```
# Collection of necessary functions

import numpy as np

def compute_cost(x,y,theta,m):
"""
Input:
--------------
x: 2D array (input feature array)
m= number of training samples
n= number of input features (including the column of all 1's)
y: 1D array of target values for each sample. Dimensions (1 x m)
theta: 1D array of fitting weights. Dimensions (1 x n)
Output:
--------------
J: Scalar value 0 <= J <= 1
"""
predictions = x.dot(theta)
errors = np.subtract(predictions,y)
sqr_errors = np.square(errors)
J = 1 / (2*m) * np.sum(sqr_errors)
return J

def gradient_descent(x,y,theta,alpha,iterations,m):
"""
Input:
--------------
x: 2D array
m= number of training samples
n= number of features (including column full of ones)
y: 1D array of labels/target values for each training example. Dimensions (m x 1)
theta: 1D array of weights. Dimensions (1 x n)
alpha: Learning rate. Scalar value between 0.1 and 0.01
iterations: Num of iterations. Scalar value.
Output:
--------------
theta: Final Value. 1D array of fitting weights. Dimensions (1 x n)
cost_history: contains value of cost per iteration. 1D array, dimensions (m x 1)
"""
cost_history = np.zeros(iterations) # Init. Mat.
for i in range(iterations): # For each iteration...
predictions = x.dot(theta) # Change predict via current theta.
errors = np.subtract(predictions,y) # Calculate errors
sum_delta = (alpha / m) * x.transpose().dot(errors) # Compute the change in theta.
theta = theta - sum_delta # Calculate new theta value
cost_history[i] = compute_cost(x,y,theta,m) # Find new cost value, then repeat
return theta, cost_history
```