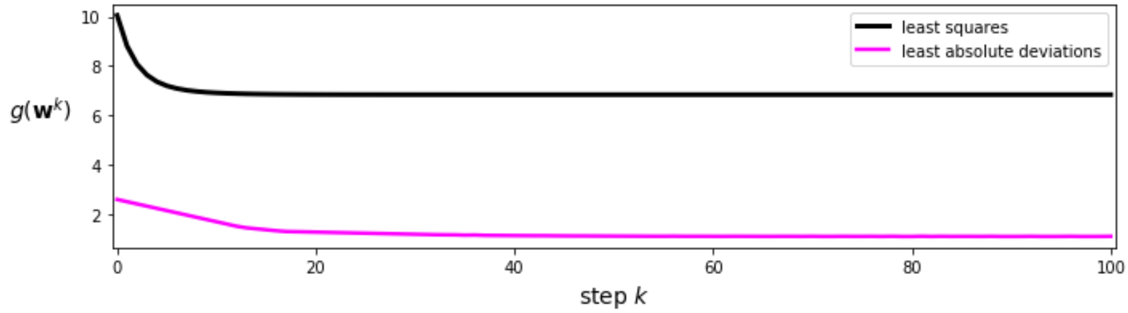
In [27]:
```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

In [28]:
```
# an implementation of the least squares cost function for linear regression
def least_squares(w):
    cost = np.sum((model(x,w) - y)**2)
    return cost/float(np.size(y))
```

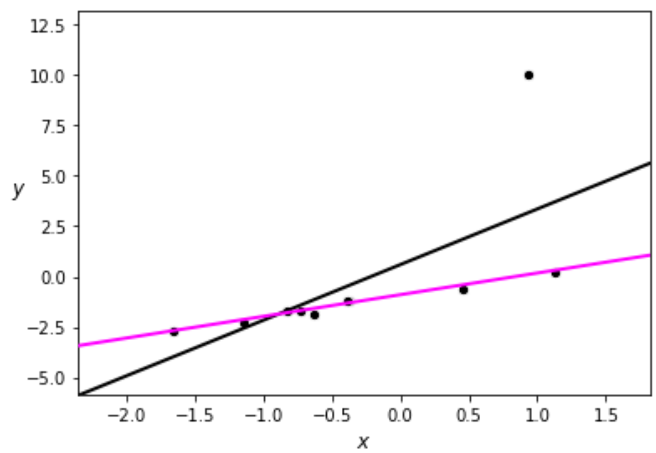
In [29]:
```
# a compact least absolute deviations cost function
def least_absolute_deviations(w):
    cost = np.sum(np.abs(model(x,w) - y))
    return cost/float(np.size(y))
```

In [30]:
```
# run gradient descent to minimize the Least Squares cost for linear regression
g = least_squares; w = np.array([1.0,1.0])[:,np.newaxis]; max_its = 100; alpha_ch
oice = 10**(-1);
weight_history_1,cost_history_1 = optimizers.gradient_descent(g,alpha_choice,max_
its,w)

# run gradient descent to minimize the Least Squares cost for linear regression
g = least_absolute_deviations; w = np.array([1.0,1.0])[:,np.newaxis]; max_its = 1
00; alpha_choice = 10**(-1);
weight_history_2,cost_history_2 = optimizers.gradient_descent(g,alpha_choice,max_
its,w)
```

In [31]:
```
## This code cell will not be shown in the HTML version of this notebook
# plot the cost function history for a given run
static_plotter.plot_cost_histories([cost_history_1,cost_history_2],start = 0,poin
ts = False,labels = ['least squares','least absolute deviations'])
```



In [32]:
```
# find best set of weights from gradient descent run on Least Absolute Deviations
cost
ind = np.argmin(cost_history_1)
least_weights = weight_history_1[ind]
ind = np.argmin(cost_history_2)
absolute_weights = weight_history_2[ind]
demo.plot_fit(plotting_weights = [least_weights,absolute_weights])
```

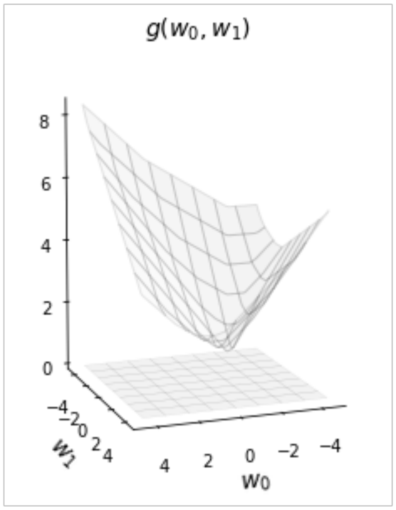


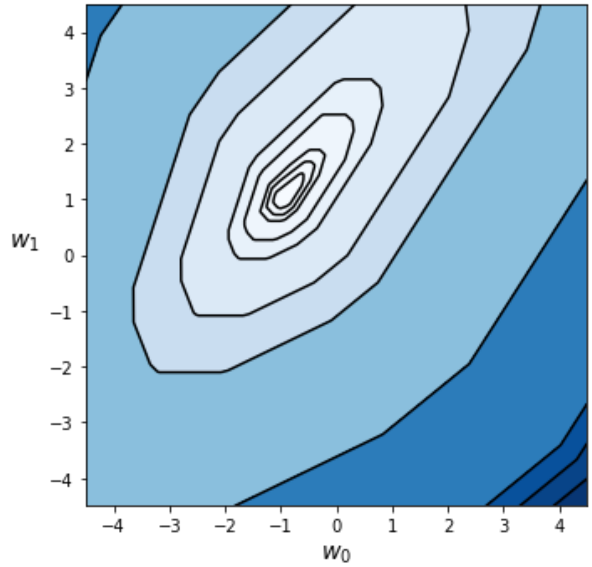
Exercise 5.7. Empirically confirm convexity for a toy dataset

In [33]:
```
# a compact least absolute deviations cost function
def least_absolute_deviations(w):
    cost = np.sum(np.abs(model(x,w) - y))
    return cost/float(np.size(y))
```

Below we plot the surface / contour plot of this cost function using the previously shown dataset - indeed it is convex.

In [34]:
```
## This code cell will not be shown in the HTML version of this notebook
# show run in both three-dimensions and just the input space via the contour plot
static_plotter.two_input_surface_contour_plot(least_absolute_deviations,[],view =
[10,70],xmin = -4.5, xmax = 4.5, ymin = -4.5, ymax = 4.5,num_contours = 20)
```





Exercise 5.8. The Least Absolute Deviations cost is convex

In order to show that the Least Absolute Deviations cost

$$g(\mathbf{w}) = \sum_p |\mathbf{x}_p^T\mathbf{w} - y_p|$$

is convex, it suffices to prove that each summand is convex. Proving convexity of the sum of two convex functions is trivial and left to the reader.

Isolating the pth summand

$$g_p(\mathbf{w}) = |\mathbf{x}_p^T\mathbf{w} - y_p|$$

we can write

$$\begin{aligned}\lambda g_p(\mathbf{w}_1) + (1-\lambda)g_p(\mathbf{w}_2) &= \lambda|\mathbf{x}_p^T\mathbf{w}_1 - y_p| + (1-\lambda)|\mathbf{x}_p^T\mathbf{w}_2 - y_p| \\ &= |\mathbf{x}_p^T\lambda\mathbf{w}_1 - \lambda y_p| + |\mathbf{x}_p^T(1-\lambda)\mathbf{w}_2 - (1-\lambda)y_p|\end{aligned}$$

Now, using the triangle inequality $|\alpha| + |\beta| \geq |\alpha + \beta|$ we may find a lowerbound on the equality above as

$$\begin{aligned}|\mathbf{x}_p^T\lambda\mathbf{w}_1 - \lambda y_p| + |\mathbf{x}_p^T(1-\lambda)\mathbf{w}_2 - (1-\lambda)y_p| &\geq |\mathbf{x}_p^T\lambda\mathbf{w}_1 - \lambda y_p + \mathbf{x}_p^T(1-\lambda)\mathbf{w}_2 - (1-\lambda)y_p| \\ &= |\mathbf{x}_p^T(\lambda\mathbf{w}_1 + (1-\lambda)\mathbf{w}_2) - y_p| = g_p(\lambda\mathbf{w}_1 + (1-\lambda)\mathbf{w}_2)\end{aligned}$$

Exercise 5.9. Housing price and Automobile Miles-per-Gallon prediction

In [35]:
```
# standard normalization function - with nan checker / filler in-er
def standard_normalizer(x):
    # compute the mean and standard deviation of the input
    x_means = np.nanmean(x,axis = 1)[:,np.newaxis]
    x_stds = np.nanstd(x,axis = 1)[:,np.newaxis]

    # check to make sure thta x_stds > small threshold, for those not
    # divide by 1 instead of original standard deviation
    ind = np.argwhere(x_stds < 10**(-2))
    if len(ind) > 0:
        ind = [v[0] for v in ind]
        adjust = np.zeros((x_stds.shape))
        adjust[ind] = 1.0
        x_stds += adjust

    # fill in any nan values with means
    ind = np.argwhere(np.isnan(x) == True)
    for i in ind:
        x[i[0],i[1]] = x_means[i[0]]

    # create standard normalizer function
    normalizer = lambda data: (data - x_means)/x_stds

    # create inverse standard normalizer
    inverse_normalizer = lambda data: data*x_stds + x_means

    # return normalizer
    return normalizer,inverse_normalizer
```

Boston housing

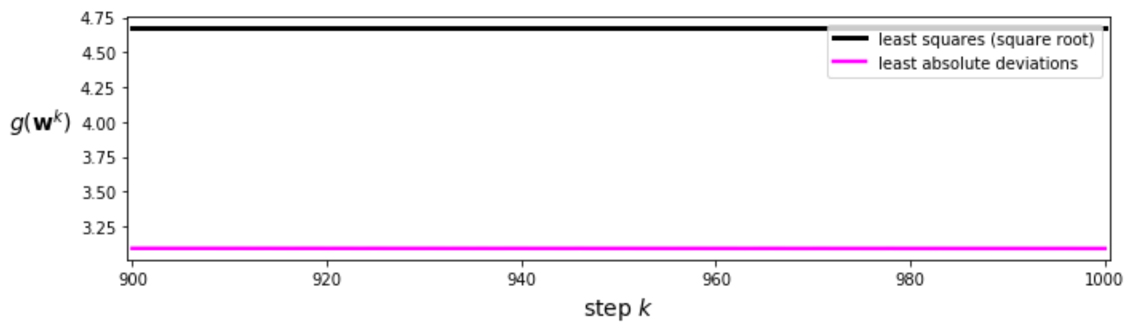
In [36]:
```
# import the dataset
csvname =  datapath + 'boston_housing.csv'
data = np.loadtxt(csvname,delimiter=',')
x = data[:-1,:]
y = data[-1:,:]
```

In [37]:
```
# standard normalize input
normalizer,inverse_normalizer = standard_normalizer(x)
x = normalizer(x)
```

In [38]:
```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T

# an implementation of the least squares cost function for linear regression
def least_squares(w):
    cost = np.sum((model(x,w) - y)**2)
    return cost/float(np.size(y))

# a compact least absolute deviations cost function
def least_absolute_deviations(w):
    cost = np.sum(np.abs(model(x,w) - y))
    return cost/float(np.size(y))
```

In [39]:
```
# run gradient descent to minimize the Least Squares cost for linear regression
g = least_squares; w = 0.1*np.random.randn(x.shape[0]+1,1); max_its = 1000; alpha
_choice = 10**(-1);
weight_history_1,cost_history_1 = optimizers.gradient_descent(g,alpha_choice,max_
its,w)
cost_history_1 = [c**(0.5) for c in cost_history_1]

# run gradient descent to minimize the Least Squares cost for linear regression
g = least_absolute_deviations;
weight_history_2,cost_history_2 = optimizers.gradient_descent(g,alpha_choice,max_
its,w)
```

In [40]:
```
## This code cell will not be shown in the HTML version of this notebook
# plot the cost function history for a given run
static_plotter.plot_cost_histories([cost_history_1,cost_history_2],start = 900,po
ints = False,labels = ['least squares (square root)','least absolute deviations
'])
```



In [41]:
```
print(cost_history_1[-1])
print(cost_history_2[-1])
```

```
[4.6795063]
[3.08963844]
```

Auto-MPG dataset

Load up the dataset.

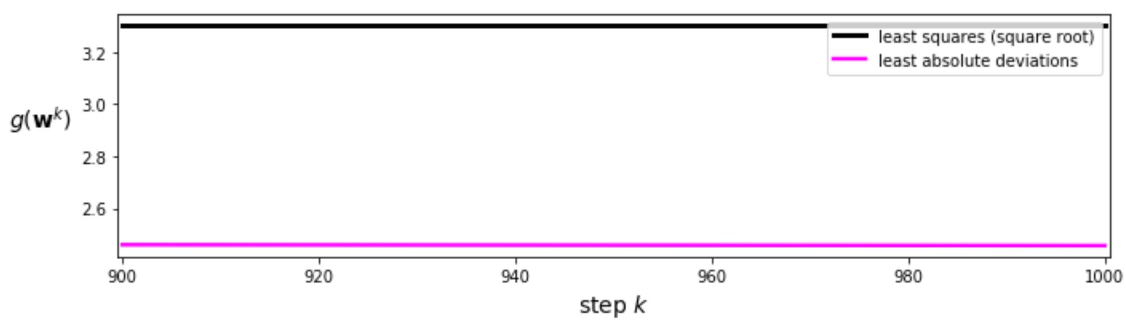
In [42]:
```
# import the dataset
csvname =  datapath + 'auto_data.csv'
data = np.loadtxt(csvname,delimiter=',')
x = data[:-1,:]
y = data[-1:,:]
```

In [43]:
```
# standard normalize input
normalizer,inverse_normalizer = standard_normalizer(x)
x = normalizer(x)
```

In [44]:
```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T

# an implementation of the least squares cost function for linear regression
def least_squares(w):
    cost = np.sum((model(x,w) - y)**2)
    return cost/float(np.size(y))

# a compact least absolute deviations cost function
def least_absolute_deviations(w):
    cost = np.sum(np.abs(model(x,w) - y))
    return cost/float(np.size(y))
```

In [45]:
```
# run gradient descent to minimize the Least Squares cost for linear regression
g = least_squares; w = 0.1*np.random.randn(x.shape[0]+1,1); max_its = 1000; alpha
_choice = 10**(-1);
weight_history_1,cost_history_1 = optimizers.gradient_descent(g,alpha_choice,max_
its,w)
cost_history_1 = [c**(0.5) for c in cost_history_1]

# run gradient descent to minimize the Least Squares cost for linear regression
g = least_absolute_deviations;
weight_history_2,cost_history_2 = optimizers.gradient_descent(g,alpha_choice,max_
its,w)
```

In [46]:
```
## This code cell will not be shown in the HTML version of this notebook
# plot the cost function history for a given run
static_plotter.plot_cost_histories([cost_history_1,cost_history_2],start = 900,po
ints = False,labels = ['least squares (square root)','least absolute deviations
'])
```



In [47]:
```
print(cost_history_1[-1])
print(cost_history_2[-1])
```

```
[3.30356516]
[2.4573954]
```

Exercise 5.10. Improper tuning and weighted regression

Remember the Least Squares cost function is *nonnegative*, with global minima value of 0.

In minimizing the pointwise weights β_p note that whenever setting these values to zero, the cost function value is zero too (and hence a global minima). This means that we can set $\mathbf{w}$ to any finite value we choose.

Thus if we were to try to minimize the pointwise weights with $\mathbf{w}$, we are defeating the purpose of tuning the weights of our linear regressor, since they could be set arbitrarily given that $\beta_p = 0$ for all p always provides a global minima.

Exercise 5.11. Multi-output regression

In [48]:
```
# compute linear combination of input points
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

`Pythonic` implementation of regression cost functions can also be implemented precisely as we have seen previously. For example, the Least Squares cost can be written as shown below.

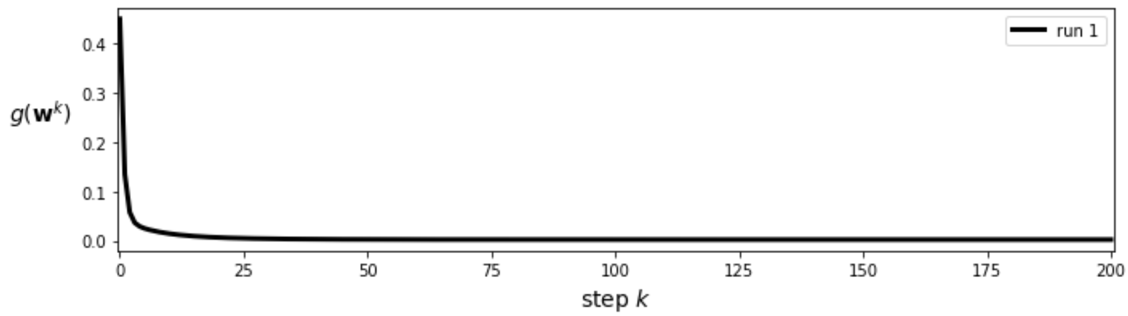
In [49]:
```
# an implementation of the least squares cost function for linear regression
def least_squares(w):
    # compute the least squares cost
    cost = np.sum((model(x,w) - y)**2)
    return cost/float(np.size(y))
```

In this example we show an example of multi-output linear regression using a toy dataset with input dimension $N = 2$ and output dimension $C = 2$. The dataset is shown below.

In [50]:
```
# load in data
csvname = datapath + 'linear_2output_regression.csv'
data = np.loadtxt(csvname,delimiter=',')
x = data[:2,:]
y = data[2:,:]

# plot
view1 = [20,40]; view2 = [20,40]
plotter.plot_data(x,y,view1=view1,view2=view2)
```

We use the `Pythonic` Least Squares function shown above, minimizing it using 200 steps of gradient descent using a steplength / learning rate $\alpha = 1$. The cost function history from this run is shown below.

In [51]:
```
# setup and run optimization
g = least_squares;
w = 0.1*np.random.randn(3,2)
max_its = 200;
alpha_choice = 1;
weight_history,cost_history = optimizers.gradient_descent(g,alpha_choice,max_its,
w)

# plot history
static_plotter.plot_cost_histories([cost_history],start = 0,points = False,labels
= ['run 1'])
```



Using the fully trained model from this run of gradient descent (implemented as shown in Section 8.4.3) we evaluate a fine mesh of points in the region over which the input of this dataset is defined to visualize our linear approximations. These are shown in light green in the panels below.

In [52]:
```
## This code cell will not be shown in the HTML version of this notebook
# determine best weights - based on lowest cost value attained
ind = np.argmin(cost_history)
w_best = weight_history[ind]

# form predictor
predictor = lambda x: model(x,w_best)

# plot data with predictions
view1 = [20,40]; view2 = [20,40]
plotter.plot_regressions(x,y,predictor,view1=view1,view2=view2)
```