Sujal Burad EE21B144 Week 6

May 5, 2023

1 One variable function:

1.1 Implement a single function for animation and plotting

[1]:
```
# The following imports are assumed for the rest of the problems
import numpy as np
from numpy import cos, sin, pi, exp
import matplotlib.animation as animation
# from sympy import * # Useful to find derivative of a function
```

[2]:
```
# Imports to visualise the gradient descent optimization

%matplotlib ipympl
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation, writers
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm # To color the 3d surface
```

The function and its derivative can be defined in the below code

[3]:
```
def function(x):
    return np.cos(x)**4 - np.sin(x)**3 - 4*np.sin(x)**2 + np.cos(x) + 1
#     return x ** 2 + 3 * x + 8
#     return x ** 2
#     return x**4 - 3*x**2 + 1*x

def derivative(x):
    return (-4 * np.cos(x) ** 3 * np.sin(x)) - (3 * np.sin(x) ** 2 * np.cos(x))␣
 ↪- (8 * np.sin(x) * np.cos(x)) - (np.sin(x))
#     return 2 * x + 3
#     return 2 * x
#     return 4 * x **3 - 6 * x + 1

# x1, and y1 define the sampling space in which to find the minimum value
# The values can be changed to visualise it better
# x1 = np.linspace(-5, 5, 1000)
# y1 = function(x1)
```

```
# plt.close()
# plt.plot(x1, y1)
# Plotting the function to get an idea of the minimas and maximas
# plt.show()
```

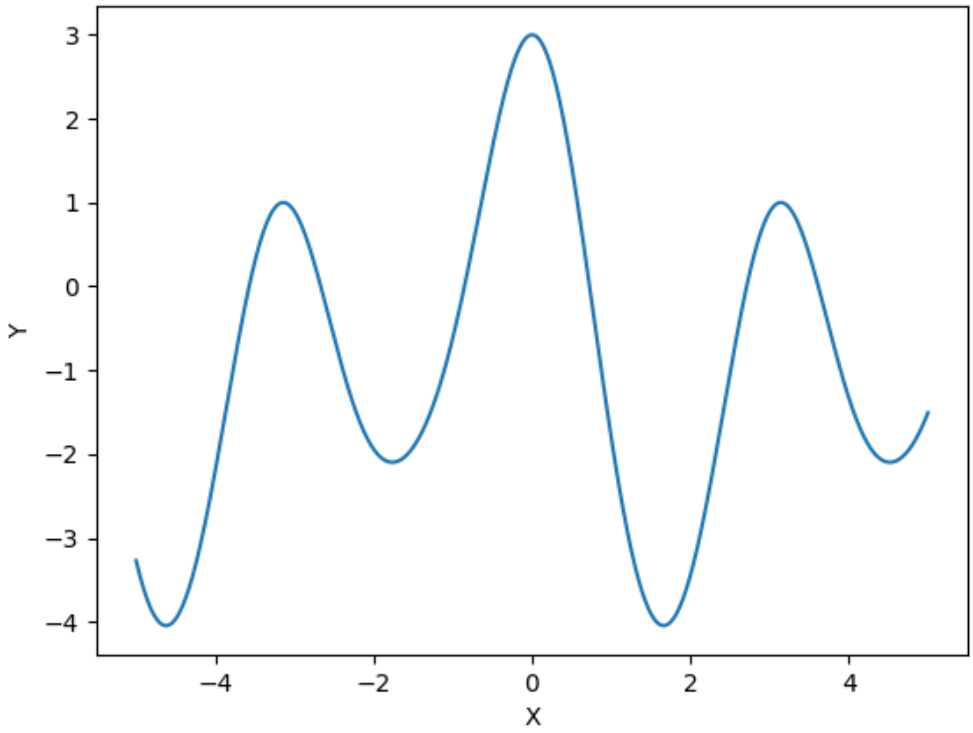
1.1.1 Minimum_using_gradient_descent () :

This function finds the minimum using gradient descent for one variable function. The paramters that I am using to find the minimum using gradient descent for one variable function are: 1] function - It should be the function of type function(variable). 2] derivative - It should be the derivative of the function in type derivative(variable). 3] starting_point - The point at which the gradient descent should start. 4] learning_rate - Learning rate for gradient descent. 5] search_range - It is the sample space inside which to search for the minimum values. The sample space should be an array with a starting point and an ending point. 6] no_of_points - The number of points for which the function would be defined. The number of points in the search_range can be user defined. I have taken the default value as 1000. 7] num_iterations - It if the number of iterations for gradient descent, defined as per the requirement. The number of iterations can be user defined to get more accurate minimum value. I have taken the default value as 1000.

[4]:
```
def minimum_using_gradient_descent(function, derivative, starting_point,␣
↪learning_rate, search_range, no_of_points = 1000, num_iterations = 1000):

    global variable_one, output_one, sp, lr
    variable_one = np.linspace(search_range[0], search_range[1], no_of_points)
    output_one = function(variable_one)
    sp = starting_point
    lr = learning_rate

    for i in range(num_iterations):
        x = starting_point - derivative(starting_point) * learning_rate
        starting_point = x
        y = function(x)

    print('Minimum value of function with the starting point', sp, 'is ', y)
    print('It occurs at x = ', starting_point)

    plt.close()
    plt.subplot().set_xlabel('X')
    plt.subplot().set_ylabel('Y')
    plt.plot(variable_one, output_one)
```

[5]:
```
minimum_using_gradient_descent(function, derivative, 0.1, 0.01, [-5, 5], 1000)
```

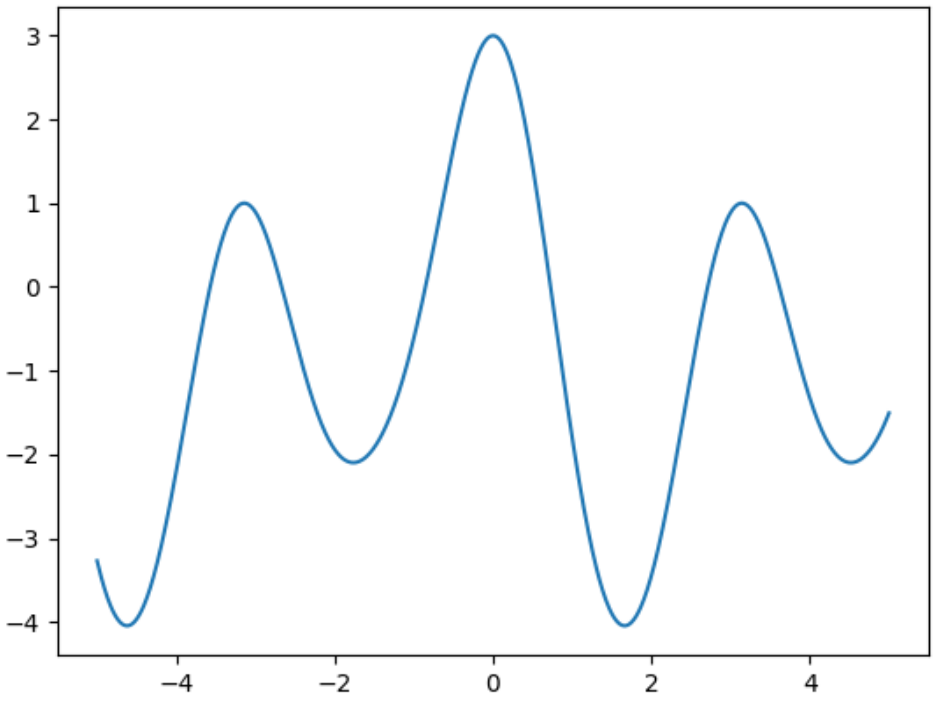
```
Minimum value of function with the starting point 0.1 is  -4.045412051572552
It occurs at x =  1.6616608120437881
```



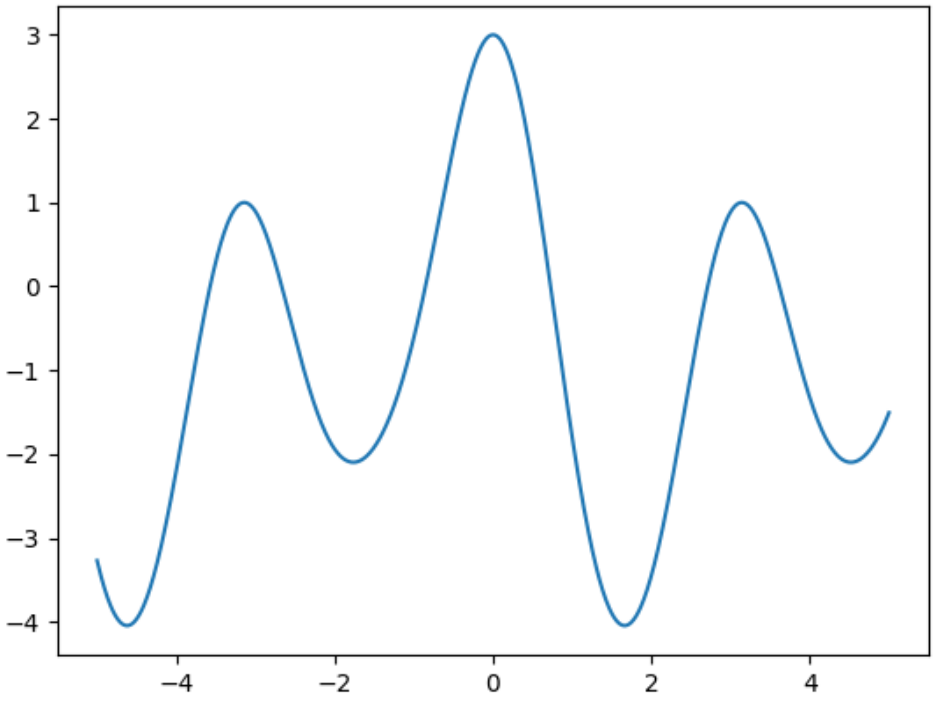
The learning rate and starting point used below are the same when defining the above function

[6]:
```
fig, ax = plt.subplots()
ax.plot(variable_one, output_one)
all_points, = ax.plot([], [], 'ro')
improved_points, = ax.plot([], [], 'go', markersize = 10)
xall, yall = [], []


learning_rate = lr
starting_point = sp



def one_variable_animation(frame):
    global starting_point, learning_rate
    ax.set_title(f'Iteration {frame}')
    x = starting_point - derivative(starting_point) * learning_rate
    starting_point = x
    y = function(x)
    improved_points.set_data(x, y)
```

```
    xall.append(x)
    yall.append(y)
    all_points.set_data(xall, yall)

    return improved_points, all_points,
```



[7]:
```
anim = FuncAnimation(fig, one_variable_animation, frames = range(1000), interval␣
↪= 10, blit = False, repeat = False)

plt.show()
```



2 Two - variables:

The below code for finding the minima for function with two or more independent variables. I am also showing the 3-D animation for only two variable function.

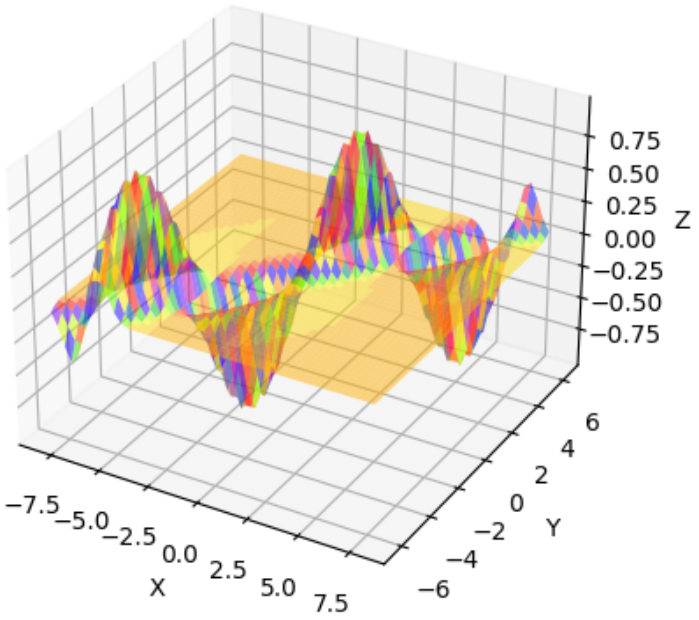
I am assuming that the function, its derivative, and search space of the respective variables will be given in the form of an array.

The multivariable function and its derivative can be defined below.

[8]:
```
def function_two(x):
#     return np.cos(x[0])**4 - np.sin(x[0])**3 - 4*np.sin(x[0])**2 + np.
 ↪cos(x[0]) + 1
    return np.exp(-(x[0] - x[1])**2)*np.sin(x[0])
#     return x[0]**4 - 16*x[0]**3 + 96*x[0]**2 - 256*x[0] + x[1]**2 - 4*x[1] +␣
 ↪262

def derivative_two(x):
#     return (-4 * np.cos(x[0]) ** 3 * np.sin(x[0])) - (3 * np.sin(x[0]) ** 2 *␣
 ↪np.cos(x[0])) - (8 * np.sin(x[0]) * np.cos(x[0])) - (np.sin(x[0]))
```

```
    return np.array([-2*np.exp(-(x[0] - x[1])**2)*np.sin(x[1])*(x[0] - x[1]), np.
↪exp(-(x[0] - x[1])**2)*np.cos(x[1]) + 2*np.exp(-(x[0] - x[1])**2)*np.
↪sin(x[1])*(x[0] - x[1])])
#     return np.array([4*x[0]**3 - 48*x[0]**2 + 192*x[0] - 256, 2*x[1] - 4])

# Here 4*x[0]**3 - 48*x[0]**2 + 192*x[0] - 256 = df/dx, and
# 2*x[1] - 4 = df/dy
```

[9]:
```
# xlim3 =  [-100, 100]
# ylim3 =  [-100, 100]

# For the search range, use the below two variables
xlim3 = np.linspace(-8, 8, 50)
ylim3 = np.linspace(-2*np.pi, 2*np.pi, 50)

# learning_rate = 0.1
# num_iterations = 100
# Define the starting point
# start = np.array([3, 3])


# learning_rate = 0.1
# num_iterations = 1000
# Define the starting point
# start = np.array([5, 5])
# start = np.array([3, 3])


# Define the figure and axes for the animation
fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

# Set up the surface plot
# Create a meshgrid of x and y values to plot the function
X, Y = xlim3, ylim3
X, Y = np.meshgrid(X, Y)
Z = np.array([[function_two([x, y]) for x, y in zip(x_row, y_row)] for x_row,␣
↪y_row in zip(X, Y)])
ax.plot_surface(X, Y, Z, cmap='prism', alpha=0.5)

# Set up the starting point plot
point, = ax.plot([], [], [], 'bo', lw = 2)
```



2.0.1 two_variables_optimisation ():

This function finds the output of the function at every gradient descent step, for every iteration and the given level of tolerance accepted, and stores it in a numpy array.

I am assuming the following conditions given for the function : - For function with more than 2 variables, I am assuming that the corresponding search ranges will be given. - The function is of the form function(x1, x2, x3, . . . , xN), where x1, x2, x3, . . . , xN are the N independent variables. - Taking the search_ranges and derivatives with respect to individual variables given in the form of an array.

Parameters of the function : - function = Function on which the gradient descent would be applied. - derivative = Derivative of the function. It should be given in the form of a numpy array. Each element of this numpy array represents the derivative of the function with the corresponding independent variable. - starting_point, learning_rate = Starting point and learning rate required for gradient descent. The starting_point should be given in the form of a numpy array. - search_range = The range in which the independent variable will be varying. It is done by the variables xlim3, and ylim3 above. - no_of_points = Number of points for which the function is defined. I am taking the default as 1000. - num_iterations = Number of iterations for which the gradient descent will run. Default value = 1000. - error_margin = The error accepted for each subsequent values. When the error is below this error_margin, the loop will stop.

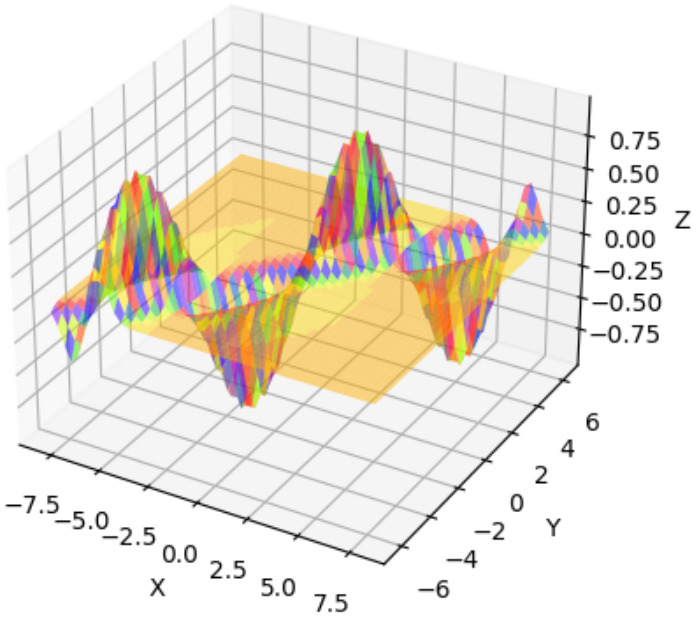
2.0.2 animate():

To see the 3-D animation of the gradient descent for two variable function

[10]:
```
def two_variables_optimisation(function, derivative, starting_point,␣
↪learning_rate, search_range=1, no_of_points = 1000, num_iterations = 100,␣
↪error_margin = 1e-8):
    path = [starting_point]
    for i in range(num_iterations):
        current_point = path[-1]
        new_point = current_point - learning_rate * derivative(current_point)
        # The below can be used for error margin
#         new = np.array(new_point)
#         curr = np.array(current_point)
#         new = new.astype(float)
#         curr = curr.astype(float)
#         # current_point = current_point.astype(float)
#         # new_point = new_point.astype(float)
#         if np.linalg.norm(new - curr) < error_margin:
#             break
        path.append(new_point)

    return np.array(path)
```

[11]:
```
# Set up the animation
path = two_variables_optimisation(function_two, derivative_two, np.array([3,␣
↪3]), 0.1)

print("The final point after gradient descent")
print(f'x:{path[-1][0]}, y:{path[-1][1]}, z:{function_two(path[-1])}')

def animate(i, path):
    point.set_data(path[:i, 0], path[:i, 1])
    point.set_3d_properties(function_two(path[:i, :].T))
    ax.set_title(f'Iteration {i}')

    return point,

ani = FuncAnimation(fig, animate, frames=len(path) + 1, interval = 10,␣
↪blit=True, fargs = (path, ))


# Show the animation
display_animation = ani
plt.show()
```

```
The final point after gradient descent
x:4.678135752228813, y:4.685646798832482, z:-0.9993570343982823
```



2.0.3 multi_variable_optima() :

This function finds the optima point based on gradient descent for the given function and its derivative, for more than or equal to two independent variables of the function.

The paramters of this function and the function should follow the conditions as explained in two_variables_optimisation().

The parameters used in this function are also similar to the one used in two_variables_optimisation().

[12]:
```
# The derivative also returns a numpy array with each element found by the␣
 ↪derivation with the corresponding independent variable
def multi_variables_optima(function, derivative, starting_point, learning_rate =␣
 ↪0.01, num_iterations = 1000, tolerance = 1e-6):

    x = starting_point
    for i in range(num_iterations):
        x_new = x - learning_rate * derivative(x)
#         if(np.linalg.norm(x_new - x) < tolerance):
#             break
        x = x_new
```

```
    x_min = x
    y_min = function(x_min)
    return np.array(x_min), np.array(y_min)
```

[13]:
```
def function_multi(x):
#     return np.cos(x[0])**4 - np.sin(x[0])**3 - 4*np.sin(x[0])**2 + np.
 ↪cos(x[0]) + 1
    return np.exp(-(x[0] - x[1])**2)*sin(x[0])
#     return x[0]**4 - 16*x[0]**3 + 96*x[0]**2 - 256*x[0] + x[1]**2 - 4*x[1] +␣
 ↪262

def derivative_multi(x):
#     return (-4 * np.cos(x[0]) ** 3 * np.sin(x[0])) - (3 * np.sin(x[0]) ** 2 *␣
 ↪np.cos(x[0])) - (8 * np.sin(x[0]) * np.cos(x[0])) - (np.sin(x[0]))
    return np.array([-2*np.exp(-(x[0] - x[1])**2)*np.sin(x[1])*(x[0] - x[1]), np.
 ↪exp(-(x[0] - x[1])**2)*np.cos(x[1]) + 2*np.exp(-(x[0] - x[1])**2)*np.
 ↪sin(x[1])*(x[0] - x[1])])
#     return np.array([4*x[0]**3 - 48*x[0]**2 + 192*x[0] - 256, 2*x[1] - 4])


x_min, y_min = multi_variables_optima(function_multi, derivative_multi, np.
 ↪array([3, 3]))

print("The optimised point = ", x_min)
print("The corresponding value of the function = ", y_min)
```

```
The optimised point =  [4.67593173 4.68392638]
The corresponding value of the function =  -0.9992716380243933
```

[ ]:

[ ]: