```
In [1]: import sympy as sym
        import numpy as np
        import matplotlib.pyplot as plt

        # import autograd ’s automatic differentiator
        from autograd import grad
        from autograd import hessian

        # datapath to data
        datapath = '/home/michaelrencheck/EE475/machine_learning_refined-gh-p
        ages/mlrefined_exercises/ed_2/mlrefined_datasets/superlearn_dataset
        s/'

        sym.init_printing()
```

Please complete exercises 4.1, 4.2, 4.5, in Chapter 4 and 5.2, 5.9 in Chapter 5 in your textbook.

Problem 4.1

a)

$$z^T C z \geq 0$$

The eiganvalue decomposition of C:

$$C = VDV^T$$

Substituting back in:

$$z^T VDV^T z \geq 0$$

We know that:

$$z^T VV^T z > 0$$

Therefore the beginning statement is true only if all the values in D are nonnegative.

b)

$$z^T C z \geq 0$$

We know that:

$$z^T z \geq 0$$

So in order for the statement to be true all values of C are nonnegative.

The eigenvalue decomposition of C is:

$C = VDV^T$ and $C \geq 0$

We know that:

$$VV^T > 0$$

Therefore in order for the original statement to hold all of the eigenvalues in D must be nonnegative.

c)

The second-order definition of convexity states that if the Hessian is semi-positive definite, then the function is convex.

Plotting the function $g(textbf w)$ reveals that this is a convex function, therefore the assumption is that the Hessian is semi-positive definite.

The Hessian of the function is:

$$\nabla^2 g(\mathbf{w}) = C$$

If C is semi-positive definite, then its eigenvalues must all be nonnegative.

```
In [6]: C = np.array([[1,1],[1,1]])

        print("The eigenvalues of C are: ")
        print(np.linalg.eig(C)[0])

        The eigenvalues of C are: 
        [ 2.  0.]
```

After performing the eigenvalue decompostion of C, we can see that all of the eigenvalues are, in fact, nonnegative, meaning the Hessian is semi-positive definite.

d)

If the eigenvalues of C are not all positive, it means that C is not greater than 0.

In order to shift this, C can be modified by adding $I\lambda$ such that C becomes greater than or equal to zero. This will then result in all nonnegative eigen values.

The smallest λ that can be used to satisfy this is the absolute value of the minimum element in C.

Problem 4.2

a)

The outer product matrix xx^T will result in a square, symetric matrix M. If v is an eigenvector to M, then the following is true:

$$Mv = \lambda v$$
$$x^T * xx^Tv = \lambda x^Tv$$
$$x^T * x = \lambda$$
$$[x_1^2, x_2^2, \cdots, x_N^2] = \lambda$$

Since a squared number is always greater than or equal to zero, all of the eigenvalues are nonnegative.

b)

Using the proof in a), each square matrix $x_p x_p^T$ is shown to have all nonnegative eigenvalues. Multiplying this matrix by a nonnegative scaling factor cannot change that fact that each individual matrix will still maintain a nonnegative eigenvalues.

Since all matricies from 0 to P have nonnegative eigenvalues, the sum of all P matrices also must have nonnegative eigenvalues.

c)

Using the proof in b) shows the summation will result in nonnegative eigenvalues. If this matrix to increased by λI this will increase every value along the matrix diagonal by λ.

The diagonal of the matrix is equal:

$$\left[\sum_1^P x_{1,p}^2, \sum_1^P x_{2,p}^2, \cdots, \sum_1^P x_{N,}^2\right]$$

By observation this is x^Tx, which in a) is shown to be equal to the eigenvales. Adding a scalar value λ that is greater than 0 will result in all eigenvalues being only positive.

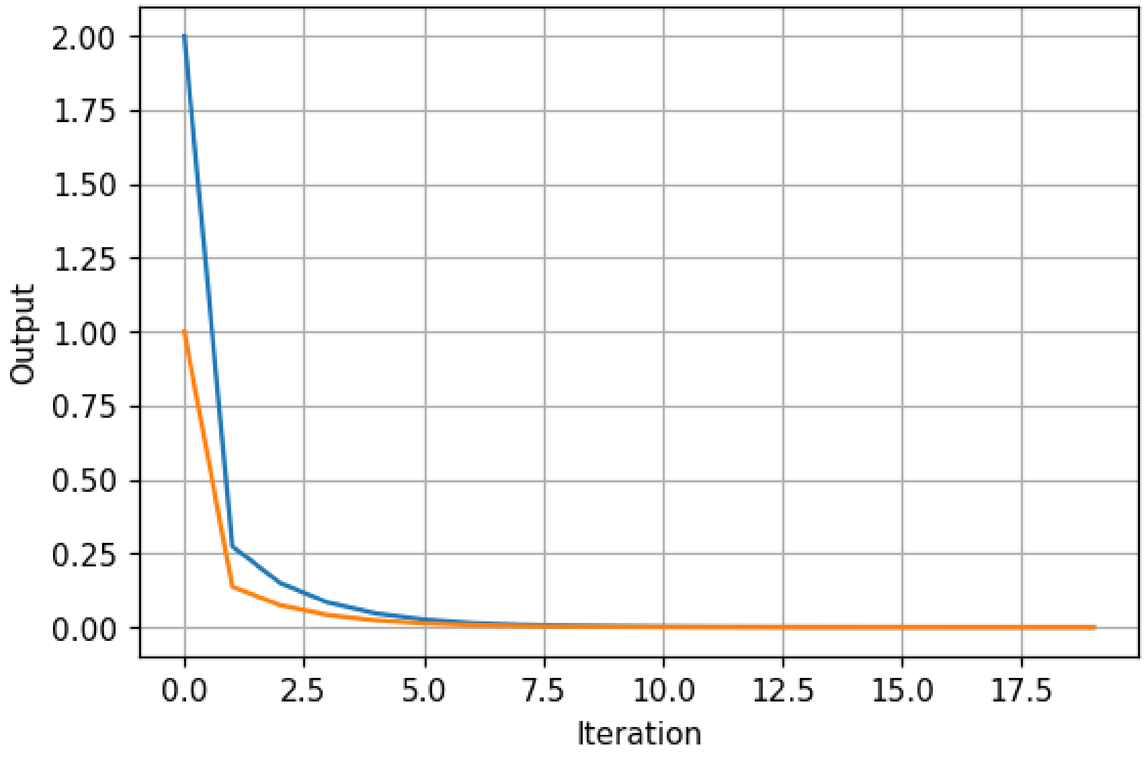
Problem 4.5

Setup

In [7]:
```
w1, w2 = sym.symbols('w_1 w_2')

w = sym.Matrix([[w1],[w2]])

f = sym.log(1 + sym.exp(w.T * w)[0], 10)

gradf = sym.Matrix([f]).jacobian(w)

hessf = sym.hessian(f, w)
hessf_inv = hessf.inv()

f_lam = sym.lambdify([w1, w2], f)
gradf_lam = sym.lambdify([w1, w2], gradf)
hessf_lam = sym.lambdify([w1, w2], hessf)
hessf_inv_lam = sym.lambdify([w1, w2], hessf_inv)
```

a)

In [73]:
```
n = 20 # iterations
a = 1 # learning rate

w_init = np.array([2,1])  # initial guess

w = np.ones([2,n])

w[0,:] = w[0,:]*w_init[0]
w[1,:] = w[1,:]*w_init[1]

for j in range(n-1):
    df = gradf_lam(w[0][j], w[1][j])

    w[:,j+1] = w[:,j] - a * df[0]

xaxis = np.arange(n)

fig1 = plt.figure(dpi=110,facecolor='w')
plt.grid(True)
plt.plot(xaxis, w[0])
plt.plot(xaxis, w[1])
plt.xlabel("Iteration")
plt.ylabel("Output")
plt.show()

print("The stationary point is at ")
print(w[:,-1])
print("Resulting in a g(w): ")
print(f_lam(w[0,-1], w[1,-1]))
```



```
The stationary point is at
[  9.16802365e-06   4.58401182e-06]
Resulting in a g(w):
0.301029995687
```

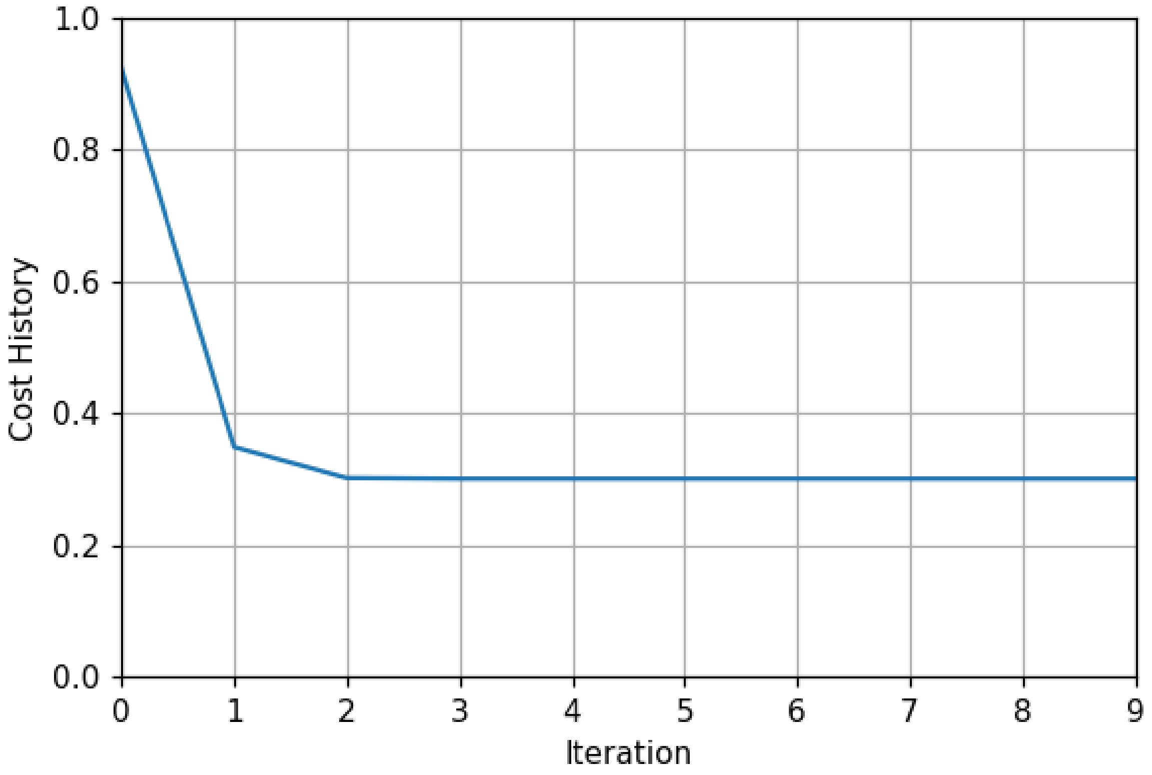
b)

In [17]:
```
hess_eval = hessf_lam(0,0)

print("The eigenvalues for the Hessian evaluated at the point found i
n a) are both positive: ", np.linalg.eig(hess_eval)[0])

print("This confirms this function is convex.")
```

```
The eigenvalues for the Hessian evaluated at the point found in a) ar
e both positive:  [ 0.43429448  0.43429448]
This confirms this function is convex.
```

c)

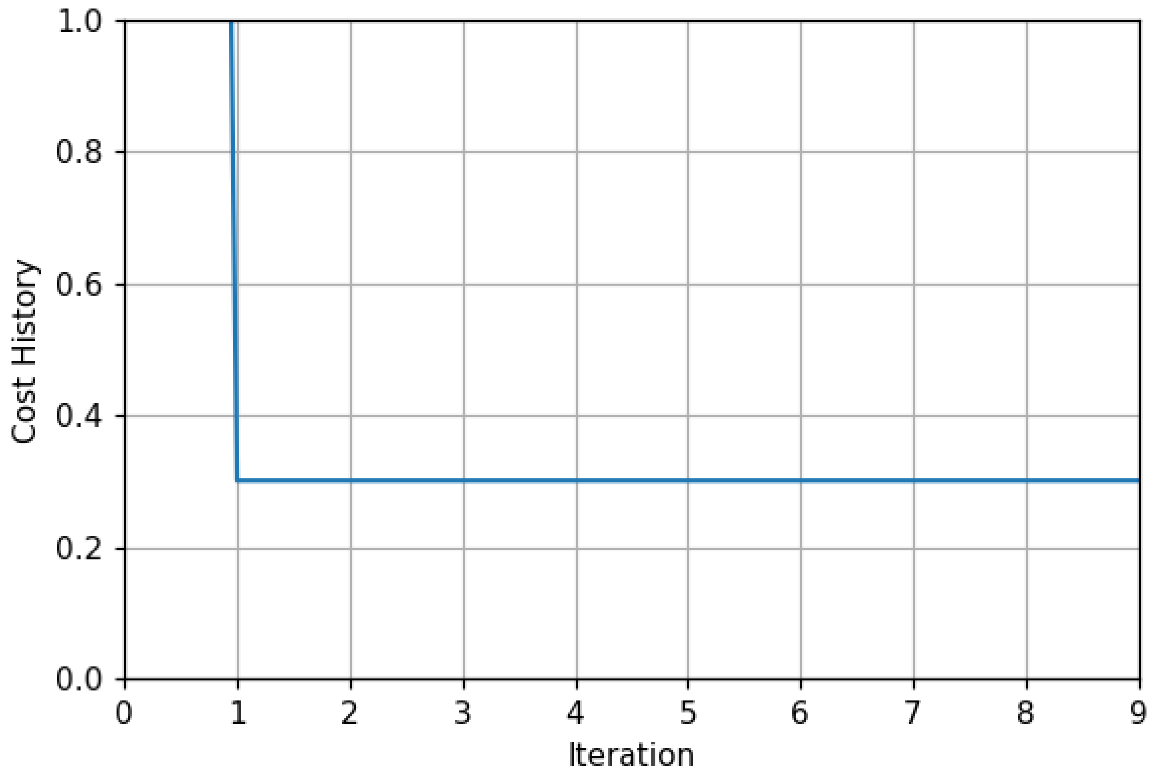
In [98]:
```
n = 10  # iterations

w_init = np.array([1,1])  # initial guess

w = np.ones([2,n])

w[0,:] = w[0,:]*w_init[0]
w[1,:] = w[1,:]*w_init[1]

cost_history = np.zeros(n)

cost_history[0] = f_lam(w_init[0],w_init[1])

for j in range(1,n):
    df = gradf_lam(w[0][j-1], w[1][j-1])
    ddf = hessf_inv_lam(w[0][j-1], w[1][j-1])

    df = np.reshape(df[0], [2,1])

    w[:,j] = w[:,j-1] - np.squeeze(np.matmul(ddf, df))

    cost_history[j] = f_lam(w[0,j],w[1,j])

xaxis = np.arange(n)

# fig1 = plt.figure(dpi=110,facecolor='w')
# plt.grid(True)
# plt.plot(xaxis, w[0])
# plt.plot(xaxis, w[1])
# plt.xlim([0,n-1])
# plt.xlabel("Iteration")
# plt.ylabel("Output")
# plt.legend(['$w_1$','$w_2$'])

fig1 = plt.figure(dpi=110,facecolor='w')
plt.grid(True)
plt.plot(xaxis, cost_history)
plt.xlabel("Iteration")
plt.ylim([0,1])
plt.xlim([0,n-1])
plt.ylabel("Cost History")

plt.show()

print("The stationary point is at ")
print(w[:,-1])
print("Resulting in a g(w): ")
print(f_lam(w[0,-1], w[1,-1]))
```



```
The stationary point is at
[ 0.  0.]
Resulting in a g(w):
0.301029995664
```

d)

In [99]:
```
n = 10 # iterations

w_init = np.array([4,4])  # initial guess

w = np.ones([2,n])

w[0,:] = w[0,:]*w_init[0]
w[1,:] = w[1,:]*w_init[1]

cost_history = np.zeros(n)

cost_history[0] = f_lam(w_init[0],w_init[1])

for j in range(1,n):
    df = gradf_lam(w[0][j-1], w[1][j-1])
    ddf = hessf_inv_lam(w[0][j-1], w[1][j-1])

    df = np.reshape(df[0], [2,1])

    w[:,j] = w[:,j-1] - np.squeeze(np.matmul(ddf, df))

    cost_history[j] = f_lam(w[0,j],w[1,j])

xaxis = np.arange(n)

# fig1 = plt.figure(dpi=110,facecolor='w')
# plt.grid(True)
# plt.plot(xaxis, w[0])
# plt.plot(xaxis, w[1])
# plt.xlim([0,n-1])
# plt.legend(['$w_1$','$w_2$'])
# plt.xlabel("Iteration")
# plt.ylabel("Output")

fig1 = plt.figure(dpi=110,facecolor='w')
plt.grid(True)
plt.plot(xaxis, cost_history)
plt.xlabel("Iteration")
plt.ylim([0,1])
plt.xlim([0,n-1])
plt.ylabel("Cost History")

plt.show()

print("The stationary point is at ")
print(w[:,-1])
print("Resulting in a g(w): ")
print(f_lam(w[0,-1], w[1,-1]))
```



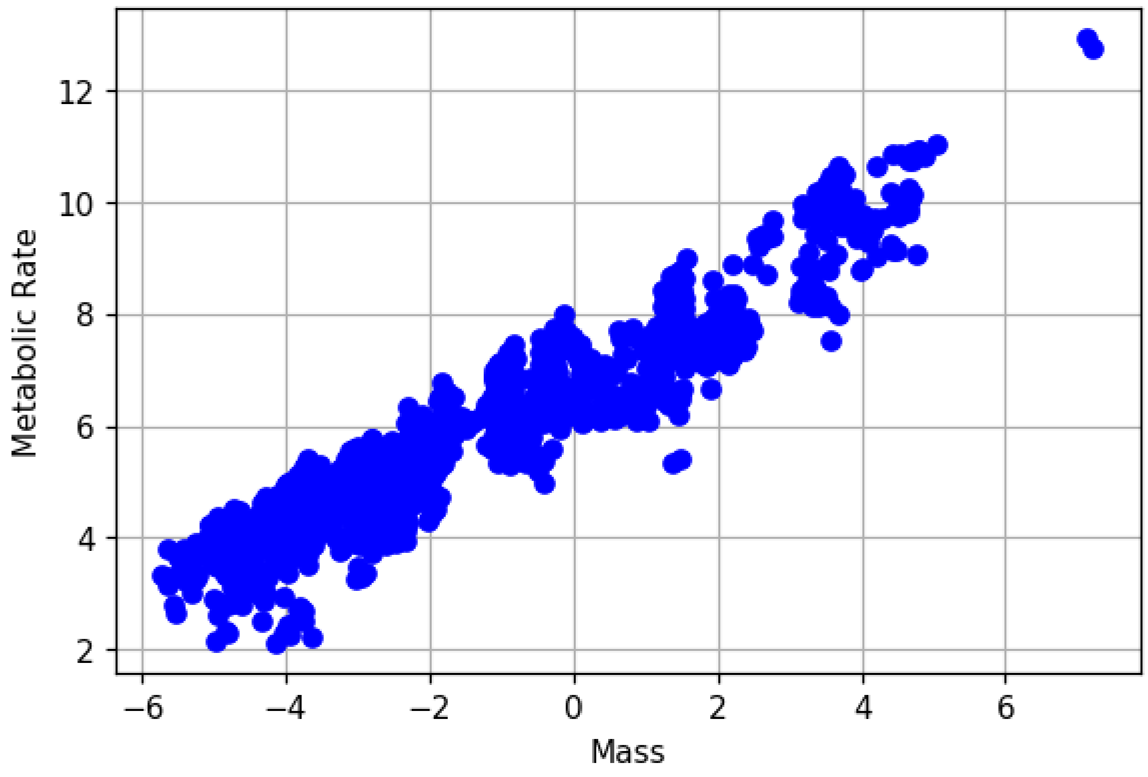
```
The stationary point is at
[ 0.  0.]
Resulting in a g(w):
0.301029995664
```

Since Newton's method is using the second derivative, we are fitting a parabola to the cost function. Meaning the error between the the fit and the function will grow exponetially as we diverge from the stationary point. Therefore, the exponential increase allows the algorithm to take exponentially larger steps toward the stationary point the farther away the input point is.

Problem 5.2

```
In [4]: # import the dataset
        csvname = datapath + 'kleibers_law_data.csv'
        data = np.loadtxt(csvname,delimiter=',')

        x = data[:-1,:]
        y = data[-1:,:]

        x_log = np.log(x)
        y_log = np.log(y)

        print(np.shape(x))
        print(np.shape(y))

        fig1 = plt.figure(dpi=110,facecolor='w')
        plt.grid(True)
        plt.plot(np.log(x),np.log(y), "bo")
        plt.xlabel("Mass")
        plt.ylabel("Metabolic Rate")

        plt.show()
```

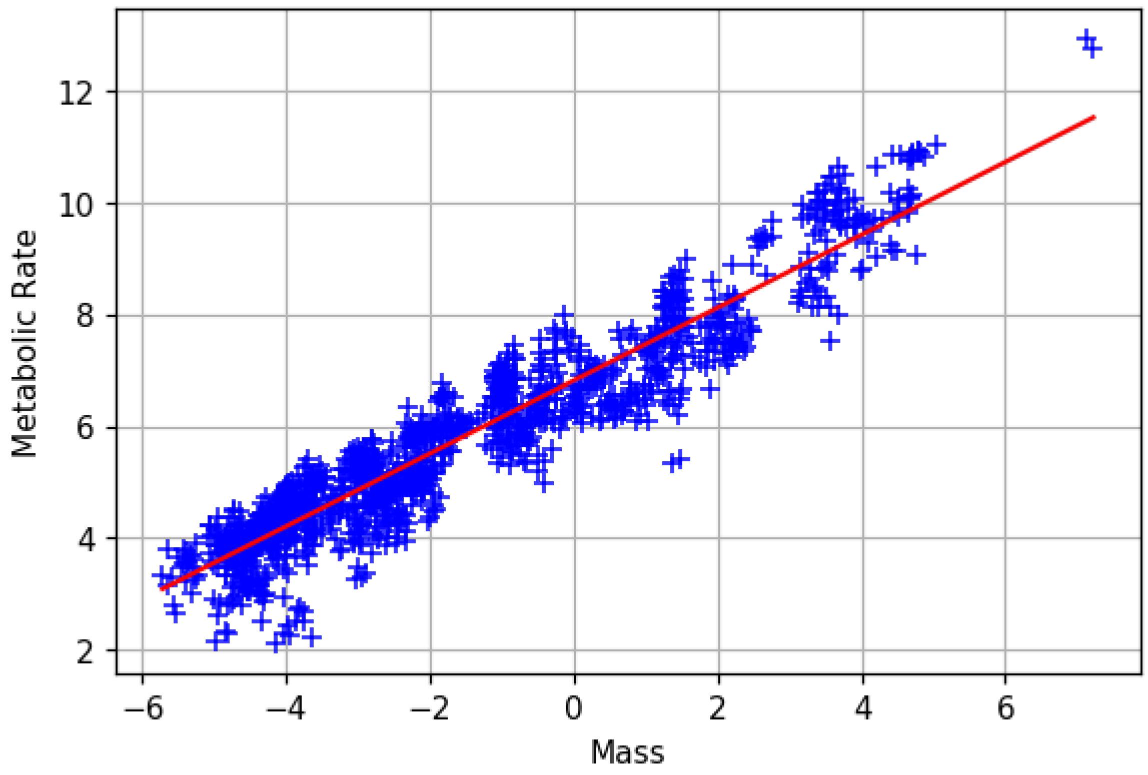
```
(1, 1498)
(1, 1498)
```



a)

```
In [5]: def model(x_p ,w):
        # compute linear combination and return
            a = w[0] + np .dot(x_p.T,w[1:])
            return a.T

        # a least squares function for linear regression
        def least_squares (w,x,y):
        # loop over points and compute cost contribution from each input/outp
        ut pair
            cost = 0
            for p in range(y.size):
                # get pth input/ output pair
                x_p = x[:,p][:, np.newaxis ]
                y_p = y[p]

                ## add to current cost
                cost += (model(x_p, w) - y_p)**2

            # return average least squares error
            return cost / float(y. size)



        w = np.ones([2,1])

        x_log = np.squeeze(np.log(x))
        y_log = np.squeeze(np.log(y))

        A = np.zeros([2,2])
        b = np.zeros([2,1])

        for i, xval in enumerate(x_log):

            # append 1 to the data set
            xp = np.reshape(np.array([1, xval]), [2,1])
            A += np.matmul(xp, xp.transpose())
            b += xp * y_log[i]

        Ainv = np.linalg.inv(A)

        # Newtons Method Update
        #  w = w - Ainv * (A * w - b)
        w = np.matmul(Ainv, b)

        print(w)

        xaxis = np.linspace(min(x_log),max(x_log))
        yaxis = w[1] * xaxis + w[0]

        fig1 = plt.figure(dpi=110,facecolor='w')
        plt.grid(True)
        plt.plot(x_log,y_log, "b+")
        plt.plot(xaxis,yaxis, 'r')
        plt.xlabel("Mass")
        plt.ylabel("Metabolic Rate")
plt.show()
```

```
[[6.81473477]
 [ 0.6528121 ]]
```



b)

The nonlinear model is:

$$\log(y) = 0.6528121 * \log(x) + 6.81473477$$
$$y = x^{0.6528121} * e^{6.81473477}$$

c)

```
In [7]: x_in = 10 # kg

        y_out = np.exp(w[1] * np.log(x_in) + w[0])

        y_out = (1000 * y_out) / 4.18

        print("A 10kg animal requires ", y_out[0], " calories/day")
```

```
A 10kg animal requires  980025.833806  calories/day
```

Problem 5.9

```
In [65]: # Standard Normalize the data
         def std_normalize(in_arr):
             u = np.mean(in_arr, axis=1, dtype=np.float64)
             sig = np.std(in_arr, axis=1, dtype=np.float64)

             out = np.zeros(in_arr.shape)

             for i, row in enumerate(in_arr):
                 for j, element in enumerate(row):
                     out[i,j] = (element - u[i])/ sig[i]

             return np.squeeze(out)

         def fit_data(x,y):
             # Create a linear fit for the data set
             w = np.ones([x.shape[0],1])

             A = np.zeros([x.shape[0]+1, x.shape[0]+1])
             b = np.zeros([x.shape[0]+1, 1])

             for i in range(x.shape[1]):
                 # append 1 to the data set
                 xp = np.concatenate((np.array([1]), x[:,i]))
                 xp = np.reshape(xp, [xp.shape[0],1])

                 A += np.dot(xp, xp.transpose())
                 b += xp * y[:,i]

             Ainv = np.linalg.inv(A)

             # Newtons Method Update
             #  w = w - Ainv * (A * w - b)

             w = np.dot(Ainv, b)

             # Find the RMSE and MAD Metrics

             P = x.shape[1]

             MAD = 0
             RMSE = 0
             for i in range(P):
                 xp = np.concatenate((np.array([1]), x[:,i]))
                 xp = np.reshape(xp, [1,xp.shape[0]])

                 diff = np.matmul(xp,w) - y[0,i]

                 MAD += abs(diff)
                 RMSE += (diff)**2

             MAD *= (1/P)
             RMSE *= (1/P)
             RMSE = np.sqrt(RMSE)

             return {"w": w, "rmse": np.squeeze(RMSE), "mad": np.squeeze(MAD)}
```

Boston Housing

In [66]:
```
# import the dataset
csvname =  datapath + 'boston_housing.csv'
data = np.loadtxt(csvname,delimiter=',')
x = data[:-1,:]
y = data[-1:,:]

# print(np.shape(x))
# print(np.shape(y))

x_norm = std_normalize(x)
sol = fit_data(x_norm,y)

print("The RMSE is ", sol["rmse"]*1000, "and the MAD is ", sol["mad"]
*1000)
```

```
The RMSE is  4679.50630064 and the MAD is  3272.944638
```

Auto Milage

In [67]:
```
# import the dataset
csvname =  datapath + 'auto_data.csv'
data = np.loadtxt(csvname,delimiter=',')
x = data[:-1,:]
y = data[-1:,:]

# print(np.shape(x))
# print(np.shape(y))

x_norm = std_normalize(x)
sol = fit_data(x_norm,y)

print("The RMSE is ", sol["rmse"], "and the MAD is ", sol["mad"])
```

```
The RMSE is  3.2935514183022025 and the MAD is  2.4993098325008707
```