STAT 5810 Homework 2

Nicole Fleming
A02079019
due 5/10/2020

by 8am

Plotting code is included after all the problems as an index.

Plots for 1b and 1d are included inline with the problem, as are the graphs and code for 2. The graphs are also included at the end in the index for a larger scale reference.

I worked on #4 with Ethan and Arianna (Ari)
I helped Ari on #6.
I got some pseudo-code guidance on #2 from Ethan. (to update position)

① Determine the Taylor series approximation for the following functions (a-c) up to the quadratic term and (d) up to the 10th term. Plot a 3D figure for part (b) displaying the original function alongside the first order & second order approximations (heat maps). Plot a 2D figure for part (d) showing the original function alongside each of the n^{th}-order approximations up to the 10th order. (Hint: in (d) look if you can find a pattern, or if some terms are likely to disappear).

a) $f(x,y,z) = x^{y+z}$, at $x=1, y=1, z=1$

b) $f(x,y) = \cos(xy)$, at $x=1$

c) $f(x,y) = e^{-x^2-y^2}$, at $x=1, y=2$

d) $f(x) = e^{-x^2}$, $x=0$

---

a) $f(x,y,z) = x^{y+z}$ $\quad x_0=1 \quad y_0=1 \quad z_0=1$

Form: $\sum_{|\vec{\alpha}|=0}^{\infty} \frac{D^{\vec{\alpha}} f}{\vec{\alpha}!}\left(\vec{x}-\vec{x}_0\right)^{\vec{\alpha}}$

$f(x) = 1^{1+1} = 1^2 = 1 \longrightarrow 1$

$\partial x = (y+z)x^{y+z-1} \longrightarrow (1+1)1^{1+1-1} = 2\cdot 1^1 = 2$

$\partial xy = ((y+z)x)\partial y = (y+z)\cdot x = x^{y+z-1} \longrightarrow 1$

$\partial xz = ((y+z)x)\partial z = (y+z)\cdot x = x^{y+z-1} \longrightarrow 1$

$\partial y = \log(x)x^{y+z} \longrightarrow \log(1)=0 \quad 0\cdot 1^{1+1} = 0$

$\partial yx = x^{y+z-1}(\log(x)(y+z)+1) \longrightarrow 1(0(2)+1) = 1\cdot 1 = 1$

$\partial yz = \log^2(x)x^{y+z} \longrightarrow 0$

$\partial z = \log(x)x^{y+z} \longrightarrow 0$

$\partial zy = \log^2(x)x^{y+z} \longrightarrow 0$

$\partial zx = x^{y+z-1}(\log(x)(y+z)+1) \longrightarrow 1^{1+1-1} = 1(0\cdot 2+1) = 1(1) = 1$

$\partial x^2 = (y+z-1)(y+z)x^{y+z-2} \longrightarrow 1(2)1^0 = 2$

$\partial y^2 = \log^2(x)x^{y+z} \longrightarrow 0$

$\partial z^2 = \log^2(x)x^{y+z} \longrightarrow 0$

$\longrightarrow 0$

$$\Rightarrow T^2_{f(x,y,z)}(x_0,y_0,z_0) = f(x_0,y_0,z_0) + (x-x_0)f_x(x_0,y_0,z_0) + (y-y_0)f_y(x_0,y_0,z_0)$$
$$+ (z-z_0)f_z(x_0,y_0,z_0) + \frac{1}{2!}\left[(x-x_0)^2\frac{\partial^2 f}{\partial x^2}(x_0,y_0,z_0)\right.$$
$$+(x-x_0)(y-y_0)\frac{\partial f}{\partial x\partial y}(x_0,y_0,z_0) + (x-x_0)(z-z_0)\frac{\partial f}{\partial x\partial z}(x_0,y_0,z_0) + (y-y_0)^2\frac{\partial^2 f}{\partial y^2}(x_0,y_0,z_0)$$
$$+(y-y_0)(x-x_0)\frac{\partial f}{\partial y\partial x}(x_0,y_0,z_0) + (y-y_0)(z-z_0)\frac{\partial f}{\partial y\partial z}(x_0,y_0,z_0) + (z-z_0)^2\frac{\partial^2 f}{\partial z^2}(x_0,y_0,z_0)$$
$$\left.+(z-z_0)(x-x_0)\frac{\partial f}{\partial z\partial x}(x_0,y_0,z_0) + (z-z_0)(y-y_0)\frac{\partial f}{\partial z\partial y}(x_0,y_0,z_0)\right]$$

$$T_{f(x,y,z)}(x_0,y_0,z_0) = 1 + 2(x-1) + \cancel{0(y-1)} + \cancel{0(z-1)}$$
$$+ \frac{1}{2!}\Big(2(x-1)^2 + 1(x-1)(y-1) + 1(x-1)(z-1) + \cancel{0(y-1)^2}$$
$$+ 1(y-1)(x-1) + \cancel{0(y-1)(z-1)} + \cancel{0(z-1)^2}$$
$$+ \cancel{0(z-1)(y-1)} + 1(z-1)(x-1)\Big)$$

$$= 1 + 2(x-1) + \frac{2}{2!}(x-1)^2 + \underline{\frac{1}{2!}(x-1)(y-1)} + \frac{1}{2!}(x-1)(z-1)$$
$$\underline{+\frac{1}{2!}(y-1)(x-1)} + \underline{\frac{1}{2!}(z-1)(x-1)}$$

due to the Associative property of Multiplication

The Quadratic term for $f(x_0,y_0,z_0$ f the Taylor Series is:

$$= 1 + 2(x-1) + \frac{2}{2!}(x-1)^2 + \left(\frac{1}{2!}(x-1)(y-1)\right)^2 + \left(\frac{1}{2!}x-)(z-1)\right)^2$$

b) $f(x,y) = \cos(xy)$ at $x_0 = 1$ and $y_0 = 2$

Form: $\sum_{|\alpha|=0}^{\infty} \frac{D^\alpha f}{\alpha!}(\vec{X} - \vec{X}_0)$

$f(xy) = \cos(y) \longrightarrow \cos(2)$

$\partial x = -y\sin(xy) \longrightarrow -2\sin(2)$

$\partial xy = -\sin(xy) - xy\cos(xy) \longrightarrow 2\cos 2 = 2\cos(2)$

$\partial y = -x\sin(xy) \longrightarrow -\sin(2)$

$\partial yx = -\sin(xy) - xy\cos(xy) \longrightarrow 2\cos(2)$

$\partial x^2 = -y^2\sin(xy) \longrightarrow -4\sin(2)$

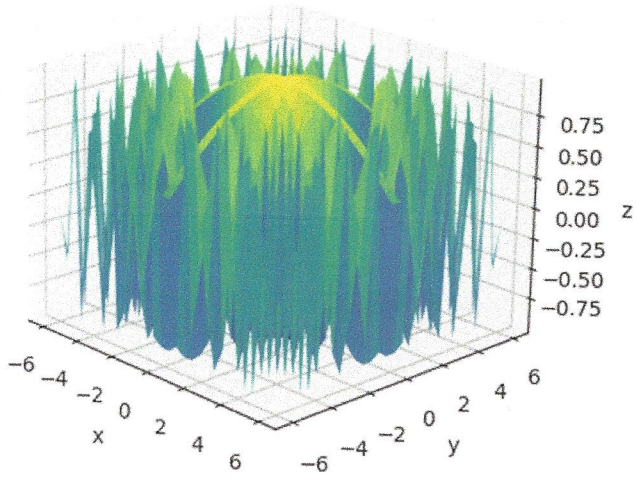
$\partial y^2 = -x^2\cos(xy) \longrightarrow -\cos(2)$

$$T^2_{f(x,y)}(x_0,y_0) = f(x_0,y_0) + (x-x_0)f(x_0,y_0) + (y-y_0)f(x_0,y_0)$$
$$+ \frac{1}{2!}\left[(x-x_0)^2\frac{\partial^2 f}{\partial x^2}(x_0,y_0) + (x-x_0)(y-y_0)\frac{\partial^2 f}{\partial x\partial y}(x_0,y_0)\right.$$
$$\left. + (y-y_0)^2\frac{\partial^2 f}{\partial y^2}(x_0,y_0) + (y-y_0)(x-x_0)\frac{\partial^2 f}{\partial y\partial x}(x_0,y_0)\right]$$

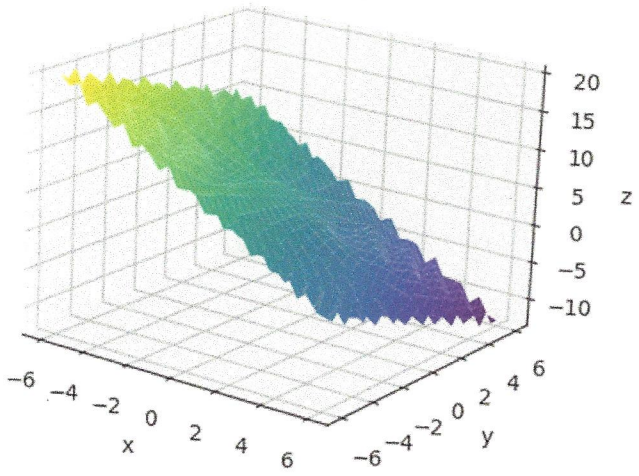
$$T^2_{f(x,y)} = \cos(2) - 2\sin(2)(x-1) - \sin(2)(y-2) - \frac{1}{2!}(x-1)^2(4\cos(2))$$
$$+ \frac{1}{2!}(x-1)(y-2)(2\cos(2)) - \frac{1}{2!}(y-2)^2(\cos(2)) + \frac{1}{2!}(y-2)(x-1)(2\cos(2))$$

Plots:

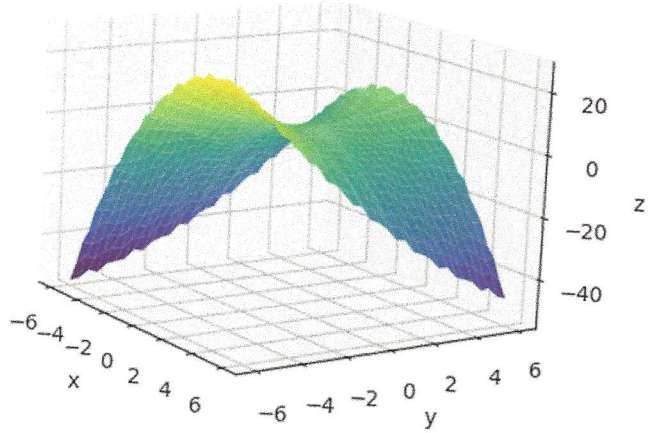
of original function:



of 1st derivative:



of second derivative:



c) $f(x,y) = e^{x^2-y^2}$ $\quad x_0 = 1 \quad y_0 = 2$

Form: $\sum_{|\vec{\alpha}|=0}^{\infty} \frac{D^{\vec{\alpha}} f}{\alpha!}(\vec{X}-\vec{X}_0)$

$f(x,y) = e^{1^2-2^2} = e^{-1} \longrightarrow e^{-1}$

$\partial x = 2x e^{x^2-y^2} \longrightarrow 2e^{-1}$

$\partial xy = -4xy e^{x^2-y^2} \longrightarrow -8e^{-1}$

$\partial y = -2y e^{x^2-y^2} \longrightarrow -4e^{-1}$

$\partial yx = -4xy e^{x^2-y^2} \longrightarrow -8e^{-1}$

$\partial x^2 = 2(2x^2+1) e^{x^2-y^2} \longrightarrow 6e^{-1}$

$\partial y^2 = 2(2y^2-1) e^{x^2-y^2} \longrightarrow 14e^{-1}$

$$T^2_{f(x,y)}(x_0,y_0) = f(x,y) + (x-x_0) f_x(x_0,y_0) + (y-y_0) f_y(x_0,y_0) + \frac{1}{2!}\left[(x-x_0)^2 \frac{\partial^2 f}{\partial x^2}(x_0,y_0) + (x-x_0)(y-y_0)\frac{\partial f}{\partial x \partial y}(x_0,y_0) + (y-y_0)^2 \frac{\partial^2 f}{\partial y^2}(x_0,y_0) + (y-y_0)(x-x_0)\frac{\partial f}{\partial y \partial x}(x_0,y_0)\right]$$

$$= e^{-1} + 2e^{-1}(x-1) - 4e^{-1}(y-2) + \frac{1}{2!}6e^{-1}(x-1)^2 - \frac{1}{2!}8e^{-1}(x-1)(y-2) + \frac{1}{2!}14e^{-1}(y-2)^2 - \frac{1}{2!}8e^{-1}(x-1)(y-2)$$

$$\boxed{T^2_{f(x,y)}(x_0,y_0) = e^{-1} + 2e^{-1}(x-1) - 4e^{-1}(y-2) + \frac{1}{2!}6e^{-1}(x-1)^2 - \frac{1}{2!}14e^{-1}(y-2)^2 \cdot -2\left(\frac{1}{2!}8e^{-1}(x-1)(y-2)\right)}$$

d) $f(x) = e^{-x^2} \qquad x = 0$

$f(x) = e^{-x^2} = e^{-0} = e^0 = 1 \longrightarrow 1$

$dx = -2xe^{-x^2} \longrightarrow 0$

$dx^2 = e^{-x^2}(4x^2 - 2) \longrightarrow 0$

$dx^3 = -4e^{-x^2} \cdot x(2x^2 - 3) \longrightarrow 0$

$dx^4 = 4e^{-x^2}(4x^4 - 12x^2 + 3) \longrightarrow 0$

$dx^5 = -8e^{-x^2} \cdot x(4x^4 - 20x^2 + 15) \longrightarrow 0$

$dx^6 = 8e^{-x^2}(8x^6 - 60x^4 + 90x^2 - 15) \longrightarrow 0$

$dx^7 = -16e^{-x^2} \cdot x(8x^6 - 84x^4 + 210x^2 - 105) \longrightarrow 0$

$dx^8 = 16e^{-x^2}(16x^8 - 224x^6 + 840x^4 - 840x^2 + 105) \longrightarrow 0$

$dx^9 = -32e^{-x^2} \cdot x(16x^8 - 288x^6 + 1512x^4 - 2520x^2 + 945) \longrightarrow 0$

$dx^{10} = 32e^{-x^2}(32x^{10} - 720x^8 + 5040x^6 - 12600x^4 + 9450x^2 - 945) \longrightarrow 0$

$$T^{10}_{f(x)}(x_0) = f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2!}(x-x_0)^2 f''(x_0) + \frac{1}{3!}(x-x_0)^3 f'''(x_0)$$
$$+ \frac{1}{4!}(x-x_0)^4 f^9(x_0) + \frac{1}{5!}f^5(x_0)(x-x_0)^5 + \frac{1}{6!}(x-x_0)^6 f^6(x_0)$$
$$+ \frac{1}{7!}(x-x_0)^7 f^7(x_0) + \frac{1}{8!}(x-x_0)f^8(x_0) + \frac{1}{9!}(x-x_0)f^9(x_0)$$
$$+ \frac{1}{10!}(x-x_0)^{10} f^{10}(x_0) + \underbrace{\frac{1}{11!}(x-x_0)^{11} f^{11}(?)}_{\text{error}}$$

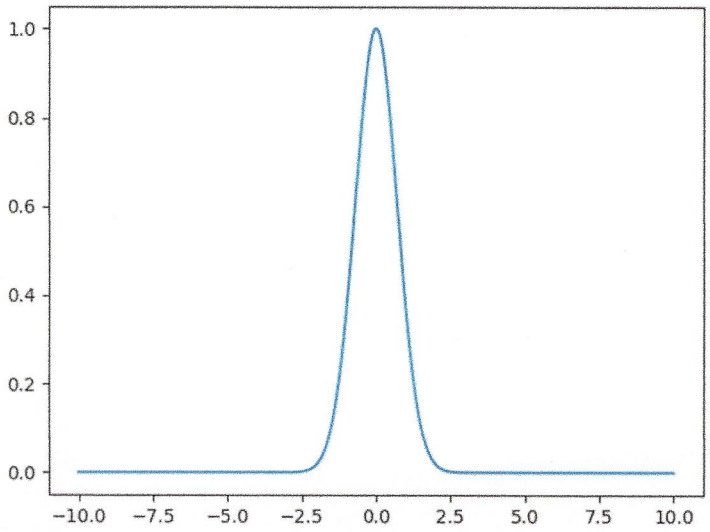
$$T^{10}_{f(x)}(x_0) = 1 + (x-0)(-2xe^{-x^2}) + \frac{1}{2!}(x-0)^2(e^{-x^2}(4x^2-2))$$
$$+ \frac{1}{3!}(x-0)^3(-4e^{-x^2}(x(2x^2-3))) + \frac{1}{4!}(x-0)^4(4e^{-x^2}(4x^4-12x^2+3))$$
$$+ \frac{1}{5!}(x-0)^5(-8e^{-x^2}(x(4x^4-20x^2+15))) + \frac{1}{6!}(x-0)^6(8e^{-x^2}(8x^6-60x^4+90x^2-15)))$$
$$+ \frac{1}{7!}(x-0)^7(-16e^{-x^2}(x(8x^6-84x^4+210x^2-105)))$$
$$+ \frac{1}{8!}(x-0)^8(16e^{-x^2}(16x^8-224x^6+840x^4-840x^2+105)))$$
$$+ \frac{1}{9!}(x-0)^9(-32e^{-x^2}(x(16x^8-288x^6+1512x^4-2520x^2+945)))$$
$$+ \frac{1}{10!}(x-0)^{10}(32e^{-x^2}(32x^{10}-720x^8+5040x^6-12600x^4+9450x^2-945))$$

$+ \ldots$

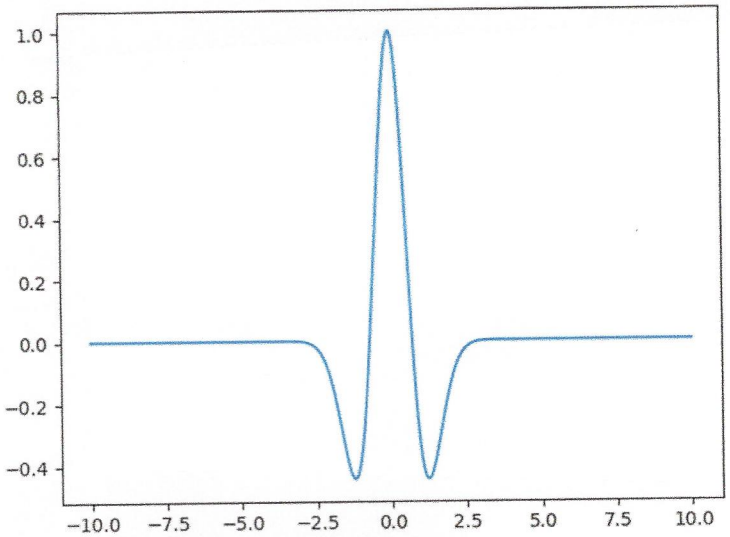
$$= 1 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + \ldots$$

$$= \boxed{1}$$

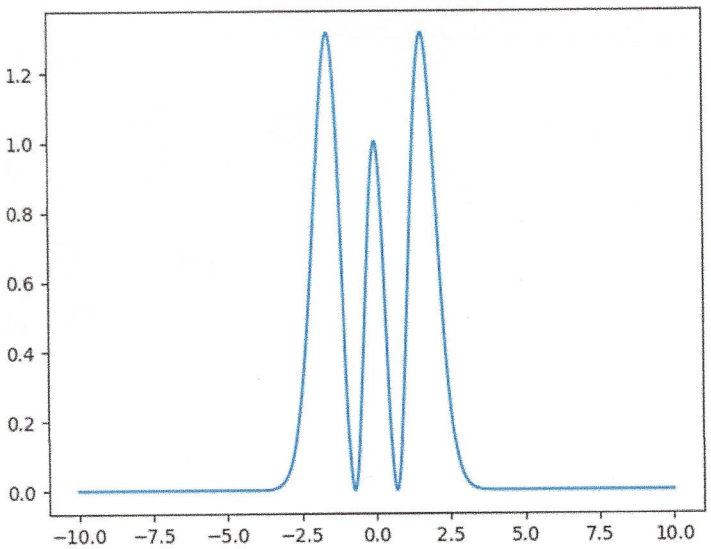
Plot of $f(x)$:



Plot of dx

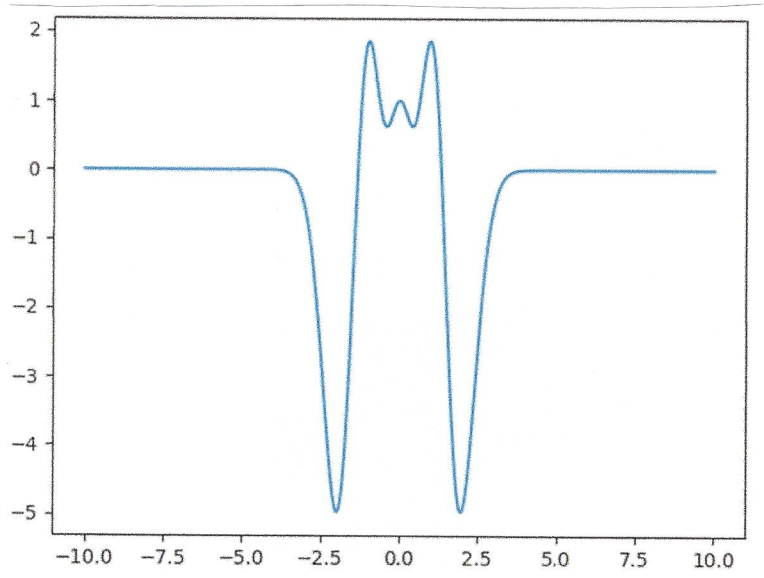


Plot of dx^2

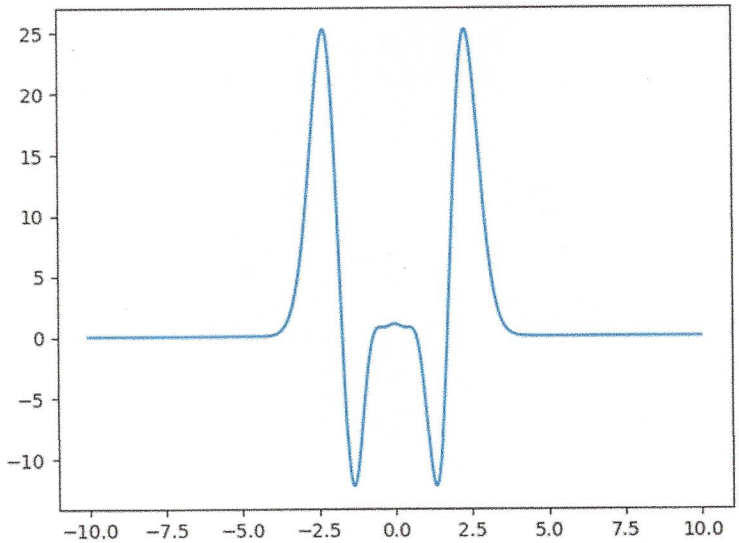


Plot of dx^3

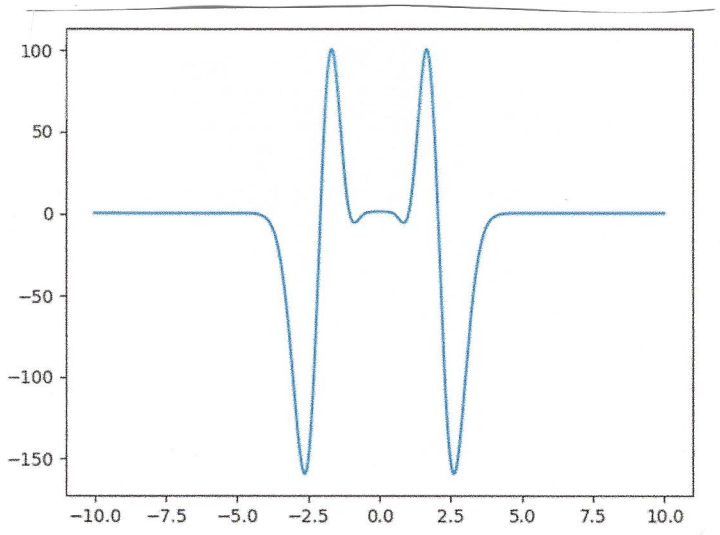
Plot of dx^3



Plot of dx^4

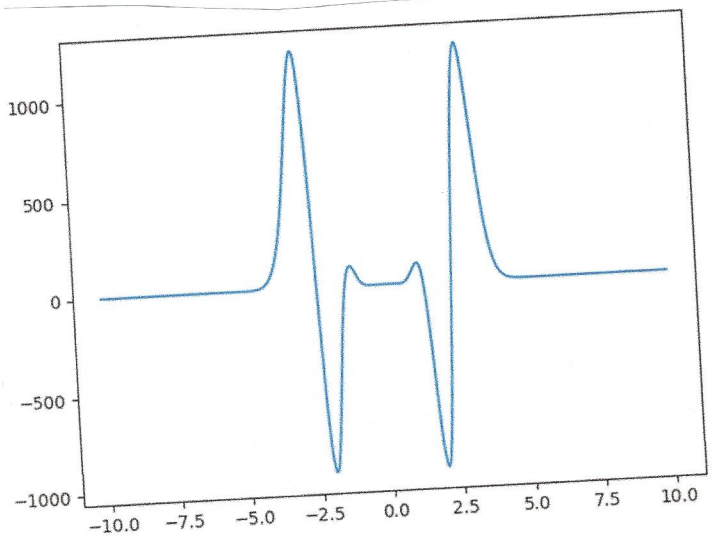


Plot of dx^5

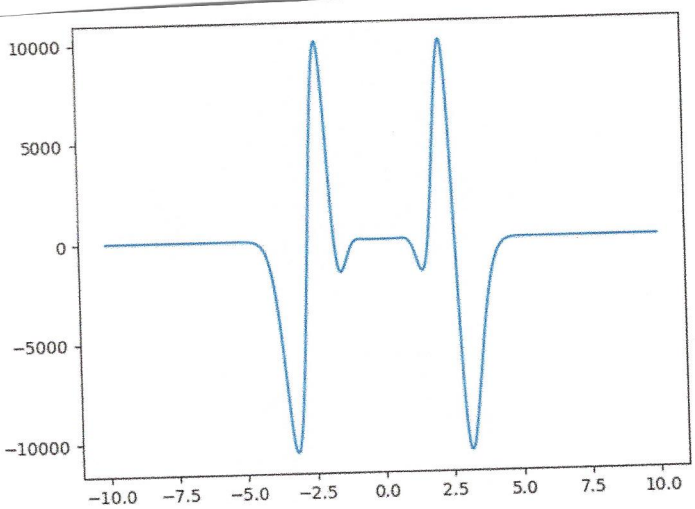


Plot of dx^6 (on Next Page at the top)

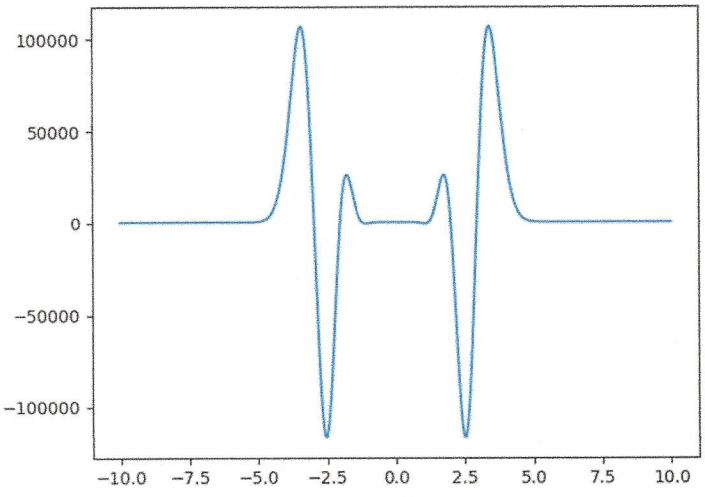
d6



Plot of dx^7

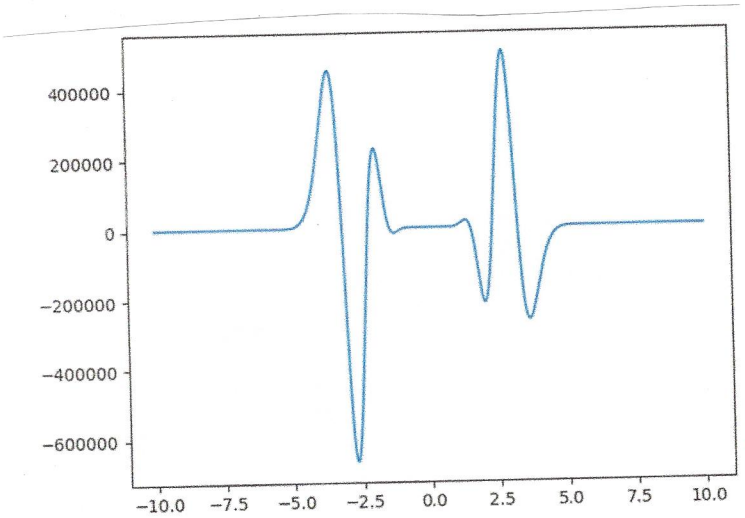


Plot of dx^8

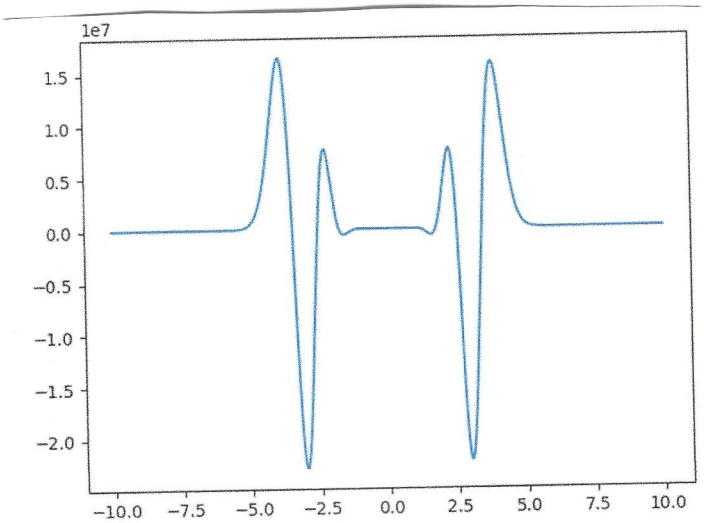


x^9

Plot of dx^9 approx.:



Plot of dx^{10}



---

(2) Write a function in the software of your preference

$(x, f_{min}, fu) = \text{gradientDescent}(f, fgrad, x0, N, gamma)$

that takes as arguements a function f, its gradient $fgrad$, an initial solution $x0 \in \mathbb{R}^n$, a number of iterations N, and a step-size parameter gamma (γ). It should output the solution x, the corresponding value $fmin$ of the input function, and a vector fu which contains all the values of the function after each update of the solution. Thus, make sure your function has a mechanism to store all the values of the function after

each update, and the solution for which you achieve the minimum value.

HOMEWORK 2 QUESTION 2 CODE

```
import numpy as np
import inline as inline
import matplotlib as plt
from matplotlib import pyplot as plt
from mpl_toolkits import mplot3d

# Initialize x0 as a 1x2 vector using the point x0 = (1,1)
x0 = [1, 1]

# define f(x) = x_1^2 + x_2^2
def f(x):
    return x[0]**2 + x[1]**2

# define f'(x) = 2x_1 + 2x_2 (as each partial is added for the total derivative)
def fgrad(x):
  return 2*x[0] + 2*x[1]

# set N equal to the max iterations as specified
N = 200

# set gamma equal to one of the three specified step sizes
gamma = 0.0001 # 0.01,10

# define the gradientDescent function
def gradientDescent(f, fgrad, x0, N, gamma):
  # initialize the starting point
  pos = x0
  #initialize the values of the function
  fu_vals = [f(pos)]
  #for N iterations update and do the following
```

```
    for i in range(0,N):
        #find new value of the gradient
        update_function = fgrad(pos)
        #update the x0 values to new values
        for j in range(0,2):
            # take last position, subtract step size, and multiply by the new gradient
            pos[j] = pos[j] - gamma * update_function
        # add the current position to the fu_vals to be able to graph later.
        fu_vals.append(f(pos))
    # return the postion, the minimum value found(f(pos)) and the fu_vals.
    return (pos, f(pos), fu_vals)



ans = gradientDescent(f, fgrad, x0, N, gamma)


# To plot (gamma needs to change each time to get different plots)
c = []
for i in range(0,N+1):
    c.append(i)


plt.plot(ans[2], c)
plt.title("Gamma is 0.0001")#"(Gamma is 0.01)#("Gamma is 10)
plt.ylabel("Iterations")
plt.xlabel("Function values of Gradient Descent")
plt.show()
```

HOMEWORK 2 QUESTION 1 PLOTTER CODE

```
# This will plot the approximations. I did not plot them all together.

import math


import inline as inline

import matplotlib

from mpl_toolkits import mplot3d


import numpy as np

import matplotlib.pyplot as plt

#%matplotlib inline

#x = np.array()

#fig = plt.figure()

#ax = plt.axes(projection='3d')


def f(x, y):

   return np.cos(x * y) - (2 * (np.sin(2)) * (x - 1)) - (1 * (np.sin(2)) * (y - 2)) + ((1/2) * ((x - 1)**2) * (4 * np.cos(2))) + ((1/2) *
(x - 1) * (y - 2) * 2 * np.cos(2)) - (1/2) * ((y - 2)**2) * np.cos(2) + (1/2) * (y - 2) * (x - 1) * 2 * np.cos(2)

      #Second Order Approximation


         #return return np.cos(x * y) - (2 * (np.sin(2)) * (x - 1)) - (1 * (np.sin(2)) * (y - 2))

         #First Order Approximation


         #return np.cos(xy)

         # Original expression



x = np.linspace(-6, 6, 30)

y = np.linspace(-6, 6, 30)


X, Y = np.meshgrid(x, y)

Z = f(X, Y)
```

```
fig = plt.figure()
#ax = plt.axes(projection='3d')
#ax.contour3D(X, Y, Z, 50, cmap='binary')
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');


#def g(x):
    #10th order approx
    #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x ** 2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16 * np.exp(-x ** 2) * x * (8 * x ** 6 - 84 * x ** 4 + 210 * x ** 2 - 105) * (x - 0) ** 7) +(16*np.exp(-x**2)*(16*x**8-224*x**6 + 840*x**4 -840*x**2+105)*(x-0)**8)-(32*np.exp(-x**2)*(16*x**8-288*x**6+1512*x**4-2520*x**2+945)*(x-0)**9)+(32*np.exp(-x**2)*(32*x**10-720*x**8+5040*x**6-12600*x**4+9450*x**2-945)*(x-0)**10)

    #9th order approx
    #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x ** 2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16 * np.exp(-x ** 2) * x * (8 * x ** 6 - 84 * x ** 4 + 210 * x ** 2 - 105) * (x - 0) ** 7) + (16 * np.exp(-x ** 2) * (16 * x ** 8 - 224 * x ** 6 + 840 * x ** 4 - 840 * x ** 2 + 105) * (x - 0) ** 8) - (32 * np.exp(-x ** 2) * (16 * x ** 8 - 288 * x ** 6 + 1512 * x ** 4 - 2520 * x ** 2 + 945) * (x - 0) ** 9

    #8th order approx
    #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x ** 2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16 * np.exp(-x ** 2) * x * (8 * x ** 6 - 84 * x ** 4 + 210 * x ** 2 - 105) * (x - 0) ** 7) +(16*np.exp(-x**2)*(16*x**8-224*x**6 + 840*x**4 -840*x**2+105)*(x-0)**8)

    #7th order approx
```

```
        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x ** 2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16*np.exp(-x**2)*x*(8*x**6-84*x**4 + 210*x**2 -105)*(x-0)**7)


        #6th order approx

        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) +(8*np.exp(-x**2)*(8*x**6-60*x**4+90*x**2-15)*(x-0)**6)


        #5th order approx

        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8*np.exp(-x**2)*x*(4*x**4-20*x**2+15)*(x-0)**5)


        #4th order approx

        #return np.exp(-x**2)-(2*x*np.exp(-x**2)*(x-0))+(np.exp(-x**2)*(4*x**2-2)*(x-0)**2)-4*np.exp(-x**2)*x*(2*x**2-3)*(x-0)**3+(4*np.exp(-x**2)*(4*x**4-12*x**2+3)*(x-0)**4)


        #3rd order approx

        #return np.exp(-x**2)-(2*x*np.exp(-x**2)*(x-0))+(np.exp(-x**2)*(4*x**2-2)*(x-0)**2) -4*np.exp(-x**2)*x*(2*x**2-3)*(x-0)**3


        #2nd order approx

        #return np.exp(-x**2)-(2*x*np.exp(-x**2)*(x-0))+(np.exp(-x**2)*(4*x**2-2)*(x-0)**2)


        #1st order approx

        #return np.exp(-x**2) -(2*x*np.exp(-x**2)*(x-0))


        #Original function

        #return np.exp(-x**2)


#x = np.linspace ( -10,  # lower limit
#                  10,     # upper limit
```

```
#              1510     # generate 151 points between -10 and 10
#           )
#y = g(x)    # This is already vectorized, that is, y will be a vector!
#plt.plot(x, y)
#plt.show()
plt.show()
```

*Note: to run g(x) the comment with the approximation order may need to follow the code to work properly. I ran it both ways, and had more success without the comment, or with it following the code. I put it ahead of the approximations for g(x), part 1d, to make it easier to read. f(x,y), 1b has it following the code for the approximation.

Now we will see how the algorithm behaves for the simple function

$f(\vec{x}) = x_1^2 + x_2^2$

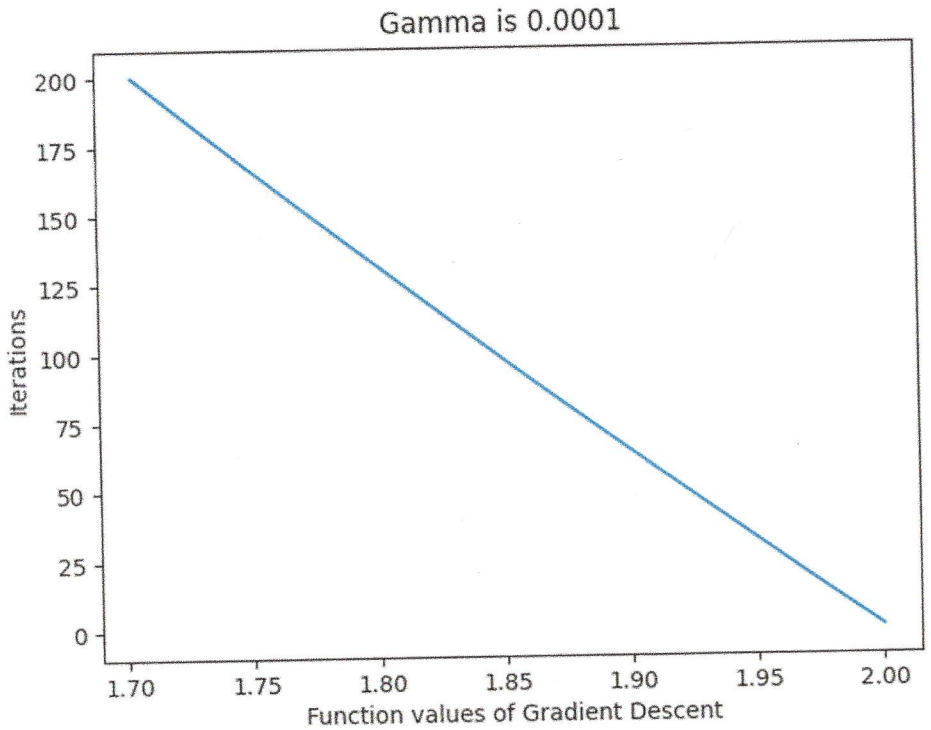
for (γ) gamma = (0.0001, 0.01, 10) where $x_0 = (1,1)$ & $N = 200$

the plots generated are:

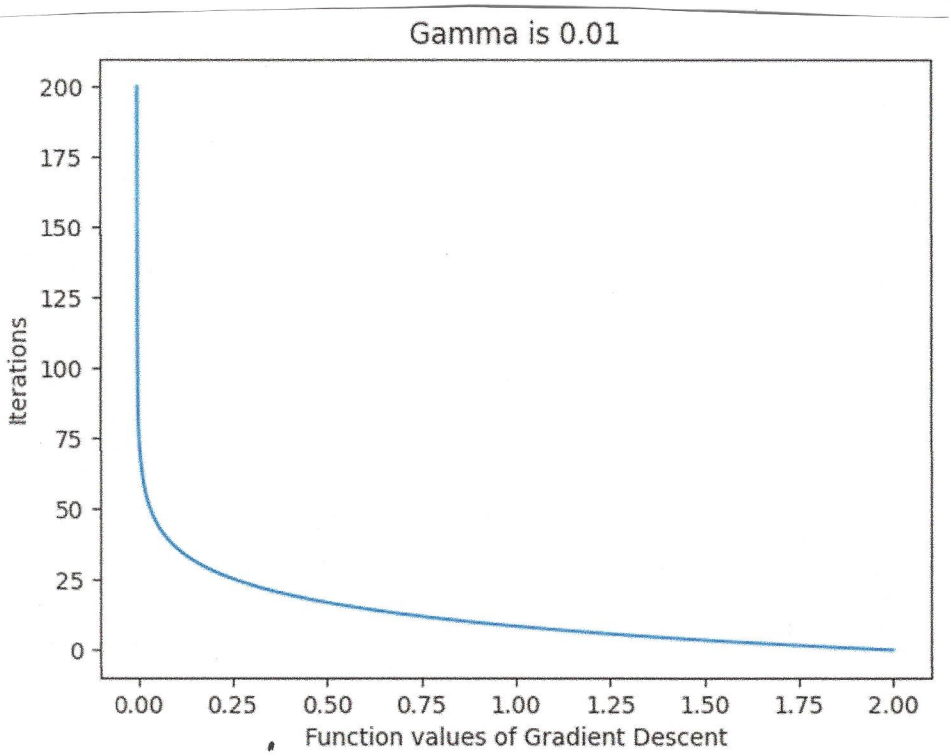
$\gamma = 0.0001$

on next page

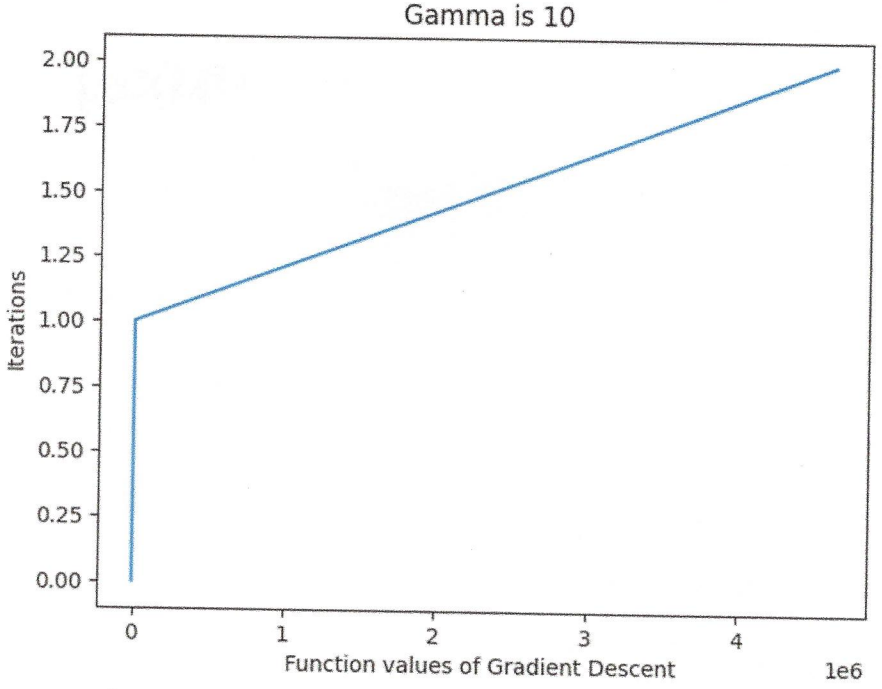
$\gamma = 0.0001$



$\gamma = 0.01$



$\gamma = 10$ graph:



$\gamma = 10$ comments:

for gamma = 10, the step size was increasing. The graph above shows 2 iterations through the code. The reason that this happens occurs in the gradient descent function, in the loop that updates the position. In this update the current position subtracts the (gradient multiplied by gamma). where gamma is 10 and large, the gradient becomes large, causing overflow/overshoot by the function. The method then continues to try to find a minimum with numbers that continuously increase as the graph of $x_1^2 + x_2^2 \Rightarrow$ a bowl  shape where initially the minimum was over shot.

---

(3) Suppose instead of doing back propagation in a neural network, we use the following approximation for each weight:

$$\frac{C(w + \epsilon e_j) - C(w)}{\epsilon} \approx \frac{\partial C}{\partial w_j}$$

$\frac{\partial C}{\partial w_j}$

where C is the cost function, w_j is the j^{th} weight in the network, $\epsilon > 0$ is a small positive number, e_j is the unit vector in the j^{th} direction (i.e. we're only perturbing the j^{th} weight). In other words, we're trying to estimate each derivative directly instead of performing back propagation. Would this approach be faster or slower than back propagation? Explain your reasoning.

Hint: You do not need to be precise in this. Consider the approximate computational cost (in terms of number of passes through the network) of computing the cost function and then consider the number of times you need to compute the cost function in the above approximation. Then compare this to the approximate computational cost of back propagation.

The backpropagation algorithm in machine learning situations is basically a recursive algorithm that computes the derivatives of all inputs in one pass, through a chain of operations. This algorithm has the cost of no more than 3 function evaluations, with the additional overhead to execute the operations, and store and access the intermediate result.

If doing this by direct derivation, then the derivative is computed individually. Once the derivative of the iteration is found, the input is applied to get an answer for the iteration. This does this again and again for each input and derivative term. This computation would end up with some $O(N^2)$ as it runs through each iteration of the derivative. and then to compute with the input ~~function~~

each time. This total cost will be N^2 (or $2 \cdot N$) function evaluations.

The more efficient and less costly way to go about solving/using the above cost function is to use back propagation, since it has a 3 function evaluation cost, where if we were to evaluate directly the function evaluation cost is N^2. The reason back propagation is more effective is due to the RECURSIVE nature of the algorithm, where it takes a lot - in this case N - inputs and is able to compute all the inputs to one output in a couple functions.

Note: Recursion in this case makes the computations from all the derivatives and inputs more efficient.

This space

is

Intentionally

left blank.

---

(4) Consider the following function:

$$L(w) = -\log\left(\prod_{i=1}^{n} \sigma_\omega(\vec{x}_i)^{y_i}\left(1-\sigma_\omega(\vec{x}_i)\right)^{(1-y_i)}\right) + \vec{w}^T\vec{w}$$

where $\sigma_{\vec{w}}(\vec{x}_i) = \frac{1}{1+e^{-(\vec{w}^T\vec{x}_i)}}$. Note that $\vec{w}, \vec{x} \in \mathbb{R}^d$ and $y_i \in \{0,1\}$ for $i = 1, \ldots, n$. This is the objective function for the regularized logistic regression problem (technically, we need an offset as well but we're ignoring that right now). Logistic regression is a common approach to classification. The goal of this problem is given labeled data $(\vec{x}_i, y_i)$ $i = 1, \ldots, n$, find the weights $\vec{w}$ that minimize this objective function. A closed-form solution does <u>not</u> exist, and so some form of gradient descent must be used. In this problem, we'll do some of the preliminary work needed to perform gradient descent and its variants.

a) To use gradient descent, we need to know the gradient for the objective function. Find the gradient $\frac{\partial L}{\partial w}$. Make sure you use matrix/vector notation in your

answer.

b) Some variations on gradient descent (like Newton's Method) are second-order methods that use the Hessian of the objective function. Find the Hessian of L. Make sure you write it using matrix/vector/notation in your answer.

---

a) I treated the $-\log(\cdot)$ as $\log_e$ or $\ln(\cdot)$.

$$= \sum_{i=1}^{n}\left[-\ln\left(\sigma_w(x_i)\right)^{y_i} - \ln\left(1-\sigma_w(x_i)\right)^{(1-y_i)}\right] + w^Tw$$

$$\Rightarrow -\sum_{i=1}^{n}\left[y_i \cdot \ln\left(\sigma_w(x_i)\right)^{y_i} - \ln\left(1-\sigma_w(x_i)\right)^{(1-y_i)}\right] + w^Tw$$

$$\Rightarrow -\sum_{i=1}^{n}\left[y_i \cdot \ln\left(\frac{1}{1+e^{-w^Tx_i}}\right) + (1-y_i)\cdot\ln\left(\frac{e^{-w^Tx_i}}{1+e^{-w^Tx_i}}\right)\right] + w^Tw$$

$$\Rightarrow -\sum_{i=1}^{n}\left[-y_i\ln\left(1+e^{-w^Tx_i}\right) + (1-y_i)\ln\left(e^{-w^Tx_i}\right) - (1-y_i)\ln\left(1+e^{-w^Tx_i}\right)\right] + w^Tw$$

$$\Rightarrow -\sum_{i=1}^{n}\left[-\ln\left(1+e^{-w^Tx_i}\right) + (1-y_i)\ln\left(e^{-w^Tx_i}\right)\right] + w^Tw$$

$$\Rightarrow -\sum_{i=1}^{n}\left[-\ln\left(1+e^{-w^Tx_i}\right) + (1-y_i)(-w^Tx_i)\right] + w^Tw$$

$$\Rightarrow \frac{\partial}{\partial \vec{w}_i}\left(-\sum_{i=1}^{n}\left[-\ln\left(1+e^{-w^Tx_i}\right) + (1-y_i)(-w^Tx_i)\right] + w^Tw\right.$$

$$= -\sum_{i=1}^{n}\left[\frac{-1}{1+e^{-w^Tx_i}}\cdot -x_ie^{-w^Tx_i} - (1-y_i)(-x_i) + 0\cdot(-w^Tx_i)\right] + 2w$$

$$\boxed{\frac{\partial L}{\partial w} = -\sum_{i=1}^{n}\left[\frac{x_i e^{-\vec{w}^T\vec{x_i}}}{1+e^{-\vec{w}^T\vec{x_i}}} - \left(\vec{x}_i + \vec{x}_i y_i\right)\right] + 2\vec{w}} \qquad J: \mathbb{R}^{1\times D}$$

b) $\dfrac{\partial^2 L}{\partial w_{ij}^2} = \dfrac{\partial}{\partial w_{ij}}\left(-\sum_{i=1}^{n}\left[\dfrac{-x_i e^{-w^Tx_i}}{1+e^{-w^Tx_i}} - \left(x_i + x_i y_i\right)\right] + 2\vec{w}_{ii}\right.$

So w/respect to $\vec{w}_{ii}$.

$$\Rightarrow -\sum_{i=1}^{n}\left[\frac{(1+e^{-w^Tx_i})(x_ix_j e^{-w^Tx_i}) - (-x_i e^{-w^Tx_i})(-x_j e^{-w^Tx_i})}{(1+e^{-w^Tx_i})^2}\right] + 2$$

Note: +2 is only on the diagonal w_{ii}

$$= -\sum_{i=1}^{n}\left[\frac{x_ix_j e^{-w^Tx_i} + \cancel{x_ix_j e^{-2(w^Tx_i)}} - \cancel{x_ix_j e^{-2(w^Tx_i)}}}{(1+e^{-w^Tx_i})^2}\right] + 2$$

$$= -\sum_{i=1}^{n}\left[\frac{x_i x_j e^{-w^Tx_i}}{(1+e^{-w^Tx_i})^2}\right] + 2$$

$$= -\sum_{i=1}^{n}\left[\frac{x_i x_j e^{-w^Tx_i}}{(1+e^{-w^Tx_i})^2}\right] + 2$$

$$\boxed{\frac{\partial^2 L}{\partial w_{ij}^2} = \sum_{i=1}^{n}\left[\frac{-x_i x_j e^{-w^Tx_i}}{(1+e^{-w^Tx_i})^2}\right] + 2}$$

Note: The +2 is only on the diagonal terms where $i=j$.

This was

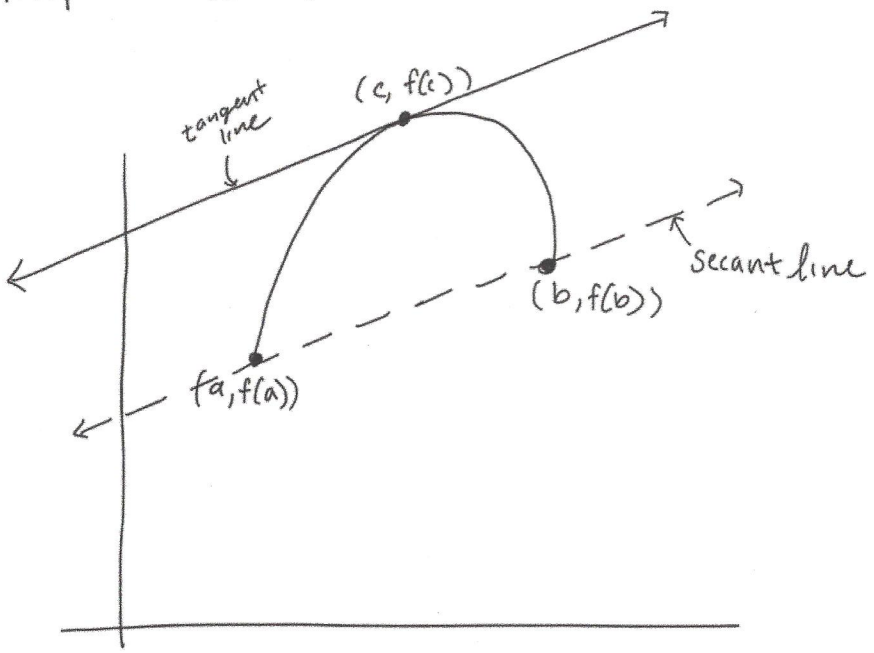
intentionally

left blank.

---

⑤ Give a geometric interpretation of the Mean Value Theorem.

Consider some function $f(x)$ that has points $(a, f(a))$ and $(b, f(b))$ as the 'end points' graphed like so:



Consider some point $(c, f(c))$ that lies in between $(a, f(a))$ and $(b, f(b))$ s.t. the tangent line at $(c, f(c))$ is parallel to the secant line formed between $(a, f(a))$ and $(b, f(b))$. The mean value theorem

says that this point $(c, f(c))$ is the only point these lines are parallel, and c is halfway between a and b.

---

(6) Consider the following function:

$$\ell(t_1, t_2) = \max(0, 1 - t_1 \cdot t_2),$$

where $t_1, t_2 \in \mathbb{R}$. This is known as the hinge loss which is used to train support vector machines (SVM). Given training data $(\vec{x}_i, y_i)$, $i = 1, \ldots, n$, if we assume a linear decision boundary for a classification problem, then we obtain the following objective function:

$$L(\vec{w}) = \sum_{i=1}^{n} \ell(y_i, \vec{w}^T x_i),$$

where $\vec{w}, \vec{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$. In general, the gradient is undefined for this function and thus gradient descent cannot be used to minimize it. However, its subdifferential is defined and thus sub-gradient descent can be used.

Compute the subdifferential of L.

Note: a regularization and offset term are typically included, which we are neglecting in this exercise. We will consider the full SVM problem later in the semester.

$$L(\vec{w}) = \sum_{i=1}^{n} \max\left(0, 1 - y_i \vec{w}^T \vec{x}_i\right)$$

then the subdifferential is defined as

$$\frac{\partial L_i}{\partial w} = \begin{cases} (\vec{0})^T & , \text{ for } 1 - y_i \vec{w}^T \vec{x}_i < 0 \\ -y_i \vec{x}_i^T & , \text{ for } 1 - y_i \vec{w}^T \vec{x}_i > 0 \\ \text{[illegible]} & , \text{ for} \end{cases}$$

This is the sub differential that is calculated at the intersection on the graph of the functions + represents all the functions whose slopes could be calculated at this point 0, but still be below the other two intersecting lines. Also, since $y \in \{-1, 1\}$ there are two options to choose from:

For each j^{th} component of $\vec{x}$, the sub differential is:

$$[0, -y_i \vec{x}_i(j)^T] \text{ or } [-y_i \vec{x}_i(j)^T, 0]$$

So the total answer for

$$\frac{\partial L_i}{\partial \vec{w}} = \begin{cases} (\vec{0})^T & , \text{ for } 1 - y_i \vec{w}^T \vec{x}_i < 0 \\ -y_i \vec{x}_i^T & , \text{ for } 1 - y_i \vec{w}^T \vec{x}_i > 0 \\ [0, -y_i \vec{x}_i(j)^T] \text{ or } [-y_i \vec{x}_i(j)^T] & , \text{ for } 1 - y_i \vec{w}^T \vec{x}_i = 0 \end{cases}$$

---

(7) State whether the following Properties and statements are true.

a) $e^? = O(n!)$, $n \in N$

b) $n^2 = O(n \log(n))$, $n \in N$

c) if $f = O(g) \Rightarrow kf = O(g)$, where $k < \infty$

d) if $f_1 = O(g)$ and $f_2 = O(h) \Rightarrow f_1 f_2 = O(gh)$

e) if $f_1 = O(g)$ and $f_2 = O(h) \Rightarrow f_1 + f_2 = O(\max(g, h))$

---

a) TRUE.

b) False.

c) TRUE.

d) TRUE.

HOMEWORK 2 QUESTION 2 CODE

```
import numpy as np
import inline as inline
import matplotlib as plt
from matplotlib import pyplot as plt
from mpl_toolkits import mplot3d

# Initialize x0 as a 1x2 vector using the point x0 = (1,1)
x0 = [1, 1]

# define f(x) = x_1^2 + x_2^2
def f(x):
        return x[0]**2 + x[1]**2

# define f'(x) = 2x_1 + 2x_2 (as each partial is added for the total derivative)
def fgrad(x):
    return 2*x[0] + 2*x[1]

# set N equal to the max iterations as specified
N = 200

# set gamma equal to one of the three specified step sizes
gamma = 0.0001 # 0.01,10

# define the gradientDescent function
def gradientDescent(f, fgrad, x0, N, gamma):
    # initialize the starting point
    pos = x0
    #initialize the values of the function
    fu_vals = [f(pos)]
    #for N iterations update and do the following
    for i in range(0,N):
        #find new value of the gradient
        update_function = fgrad(pos)
        #update the x0 values to new values
        for j in range(0,2):
            # take last position, subtract step size, and multiply by the new gradient
            pos[j] = pos[j] - gamma * update_function
        # add the current position to the fu_vals to be able to graph later.
        fu_vals.append(f(pos))
    # return the postion, the minimum value found(f(pos)) and the fu_vals.
    return (pos, f(pos), fu_vals)


ans = gradientDescent(f, fgrad, x0, N, gamma)

# To plot (gamma needs to change each time to get different plots)
c = []
for i in range(0,N+1):
    c.append(i)

plt.plot(ans[2], c)
plt.title("Gamma is 0.0001")#"(Gamma is 0.01)#("Gamma is 10)
plt.ylabel("Iterations")
plt.xlabel("Function values of Gradient Descent")
plt.show()
```

due to lack of ability to read the formatted code ↑
I've included Raw text versions as well.
The Raw text is what is displayed in the problem.

HOMEWORK 2 QUESTION 1 PLOTTER CODE

```
# This will plot the approximations. I did not plot them all together.
import math

import inline as inline
import matplotlib
from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt
#%matplotlib inline
#x = np.array()
#fig = plt.figure()
#ax = plt.axes(projection='3d')

def f(x, y):
    return np.cos(x * y) - (2 * (np.sin(2)) * (x - 1)) - (1 * (np.sin(2)) * (y - 2)) +
((1/2) * ((x - 1)**2) * (4 * np.cos(2))) + ((1/2) * (x - 1) * (y - 2) * 2 * np.cos(2)) -
(1/2) * ((y - 2)**2) * np.cos(2) + (1/2) * (y - 2) * (x - 1) * 2 * np.cos(2)
        #Second Order Approximation

        #return return np.cos(x * y) - (2 * (np.sin(2)) * (x - 1)) - (1 * (np.sin(2)) * (y
- 2))
        #First Order Approximation

        #return np.cos(xy)
        # Original expression


x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)

fig = plt.figure()
#ax = plt.axes(projection='3d')
#ax.contour3D(X, Y, Z, 50, cmap='binary')
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');

#def g(x):
        #10th order approx
        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) *
(4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0)
** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 *
np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x **
2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16 * np.exp(-x **
2) * x * (8 * x ** 6 - 84 * x ** 4 + 210 * x ** 2 - 105) * (x - 0) ** 7) +(16*np.exp(-
x**2)*(16*x**8-224*x**6 + 840*x**4 -840*x**2+105)*(x-0)**8)-(32*np.exp(-x**2)*(16*x**8-
288*x**6+1512*x**4-2520*x**2+945)*(x-0)**9)+(32*np.exp(-x**2)*(32*x**10-
720*x**8+5040*x**6-12600*x**4+9450*x**2-945)*(x-0)**10)

        #9th order approx
        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) *
(4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0)
** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 *
np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x **
```

```
2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16 * np.exp(-x **
2) * x * (8 * x ** 6 - 84 * x ** 4 + 210 * x ** 2 - 105) * (x - 0) ** 7) + (16 * np.exp(-
x ** 2) * (16 * x ** 8 - 224 * x ** 6 + 840 * x ** 4 - 840 * x ** 2 + 105) * (x - 0) **
8) - (32 * np.exp(-x ** 2) * (16 * x ** 8 - 288 * x ** 6 + 1512 * x ** 4 - 2520 * x ** 2
+ 945) * (x - 0) ** 9

        #8th order approx
        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) *
(4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0)
** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 *
np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x **
2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16 * np.exp(-x **
2) * x * (8 * x ** 6 - 84 * x ** 4 + 210 * x ** 2 - 105) * (x - 0) ** 7) +(16*np.exp(-
x**2)*(16*x**8-224*x**6 + 840*x**4 -840*x**2+105)*(x-0)**8)

        #7th order approx
        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) *
(4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0)
** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 *
np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x **
2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16*np.exp(-
x**2)*x*(8*x**6-84*x**4 + 210*x**2 -105)*(x-0)**7)

        #6th order approx
        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) *
(4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0)
** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 *
np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) +(8*np.exp(-
x**2)*(8*x**6-60*x**4+90*x**2-15)*(x-0)**6)

        #5th order approx
        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) *
(4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0)
** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) -(8*np.exp(-
x**2)*x*(4*x**4-20*x**2+15)*(x-0)**5)

        #4th order approx
        #return np.exp(-x**2)-(2*x*np.exp(-x**2)*(x-0))+(np.exp(-x**2)*(4*x**2-2)*(x-
0)**2)-4*np.exp(-x**2)*x*(2*x**2-3)*(x-0)**3+(4*np.exp(-x**2)*(4*x**4-12*x**2+3)*(x-
0)**4)

        #3rd order approx
        #return np.exp(-x**2)-(2*x*np.exp(-x**2)*(x-0))+(np.exp(-x**2)*(4*x**2-2)*(x-
0)**2) -4*np.exp(-x**2)*x*(2*x**2-3)*(x-0)**3

        #2nd order approx
        #return np.exp(-x**2)-(2*x*np.exp(-x**2)*(x-0))+(np.exp(-x**2)*(4*x**2-2)*(x-
0)**2)

        #1st order approx
        #return np.exp(-x**2) -(2*x*np.exp(-x**2)*(x-0))

        #Original function
        #return np.exp(-x**2)

#x = np.linspace ( -10,  # lower limit
#                  10,      # upper limit
#                  1510      # generate 151 points between -10 and 10
#                )
#y = g(x)     # This is already vectorized, that is, y will be a vector!
#plt.plot(x, y)
#plt.show()
plt.show()
```

HOMEWORK 2 QUESTION 2 CODE

```
import numpy as np
import inline as inline
import matplotlib as plt
from matplotlib import pyplot as plt
from mpl_toolkits import mplot3d

# Initialize x0 as a 1x2 vector using the point x0 = (1,1)
x0 = [1, 1]

# define f(x) = x_1^2 + x_2^2
def f(x):
    return x[0]**2 + x[1]**2

# define f'(x) = 2x_1 + 2x_2 (as each partial is added for the total derivative)
def fgrad(x):
  return 2*x[0] + 2*x[1]

# set N equal to the max iterations as specified
N = 200

# set gamma equal to one of the three specified step sizes
gamma = 0.0001 # 0.01,10

# define the gradientDescent function
def gradientDescent(f, fgrad, x0, N, gamma):
  # initialize the starting point
  pos = x0
  #initialize the values of the function
  fu_vals = [f(pos)]
  #for N iterations update and do the following
```

```
    for i in range(0,N):
        #find new value of the gradient
        update_function = fgrad(pos)
        #update the x0 values to new values
        for j in range(0,2):
            # take last position, subtract step size, and multiply by the new gradient
            pos[j] = pos[j] - gamma * update_function
        # add the current position to the fu_vals to be able to graph later.
        fu_vals.append(f(pos))
    # return the postion, the minimum value found(f(pos)) and the fu_vals.
    return (pos, f(pos), fu_vals)



ans = gradientDescent(f, fgrad, x0, N, gamma)

# To plot (gamma needs to change each time to get different plots)
c = []
for i in range(0,N+1):
    c.append(i)

plt.plot(ans[2], c)
plt.title("Gamma is 0.0001")#"(Gamma is 0.01)#("Gamma is 10)
plt.ylabel("Iterations")
plt.xlabel("Function values of Gradient Descent")
plt.show()
```

HOMEWORK 2 QUESTION 1 PLOTTER CODE

```
# This will plot the approximations. I did not plot them all together.

import math


import inline as inline

import matplotlib

from mpl_toolkits import mplot3d


import numpy as np

import matplotlib.pyplot as plt

#%matplotlib inline

#x = np.array()

#fig = plt.figure()

#ax = plt.axes(projection='3d')


def f(x, y):

   return np.cos(x * y) - (2 * (np.sin(2)) * (x - 1)) - (1 * (np.sin(2)) * (y - 2)) + ((1/2) * ((x - 1)**2) * (4 * np.cos(2))) + ((1/2) *
(x - 1) * (y - 2) * 2 * np.cos(2)) - (1/2) * ((y - 2)**2) * np.cos(2) + (1/2) * (y - 2) * (x - 1) * 2 * np.cos(2)

      #Second Order Approximation


         #return return np.cos(x * y) - (2 * (np.sin(2)) * (x - 1)) - (1 * (np.sin(2)) * (y - 2))

         #First Order Approximation


         #return np.cos(xy)

         # Original expression



x = np.linspace(-6, 6, 30)

y = np.linspace(-6, 6, 30)


X, Y = np.meshgrid(x, y)

Z = f(X, Y)
```

```
fig = plt.figure()
#ax = plt.axes(projection='3d')
#ax.contour3D(X, Y, Z, 50, cmap='binary')
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');


#def g(x):
    #10th order approx
    #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x ** 2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16 * np.exp(-x ** 2) * x * (8 * x ** 6 - 84 * x ** 4 + 210 * x ** 2 - 105) * (x - 0) ** 7) +(16*np.exp(-x**2)*(16*x**8-224*x**6 + 840*x**4 -840*x**2+105)*(x-0)**8)-(32*np.exp(-x**2)*(16*x**8-288*x**6+1512*x**4-2520*x**2+945)*(x-0)**9)+(32*np.exp(-x**2)*(32*x**10-720*x**8+5040*x**6-12600*x**4+9450*x**2-945)*(x-0)**10)

    #9th order approx
    #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x ** 2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16 * np.exp(-x ** 2) * x * (8 * x ** 6 - 84 * x ** 4 + 210 * x ** 2 - 105) * (x - 0) ** 7) + (16 * np.exp(-x ** 2) * (16 * x ** 8 - 224 * x ** 6 + 840 * x ** 4 - 840 * x ** 2 + 105) * (x - 0) ** 8) - (32 * np.exp(-x ** 2) * (16 * x ** 8 - 288 * x ** 6 + 1512 * x ** 4 - 2520 * x ** 2 + 945) * (x - 0) ** 9

    #8th order approx
    #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x ** 2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16 * np.exp(-x ** 2) * x * (8 * x ** 6 - 84 * x ** 4 + 210 * x ** 2 - 105) * (x - 0) ** 7) +(16*np.exp(-x**2)*(16*x**8-224*x**6 + 840*x**4 -840*x**2+105)*(x-0)**8)

    #7th order approx
```

```
        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) + (8 * np.exp(-x ** 2) * (8 * x ** 6 - 60 * x ** 4 + 90 * x ** 2 - 15) * (x - 0) ** 6) - (16*np.exp(-x**2)*x*(8*x**6-84*x**4 + 210*x**2 -105)*(x-0)**7)


        #6th order approx

        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8 * np.exp(-x ** 2) * x * (4 * x ** 4 - 20 * x ** 2 + 15) * (x - 0) ** 5) +(8*np.exp(-x**2)*(8*x**6-60*x**4+90*x**2-15)*(x-0)**6)


        #5th order approx

        #return np.exp(-x ** 2) - (2 * x * np.exp(-x ** 2) * (x - 0)) + (np.exp(-x ** 2) * (4 * x ** 2 - 2) * (x - 0) ** 2) - 4 * np.exp(-x ** 2) * x * (2 * x ** 2 - 3) * (x - 0) ** 3 + (4 * np.exp(-x ** 2) * (4 * x ** 4 - 12 * x ** 2 + 3) * (x - 0) ** 4) - (8*np.exp(-x**2)*x*(4*x**4-20*x**2+15)*(x-0)**5)


        #4th order approx

        #return np.exp(-x**2)-(2*x*np.exp(-x**2)*(x-0))+(np.exp(-x**2)*(4*x**2-2)*(x-0)**2)-4*np.exp(-x**2)*x*(2*x**2-3)*(x-0)**3+(4*np.exp(-x**2)*(4*x**4-12*x**2+3)*(x-0)**4)


        #3rd order approx

        #return np.exp(-x**2)-(2*x*np.exp(-x**2)*(x-0))+(np.exp(-x**2)*(4*x**2-2)*(x-0)**2) -4*np.exp(-x**2)*x*(2*x**2-3)*(x-0)**3


        #2nd order approx

        #return np.exp(-x**2)-(2*x*np.exp(-x**2)*(x-0))+(np.exp(-x**2)*(4*x**2-2)*(x-0)**2)


        #1st order approx

        #return np.exp(-x**2) -(2*x*np.exp(-x**2)*(x-0))


        #Original function

        #return np.exp(-x**2)


#x = np.linspace ( -10,  # lower limit

#               10,     # upper limit
```

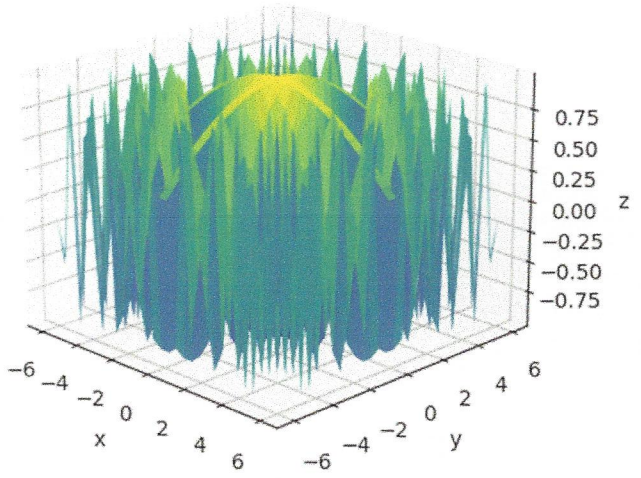
```
#              1510     # generate 151 points between -10 and 10
#           )
#y = g(x)    # This is already vectorized, that is, y will be a vector!
#plt.plot(x, y)
#plt.show()
plt.show()
```

*Note: to run g(x) the comment with the approximation order may need to follow the code to work properly. I ran it both ways, and had more success without the comment, or with it following the code. I put it ahead of the approximations for g(x), part 1d, to make it easier to read. f(x,y), 1b has it following the code for the approximation.

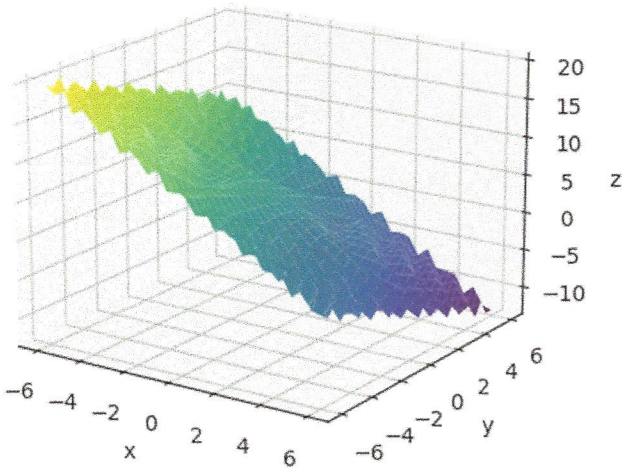
*Note: to run g(x) the comment with the approximation order may need to follow the code to work properly. I ran it both ways, and had more success without the comment, or with it following the code. I put it ahead of the approximations for g(x), part 1d, to make it easier to read. f(x,y), 1b has it following the code for the approximation.

1 b) graphs

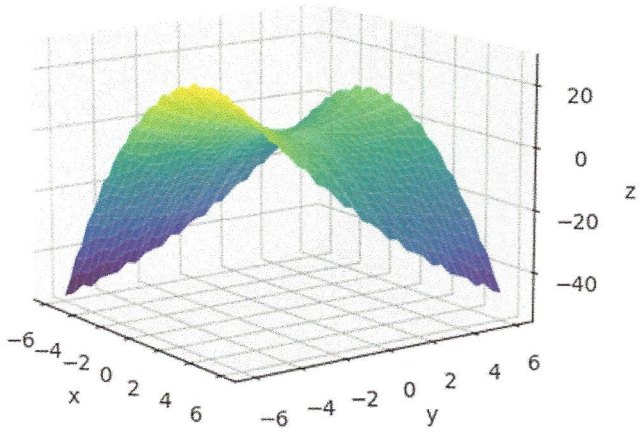
Original



First Order Approximation

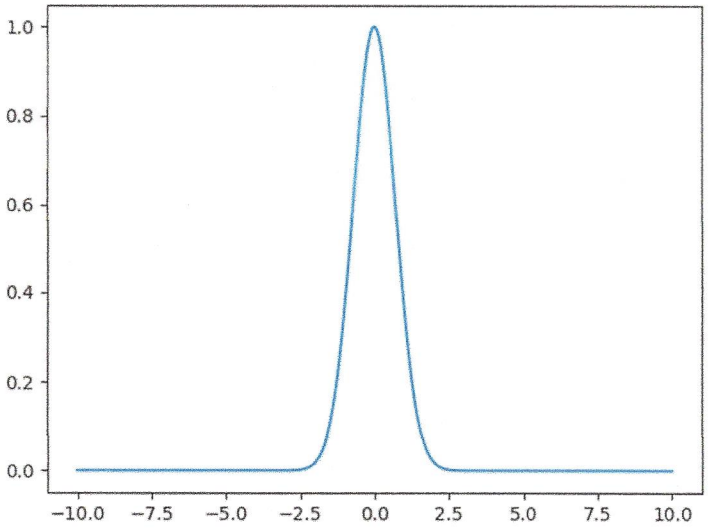


Second Order Approximation

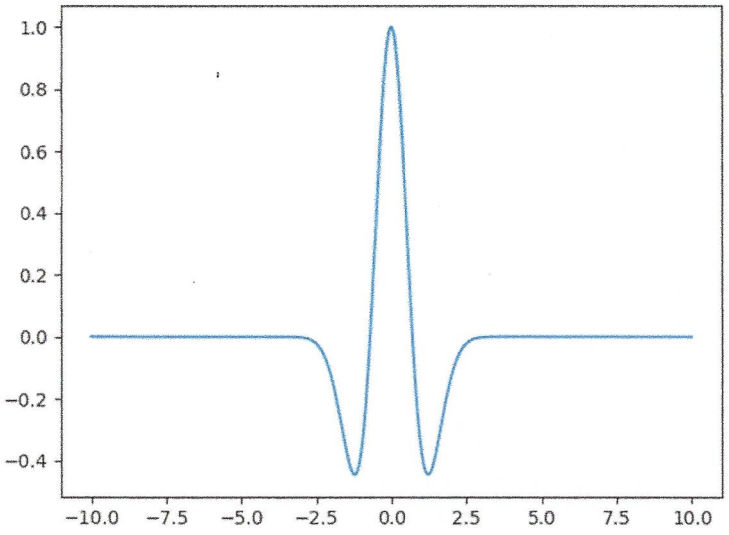


1 d) Graphs

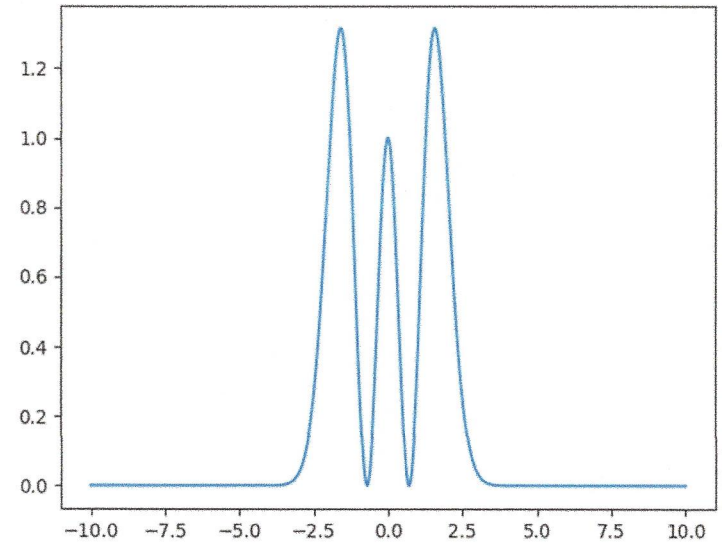
Original



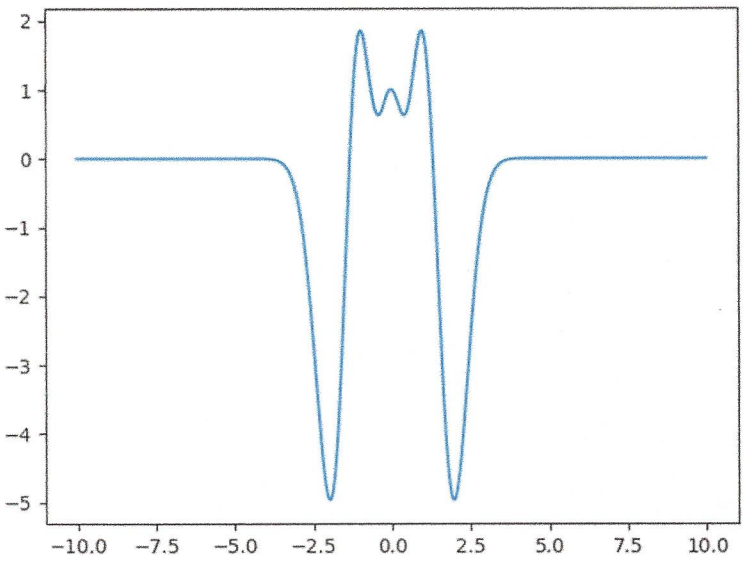
Dx1


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

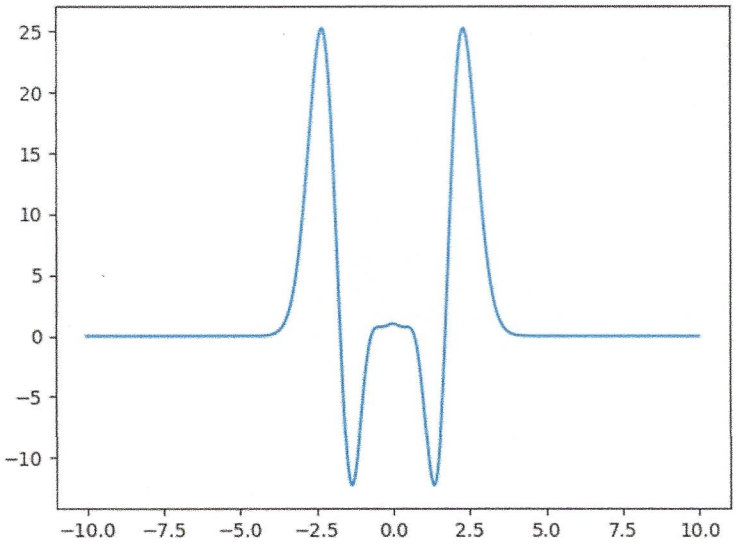
Dx2


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

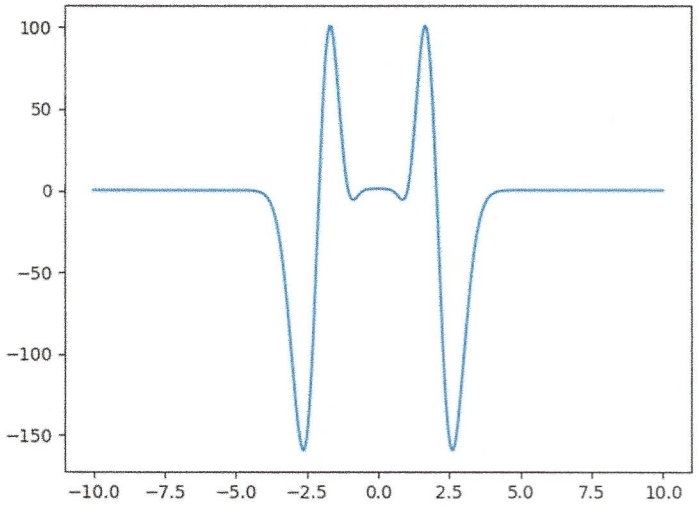
Dx3


2
1
0
−1
−2
−3
−4
−5
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

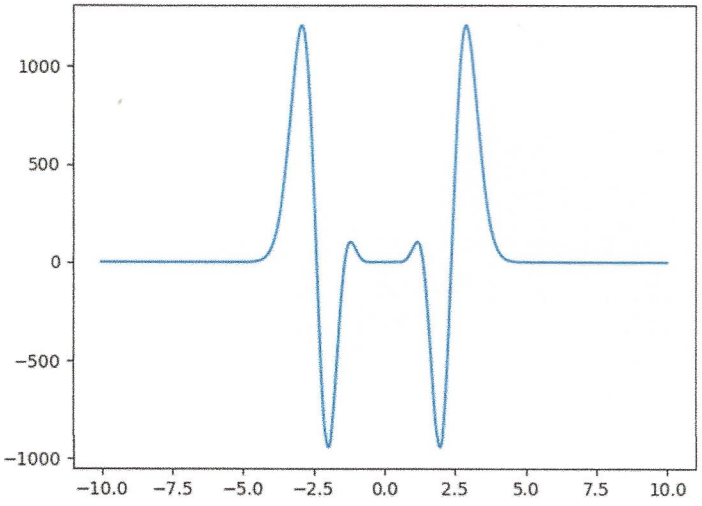
Dx4


25
20
15
10
5
0
−5
−10
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Dx5

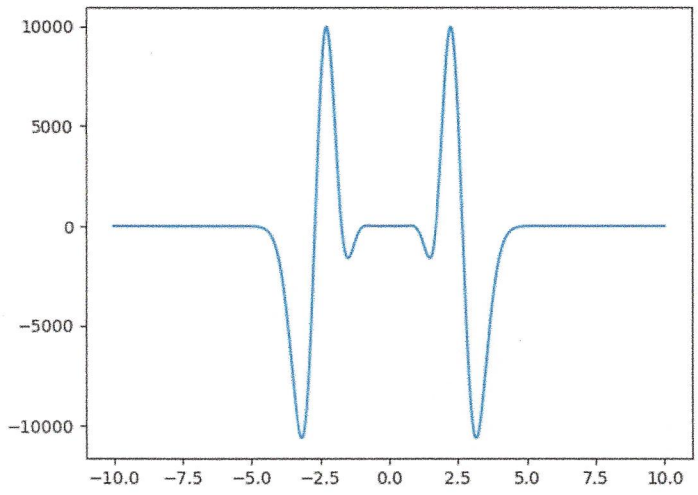

100
50
0
−50
−100
−150
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Dx6

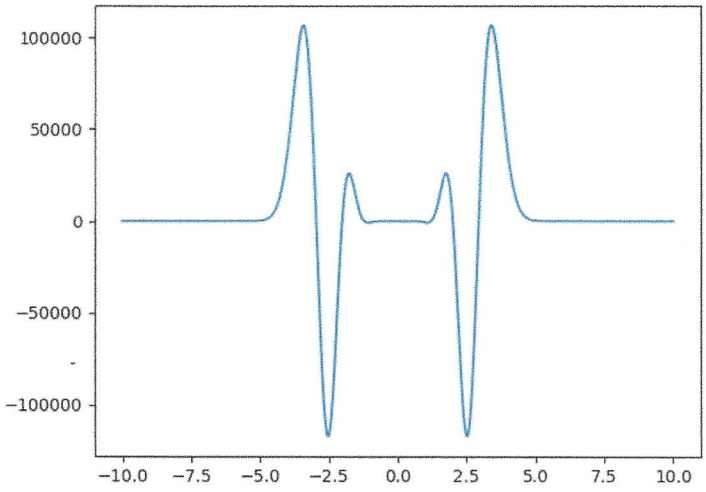

1000
500
0
−500
−1000
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Dx7

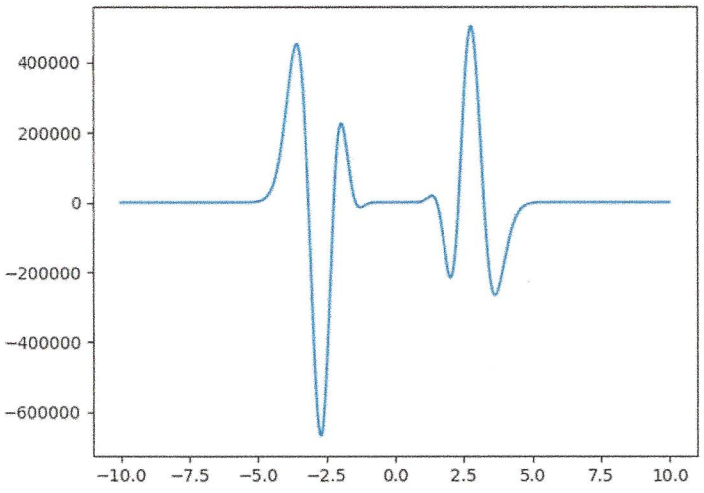
Dx7


10000
5000
0
−5000
−10000
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

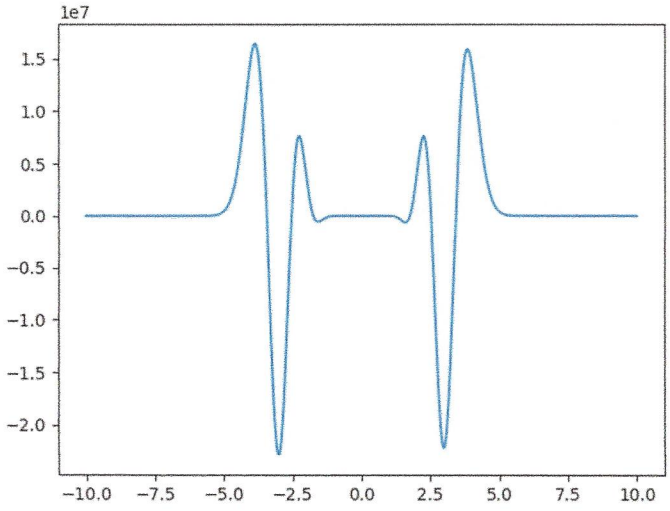
Dx8


100000
50000
0
−50000
−100000
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

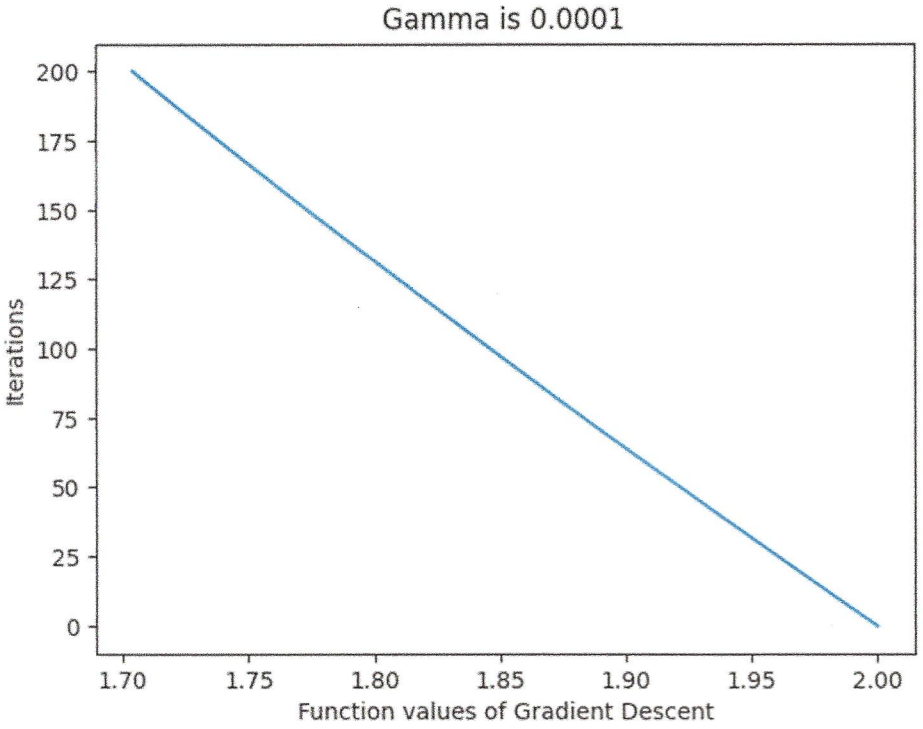
Dx9

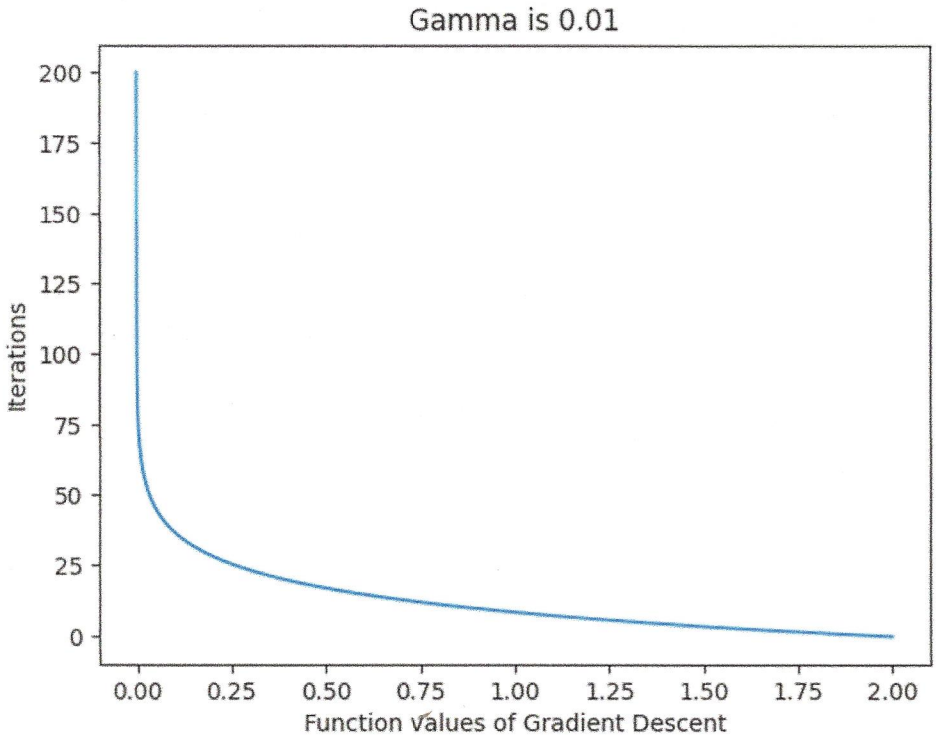

400000
200000
0
−200000
−400000
−600000
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

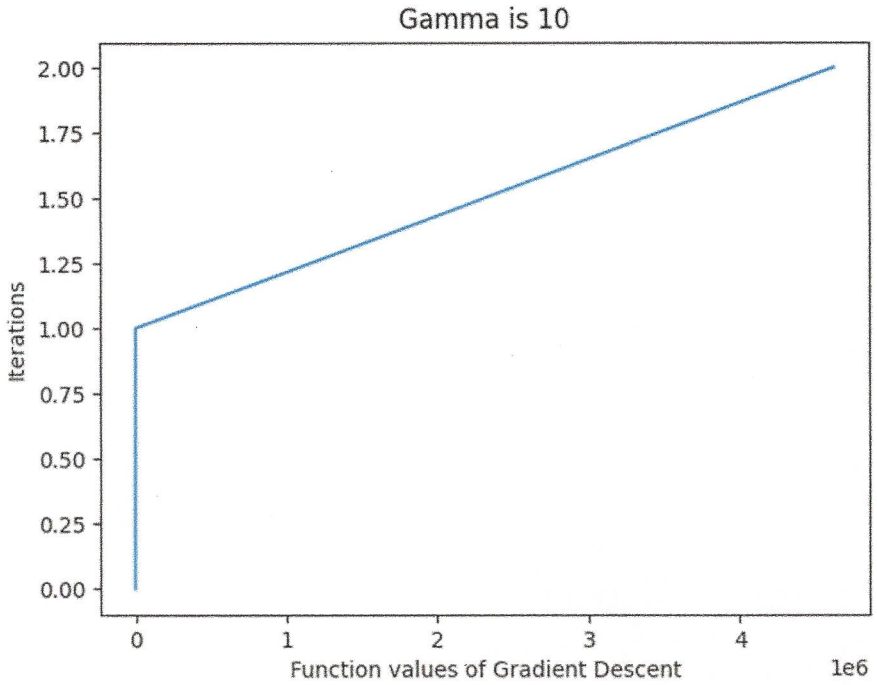
Dx10


1e7
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

2) Gamma graphs






Gamma is 10
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Iterations
0
1
2
3
4
Function values of Gradient Descent
1e6