Supervised Regression with Multiple Variables

0. Import library

In [3]:

```
# Import libraries

# math library
import numpy as np

# visualization library
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png2x','pdf')
import matplotlib.pyplot as plt

# machine learning library
from sklearn.linear_model import LinearRegression

# 3d visualization
from mpl_toolkits.mplot3d import axes3d

# computational time
import time
```

1. Load dataset

Load a set of data points $\{(x_i, y_i, z_i)\}_{i=1}^{n}$ where x_i and y_i are considered as an input and z_i is considered as an output for i-th data point.

In [15]:

```
# import data with numpy
data_clean = np.genfromtxt('data_clean.txt', delimiter=',')
data_noisy = np.genfromtxt('data_noisy.txt', delimiter=',')

data_clean = data_clean[:,0:3]  # do not change it
data_noisy = data_noisy[:,0:3]  # do not change it

# number of training data
n_clean = len(data_clean)
n_noisy = len(data_noisy)
print("Number of clean data =", n_clean)
print("Number of noisy data =", n_noisy)
print("Size of clean data =", data_clean.shape)
print("Type of clean data =", data_clean.dtype)
print("Size of noisy data =", data_noisy.shape)
print("Type of noisy data =", data_clean.dtype)
```

```
Number of clean data = 3600
Number of noisy data = 3600
Size of clean data = (3600, 3)
Type of clean data = float64
Size of noisy data = (3600, 3)
Type of noisy data = float64
```

2. Explore the dataset distribution

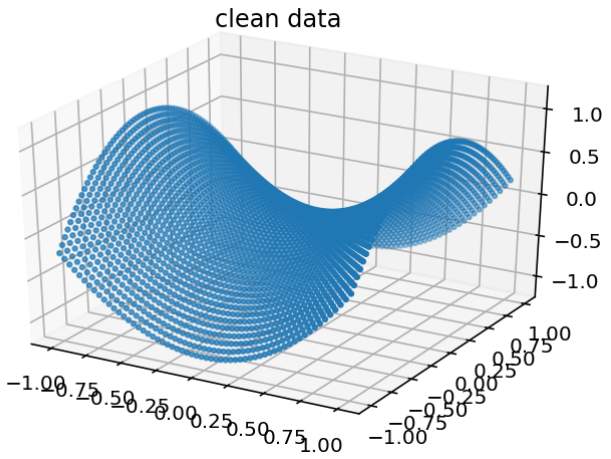
Plot the training data points in 3D cartesian coordinate system. (You may use matplotlib function `scatter3D()` .)

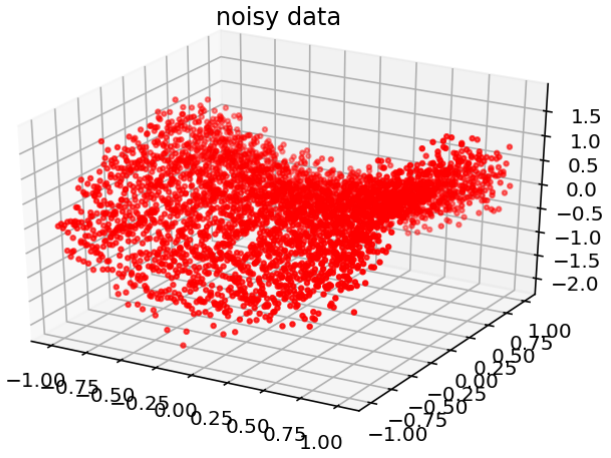
In [48]:

```
x_clean = data_clean[:,0]
y_clean = data_clean[:,1]
z_clean = data_clean[:,2]

x_train = data_noisy[:,0]
y_train = data_noisy[:,1]
z_train = data_noisy[:,2]

fig1 = plt.figure()
fig2 = plt.figure()
ax1 = fig1.gca(projection='3d')
ax2 = fig2.gca(projection='3d')

# Customize the z axis.
ax1.scatter(x_clean, y_clean, z_clean, marker='.')
ax2.scatter(x_train, y_train, z_train, marker='.', color='r')

ax1.set_title('clean data')
ax2.set_title('noisy data')
```

Out[48]:

```
Text(0.5, 0.92, 'noisy data')
```





3. Define the prediction function

$$f_w(x,y) = w_0 f_0(x,y) + w_1 f_1(x,y) + w_2 f_2(x,y) + w_3 f_3(x,y) + w_4 f_4(x,y) + w_5 f_5(x,y) + w_6 f_6(x,y) + w_7 f_7(x,y) + w_8 f_8(x,y) + w_9 f_9(x,y)$$

where feature function f is defined as follows:

$$\begin{aligned} f_0(x,y) &= \\ f_1(x,y) &= \\ f_2(x,y) &= \\ f_3(x,y) &= \\ f_4(x,y) &= \\ f_5(x,y) &= \\ f_6(x,y) &= \\ f_7(x,y) &= \\ f_8(x,y) &= \\ f_9(x,y) &= \end{aligned}$$

Vectorized implementation:

$$f_w(x) = Xw$$

with

$$X = \begin{bmatrix} f_0(x_1,y_1) & f_1(x_1,y_1) & f_2(x_1,y_1) & f_3(x_1,y_1) & f_4(x_1,y_1) & f_5(x_1,y_1) & f_6(x_1,y_1) & f_7(x_1,y_1) \\ f_0(x_2,y_2) & f_1(x_2,y_2) & f_2(x_2,y_2) & f_3(x_2,y_2) & f_4(x_2,y_2) & f_5(x_2,y_2) & f_6(x_2,y_2) & f_7(x_2,y_2) \\ \vdots & & & & & & & \\ f_0(x_n,y_n) & f_1(x_n,y_n) & f_2(x_n,y_n) & f_3(x_n,y_n) & f_4(x_n,y_n) & f_5(x_n,y_n) & f_6(x_n,y_n) & f_7(x_n,y_n) \end{bmatrix}$$

$$\text{and} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \\ w_9 \end{bmatrix}$$

Implement the vectorized version of the predictive function.

In [93]:

```
# construct data matrix

n = data_noisy.shape[0]
X = np.ones([n,10])

X[:,1] = x_train
X[:,2] = y_train
X[:,3] = x_train + y_train
X[:,4] = x_train * x_train
X[:,5] = y_train * y_train
X[:,6] = (x_train * x_train) + y_train
X[:,7] = x_train + (y_train * y_train)
X[:,8] = x_train * x_train * x_train
X[:,9] = y_train * y_train * y_train

print(X.shape)

# parameters vector
w = np.array([1,1,1,1,1,1,1,1,1,1])[:,None] # [:,None] adds a singleton dimension

# predictive function definition
def f_pred(X,w):

    f = X @ w

    return f

# Test predicitive function
z_pred = f_pred(X,w)
```

(3600, 10)

4. Define the regression loss

$$L(w) = \frac{1}{n}\sum_{i=1}^{n}\left(f_w(x_i, y_i) - z_i\right)^2$$

Vectorized implementation:

$$L(w) = \frac{1}{n}(Xw - z)^T(Xw - z)$$

with

$$Xw = \begin{bmatrix} f_w(x_1, y_1) \\ f_w(x_2, y_2) \\ \vdots \\ f_w(x_n, y_n) \end{bmatrix} \quad \text{and} \quad z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}$$

Implement the vectorized version of the regression loss function.

In [94]:

```
# loss function definition
def loss_mse(z_pred,z):

    loss = (z_pred - z).T @ (z_pred - z) / n

    return loss


# Test loss function
z = z_train # label
z_pred = f_pred(X, w).reshape(3600,) # prediction

loss = loss_mse(z_pred,z)
print((z_pred-z).shape)
print(loss.shape)
print(loss)
```

```
(3600,)
()
15.41576230085457
```

5. Define the gradient of the regression loss

• Vectorized implementation: Given the loss

$$L(w) = \frac{1}{n}(Xw - z)^T(Xw - z)$$

The gradient is given by

$$\frac{\partial}{\partial w}L(w) = \frac{2}{n}X^T(Xw - z)$$

Implement the vectorized version of the gradient of the regression loss function.

In [95]:

```
# gradient function definition
def grad_loss(z_pred,z,X):

    grad = X.T @ (z_pred - z) * 2 / n

    return grad


# Test grad function
z_pred = f_pred(X,w)
grad = grad_loss(z_pred,z,X)
```

6. Implement the gradient descent algorithm

Vectorized implementation:

$$w^{k+1} = w^k - \tau \frac{2}{n} X^T (X w^k - z)$$

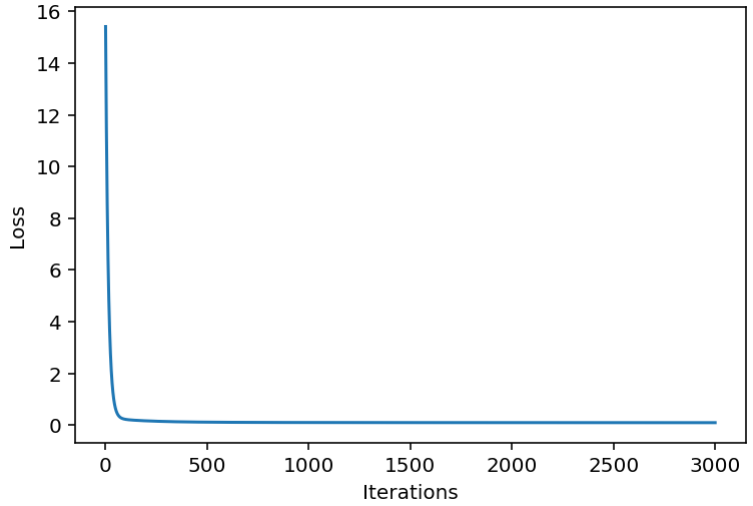
Implement the vectorized version of the gradient descent function.

Plot the loss values $L(w^k)$ with respect to iteration k the number of iterations.

In [107]:

```
# gradient descent function definition
def grad_desc(X, z , w_init=np.array([0,0])[:,None] ,tau=0.01, max_iter=500):

    L_iters = np.zeros([max_iter]) # record the loss values
    w_iters = np.zeros([max_iter,10]) # record the loss values
    w = w_init # initialization

    for i in range(max_iter): # loop over the iterations

        z_pred = f_pred(X,w).reshape(3600,) # linear predicition function
        grad_f = grad_loss(z_pred,z,X) # gradient of the loss
        w = w.reshape(10,) - tau * grad_f # update rule of gradient descent
        L_iters[i] = loss_mse(z_pred,z) # save the current loss value
        w_iters[i,:] = w # save the current w value

    return w, L_iters, w_iters


# run gradient descent algorithm
start = time.time()
w_init = np.array([1,1,1,1,1,1,1,1,1,1])[:,None] # [:,None] adds a singleton dimension
tau = 0.01
max_iter = 3000

w, L_iters, w_iters = grad_desc(X,z,w_init,tau,max_iter)

print('Time=',time.time() - start) # plot the computational cost
print(L_iters[-1]) # plot the last value of the loss
print(w_iters[-1].reshape(10,1)) # plot the last value of the parameter w


# plot
plt.figure(2)
plt.plot(L_iters) # plot the loss curve
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.show()
```

```
Time= 0.23143887519836426
0.0893191138081287
[[-0.01091137]
 [ 0.71469775]
 [ 0.0978672 ]
 [-0.18743504]
 [ 0.9445939 ]
 [-0.34041818]
 [ 0.04246111]
 [-0.62572043]
 [ 0.28014542]
 [ 0.18234041]]
```



7. Plot the prediction function

$$f_w(x,y) = w_0 f_0(x,y) + w_1 f_1(x,y) + w_2 f_2(x,y) + w_3 f_3(x,y) + w_4 f_4(x,y) + w_5 f_5(x,y) + w_6 f_6(x,y) + w_7 f_7(x,y) + w_8 f_8(x,y) + w_9 f_9(x,y)$$

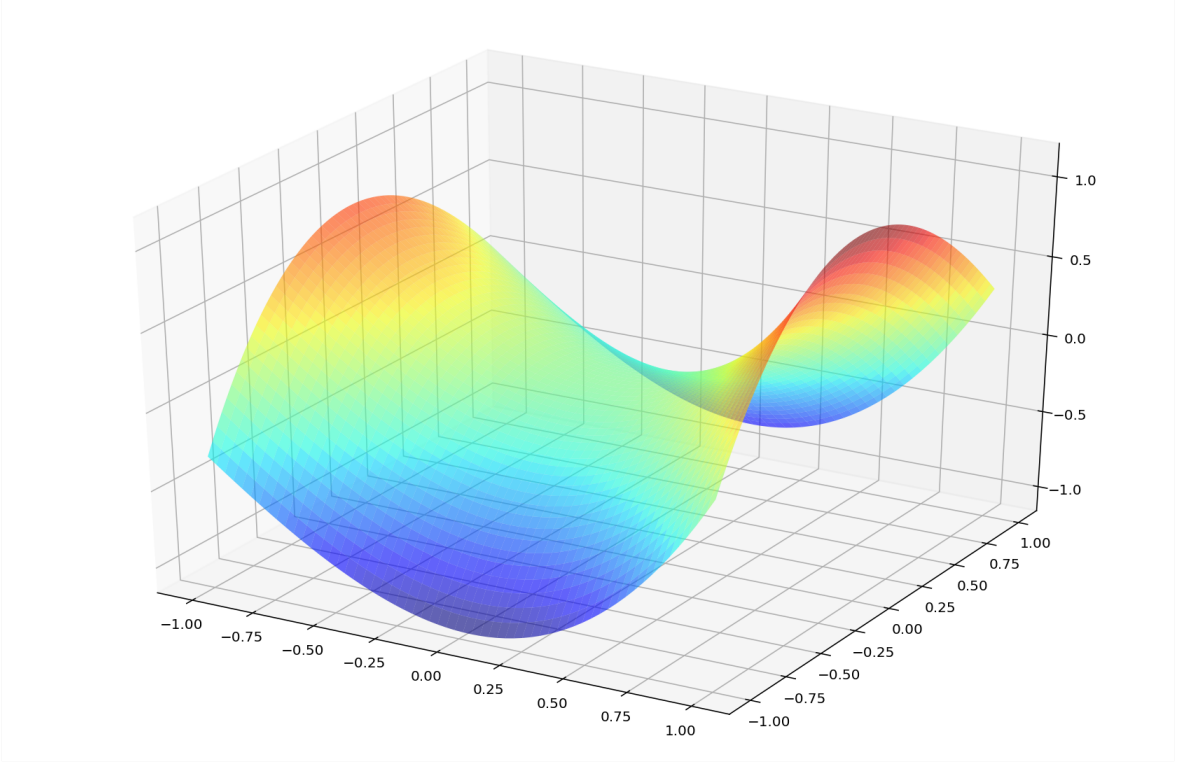
(You may use numpy function `meshgrid` and `plot_surface` for plot the linear prediction function.)

In [157]:

```
x_coordinate = np.linspace(-1, 1, 60)
y_coordinate = np.linspace(-1, 1, 60)

x_pred, y_pred = np.meshgrid(x_coordinate, y_coordinate, indexing='xy')
z_pred = w_iters[-1][0] + w_iters[-1][1] * (x_pred) + w_iters[-1][2] * (y_pred) + w_iters[-1][3] *

# plot
fig = plt.figure(figsize=(15,10))
ax = fig.add_subplot(1, 1, 1, projection='3d')

ax.plot_surface(x_pred, y_pred, z_pred, rstride=1, cstride=1, alpha=0.6, cmap=plt.cm.jet)
ax.set_zlim(z_pred.min(),z_pred.max())
```

Out[157]:

```
(-1.1147011683048522, 1.1583513174164108)
```



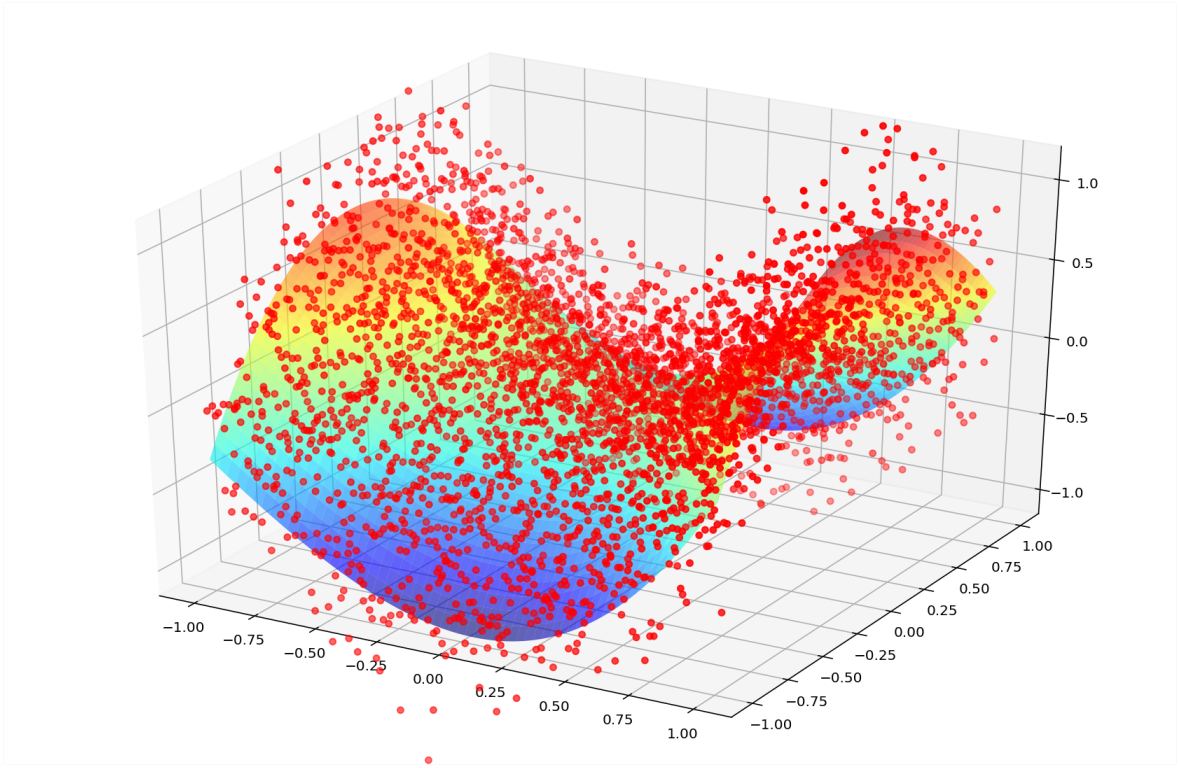
8. Plot the prediction function superimposed on the training data

In [161]:

```
x_coordinate = np.linspace(-1, 1, 60)
y_coordinate = np.linspace(-1, 1, 60)

x_pred, y_pred = np.meshgrid(x_coordinate, y_coordinate, indexing='xy')
w_final = w_iters[-1]

z_pred = w_iters[-1][0] + w_iters[-1][1] * (x_pred) + w_iters[-1][2] * (y_pred) + w_iters[-1][3] *

# plot
fig = plt.figure(figsize=(15,10))
ax = fig.add_subplot(1, 1, 1, projection='3d')

ax.plot_surface(x_pred, y_pred, z_pred, rstride=1, cstride=1, alpha=0.6, cmap=plt.cm.jet)
ax.set_zlim(z_pred.min(),z_pred.max())
ax.scatter(x_train, y_train, z_train, color='r')
```

Out[161]:

<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x16a79da7e48>



Output results

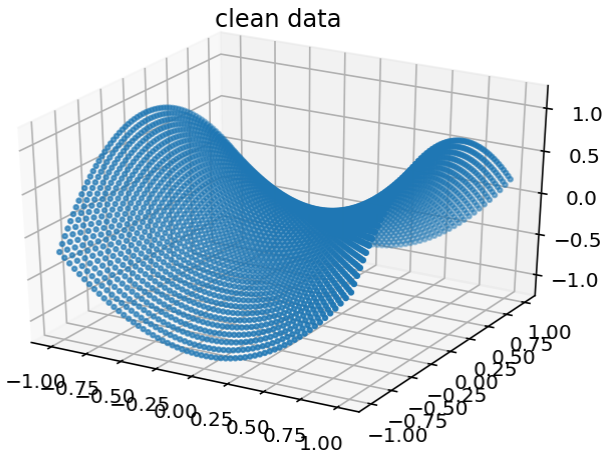
1. Plot the clean data in 3D cartesian coordinate system (1pt)

In [169]:

```
fig1 = plt.figure()
ax1 = fig1.gca(projection='3d')
ax1.scatter(x_clean, y_clean, z_clean, marker='.')
ax1.set_title('clean data')
```

Out[169]:

```
Text(0.5, 0.92, 'clean data')
```



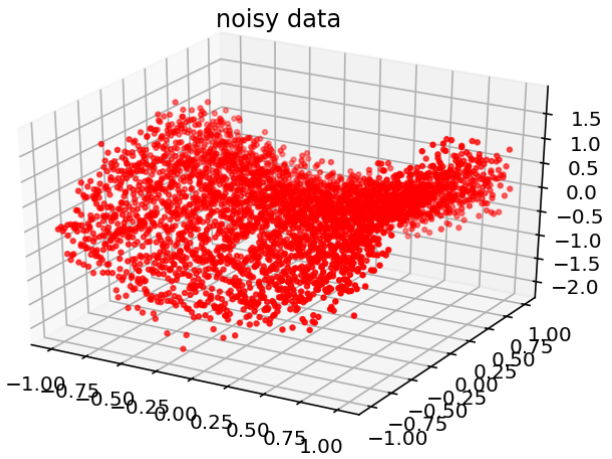
2. Plot the noisy data in 3D cartesian coordinate system (1pt)

In [168]:

```
fig2 = plt.figure()
ax2 = fig2.gca(projection='3d')
ax2.scatter(x_train, y_train, z_train, marker='.', color='r')
ax2.set_title('noisy data')
```

Out[168]:

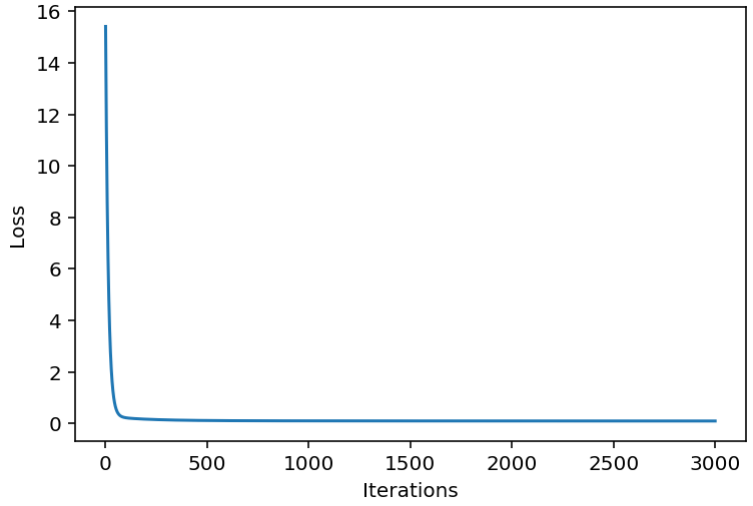
```
Text(0.5, 0.92, 'noisy data')
```



3. Plot the loss curve in the course of gradient descent (2pt)

In [167]:

```
plt.figure(2)
plt.plot(L_iters) # plot the loss curve
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.show()
```



4. Print out the final loss value at convergence of the gradient descent (1pt)

In [166]:

```
print('loss at convergence = ', L_iters[-1]) # plot the last value of the loss
```

```
loss at convergence =  0.0893191138081287
```

5. Print out the final model parameter values at convergence of the gradient descent (1pt)

In [165]:

```
i = 0
for w in w_iters[-1].reshape(10,1):
    print('model parameter: w_' + str(i) + ' = ' + str(w))
    i = i + 1
#print(w_iters[-1].reshape(10,1)) # plot the last value of the parameter w
```

```
model parameter: w_0 = [-0.01091137]
model parameter: w_1 = [0.71469775]
model parameter: w_2 = [0.0978672]
model parameter: w_3 = [-0.18743504]
model parameter: w_4 = [0.9445939]
model parameter: w_5 = [-0.34041818]
model parameter: w_6 = [0.04246111]
model parameter: w_7 = [-0.62572043]
model parameter: w_8 = [0.28014542]
model parameter: w_9 = [0.18234041]
```

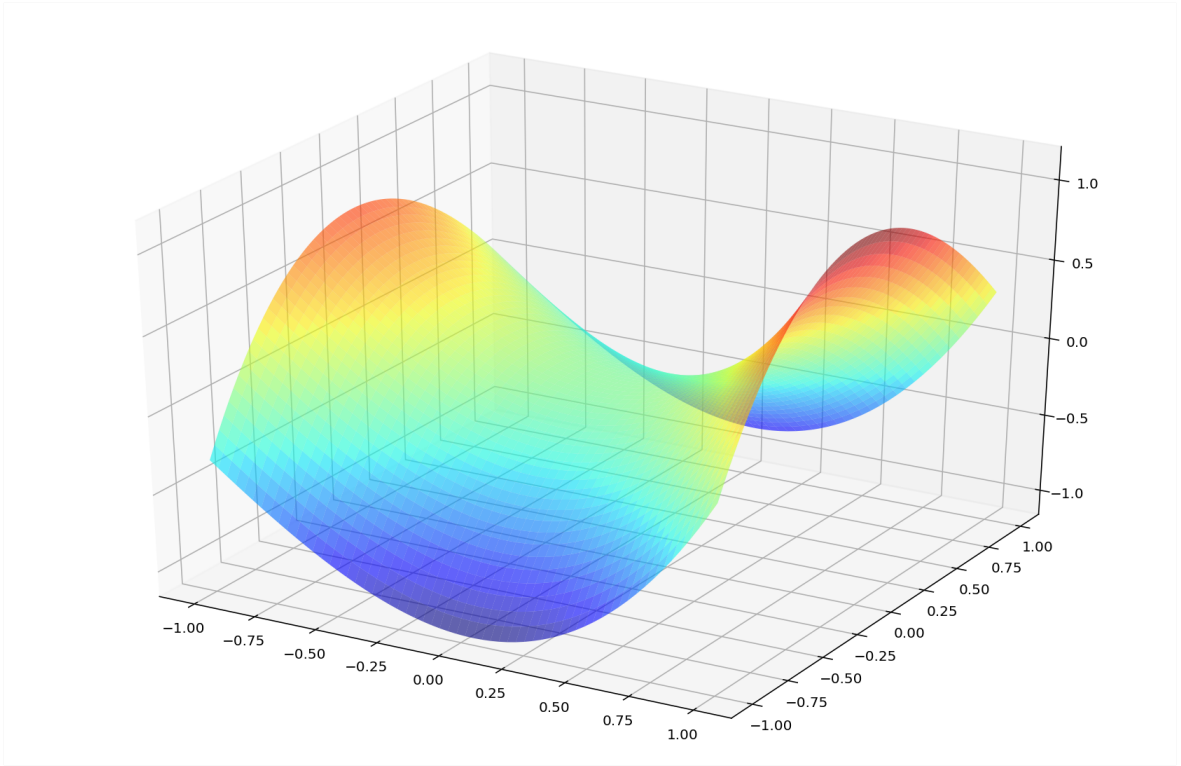
6. Plot the prediction function in 3D cartesian coordinate system (2pt)

In [164]:

```
fig = plt.figure(figsize=(15,10))
ax = fig.add_subplot(1, 1, 1, projection='3d')

ax.plot_surface(x_pred, y_pred, z_pred, rstride=1, cstride=1, alpha=0.6, cmap=plt.cm.jet)
ax.set_zlim(z_pred.min(),z_pred.max())
```

Out[164]:

```
(-1.1147011683048522, 1.1583513174164108)
```



7. Plot the prediction functions superimposed on the training data (2pt)

In [163]:

```
fig = plt.figure(figsize=(15,10))
ax = fig.add_subplot(1, 1, 1, projection='3d')

ax.plot_surface(x_pred, y_pred, z_pred, rstride=1, cstride=1, alpha=0.6, cmap=plt.cm.jet)
ax.set_zlim(z_pred.min(),z_pred.max())
ax.scatter(x_train, y_train, z_train, color='k')
```

Out[163]:

```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x16a047677c8>
```