Optimisation Algorithms for Data Analysis Week 4 Assignment

Jiaming Deng 22302794

Function

The two functions that are used in this assignment are:

$$f_1(x, y) = 3(x-3)^4 + 9(y-8)^2$$
$$f_2(x, y) = \mathrm{Max}(x-3, 0) + 9 \bullet |y-8|$$

I use sympy to initialise the two variables x and y, define two equations based on the above functions and use the diff() function to get the partial derivatives of the two functions.

$$\frac{\partial f_1}{\partial x} = 12(x-3)^3, \ \frac{\partial f_1}{\partial y} = 18y - 144$$

$$\frac{\partial f_2}{\partial x} = \theta(x-3), \ \frac{\partial f_2}{\partial y} = 9 \bullet \mathrm{sign}(y-2)$$

(a) Firstly I implemented each of the following four gradient descent algorithms, Polyak step size, RMSProp, Heavy Ball and Adam.

1. Polyak step size

In each iteration, a constant step is calculated by taking the value of the function for a given parameter value and dividing it by the sum of the squares of the partial derivatives of each parameter. To prevent division by zero, a very small epsilon is added to the denominator.

```
for _ in range(300):
    cnt = 0
    for i in range(n):
        cnt = sum(df[i](x[i]) ** 2)
    step = f(*x) / (cnt + epsilon)
    for i in range(n):
        x[i] -= step * df[i](x[i])
    x_list.append(deepcopy(x))
    f_list.append(f(*x))
    step_list.append(step)
```

2. RMSProp

The formula for the RMSProp gradient descent algorithm is as follows. Firstly f_list and step_list are initialised with two empty arrays containing the step size and the sum of the historical squared gradients for each parameter respectively. The step length is initially set to the value of the alpha0 parameter. At each iteration, the parameter values are first subtracted by the product of the initial step length and the parameter gradient. The squared beta parameter of the historical gradient is then updated by adding the current square of the parametric gradient to the sum. The step size is then updated by dividing the constant alpha0 value by the square root of the sum of the current squared gradients.

$$\alpha_t = \frac{\alpha_0}{\sqrt{(1-\beta)\beta^t \frac{df}{dx}(x_0)^2 + \ldots + (1-\beta)\frac{df}{dx}(x_0)^2 + \varepsilon}}$$

```
for _ in range(300):
    for i in range(n):
        x[i] -= alphas[i] * df[i](x[i])
        sums[i] = (beta * sums[i]) + ((1 - beta) * (df[i](x[i]) ** 2))
        alphas[i] = alpha0 / ((sums[i] ** 0.5) + epsilon)
    x_list.append(deepcopy(x))
    f_list.append(f(*x))
    step_list.append(deepcopy(alphas))
```

3. Heavy Ball

The formula for the Heavy Ball gradient descent algorithm is as follows. Z is the sum of the historical squared gradients and is set to zero in the initialisation. In each iteration, the sum of the historical squared gradients is updated, with the beta parameter determining the weight of the historical sum and the alpha parameter determining the weight of the current sum. Only one step is used and the sum of squared gradients includes each parameter. Each parameter is then subtracted by the product of the new step size Z and its gradient.

$$Z_{t+1} = \beta Z_t + \alpha \nabla f(X_t)$$

```
for _ in range(num_iters):
    cnt = 0
    for i in range(n):
        cnt = sum(df[i](x[i])**2)
    z = (beta * z) + (alpha * f(*x) / (cnt + epsilon))
    for i in range(n):
        x[i] -= z * df[i](x[i])
    x_list.append(deepcopy(x))
    f_list.append(f(*x))
    step_list.append(z)
```

4. Adam

The Adam algorithm is equivalent to the combination of the RMSprop and Heavy Ball algorithms, with the following formula. Firstly f_list and step_list are initialised with two empty arrays containing the step size and the sum of the historical squared gradients for each parameter respectively. The iteration counter t is also set to zero. In each iteration, the counter t is incremented by 1. The parameter beta_1 determines the weight of the regular gradient and the parameter beta_2 determines the weight of the square gradient. Finally, each parameter is subtracted by the product of the alpha parameter and the square root of the sum of the scaled normal gradients divided by the sum of the scaled normal gradients.

$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla f(x_t)$$

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)[\frac{\partial f}{\partial x_1}(x_t)^2, \frac{\partial f}{\partial x_1}(x_t)^2, \ldots, \frac{\partial f}{\partial x_n}(x_t)^2]$$

$$m' = \frac{m_{t+1}}{(1-\beta_2^t)} \quad v' = \frac{v_{t+1}}{(1-\beta_2^t)}$$

$$x_{t+1} = x_t - \alpha[\frac{m'_1}{\sqrt{v'_1}+\varepsilon}, \frac{m'_2}{\sqrt{v'_2}+\varepsilon}, \ldots, \frac{m'_n}{\sqrt{v'_n}+\varepsilon}]$$

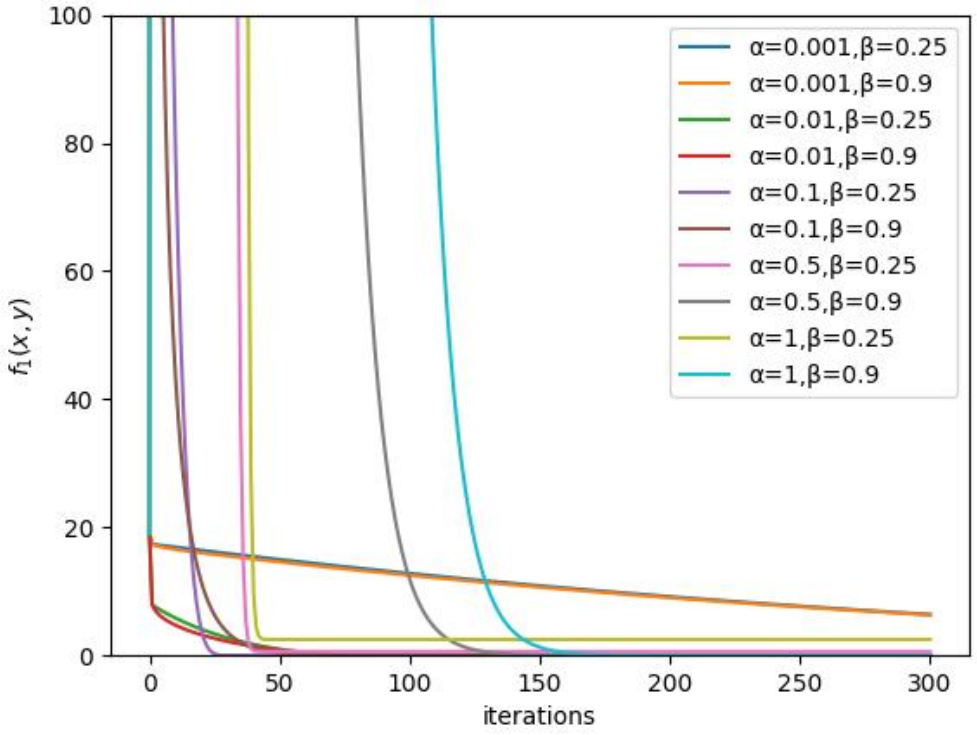
```
for _ in range(300):
    t += 1
    for i in range(n):
        ms[i] = (beta1 * ms[i]) + ((1 - beta1) * df[i](x[i]))
        vs[i] = (beta2 * vs[i]) + ((1 - beta2) * (df[i](x[i]) ** 2))
        m_hat = ms[i] / (1 - (beta1 ** t))
        v_hat = vs[i] / (1 - (beta2 ** t))
        step[i] = alpha * (m_hat / ((v_hat ** 0.5) + epsilon))
        x[i] -= step[i]
    x_list.append(deepcopy(x))
    f_list.append(f(*x))
    step_list.append(deepcopy(step))
```

(b)

(i) RMSProp

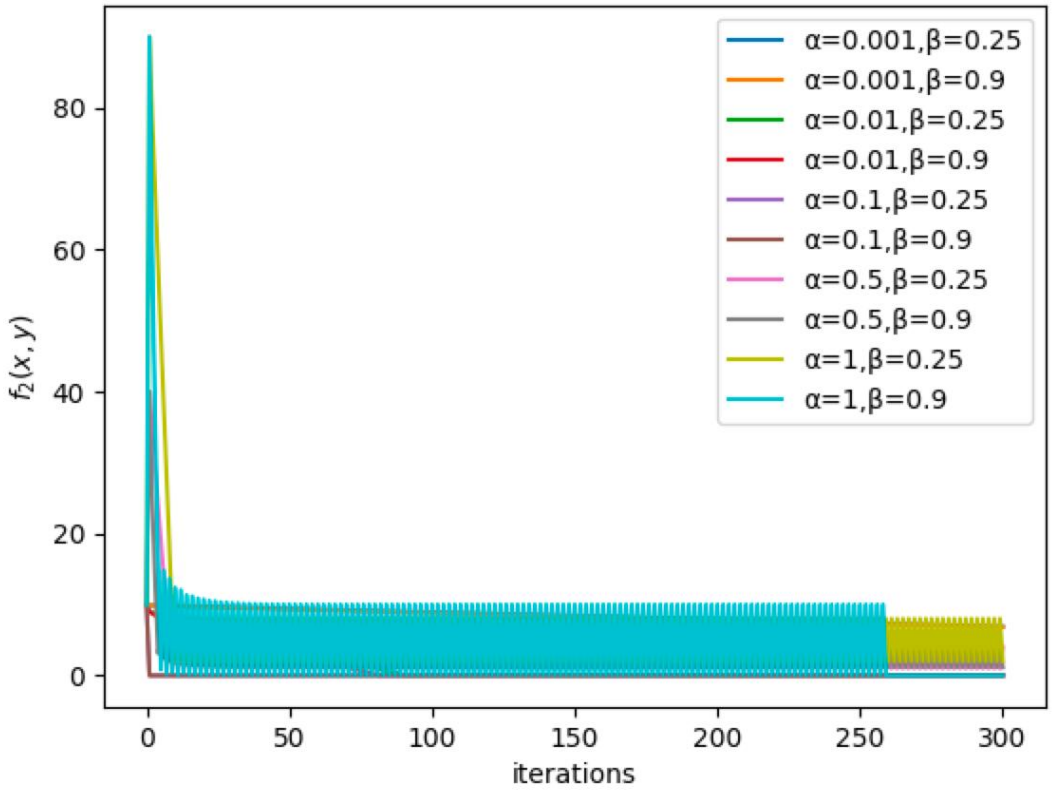
Firstly I set the value of the parameter alpha to [0.001, 0.01, 0.1, 0.5, 1], the value of the parameter beta to [0.25, 0.9], the value of X0 to 4, the value of Y0 to 1 and the number of iterations to 300.

For the function1, I observed the image and found that the function converged at nearly 50 iterations when alpha was equal to 0.01, so this alpha was the best. The two curves are almost stacked when the beta is 0.9 or 0.25 and the alpha is equal to 0.1. It is not clear from this which is the more appropriate choice of beta, but by looking at alpha equal to 0.5 and 1, I can see that the curve converges much faster when the beta is equal to 0.25, so a beta of 0.25 is better.



For the function2, when the alpha is 0.01, the function has converged to close to 0 in the first few iterations, at around 0.02. When the alpha is 0.5 and 1, the function fluctuates between 3 and 15, although it drops quickly at the beginning, which is a

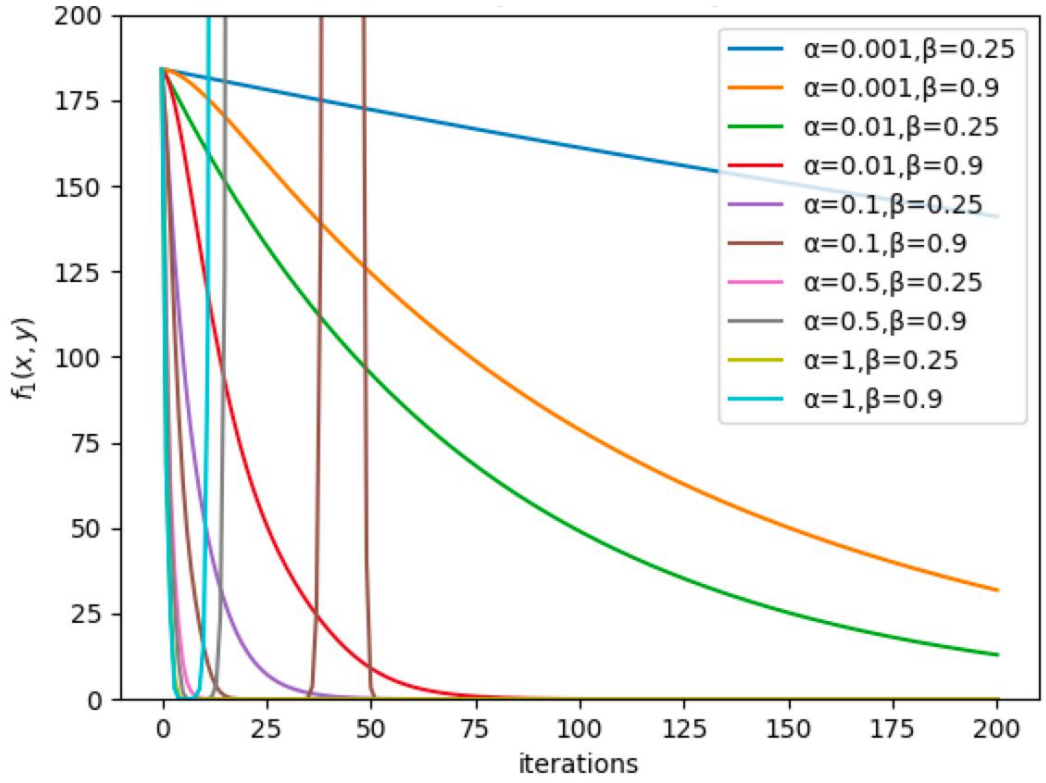
poor result. Similarly at beta equal to 0.25 the function converges fastest for each choice of alpha. So as above, both have the best results at an alpha of 0.01 and a beta of 0.25.



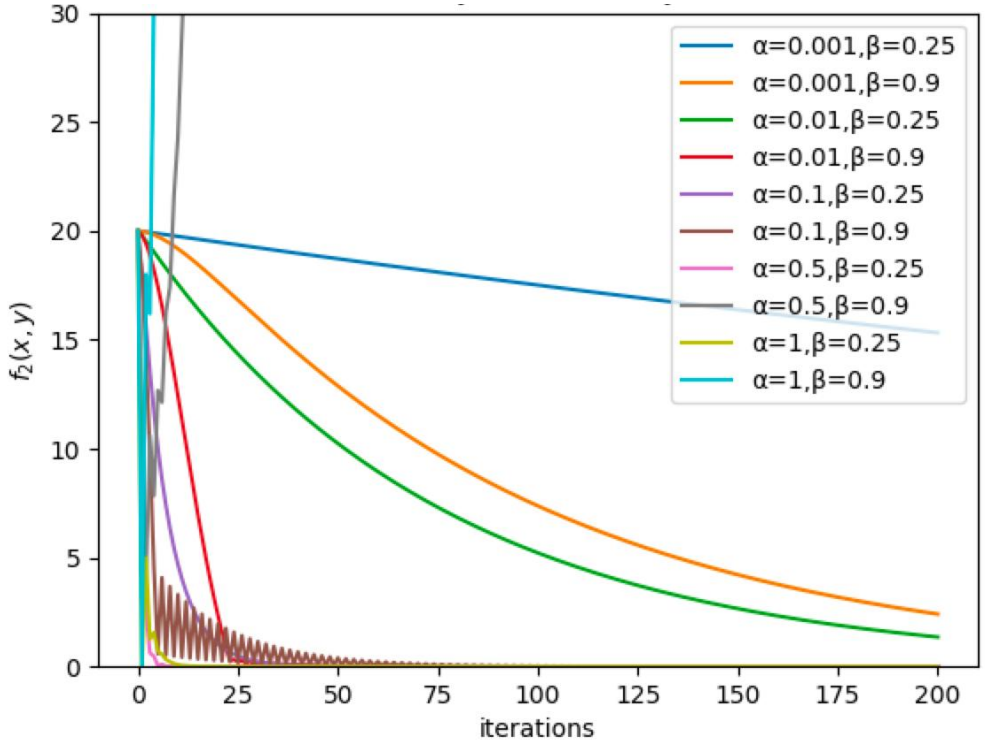
(ii) Heavy Ball

Firstly I set the value of the parameter alpha to [0.001, 0.01, 0.1, 0.5, 1], the value of the parameter beta to [0.25, 0.9], the value of X0 to 4, the value of Y0 to 1 and the number of iterations to 200.

For the function1, when alpha is equal to 0.001, the function curve falls slowly and does not converge even after 200 iterations, when alpha is equal to 0.01 and beta is 0.25 it does not converge even after 200 iterations, but converges after 80 iterations when beta is 0.9. When alpha is equal to 0.1, 0.5, 1, 0.25 or 0.9 the function curve converges, but when alpha is 1 and beta is 0.9, it converges in about 5 iterations and is the fastest, so the alpha and beta in this case are optimal.



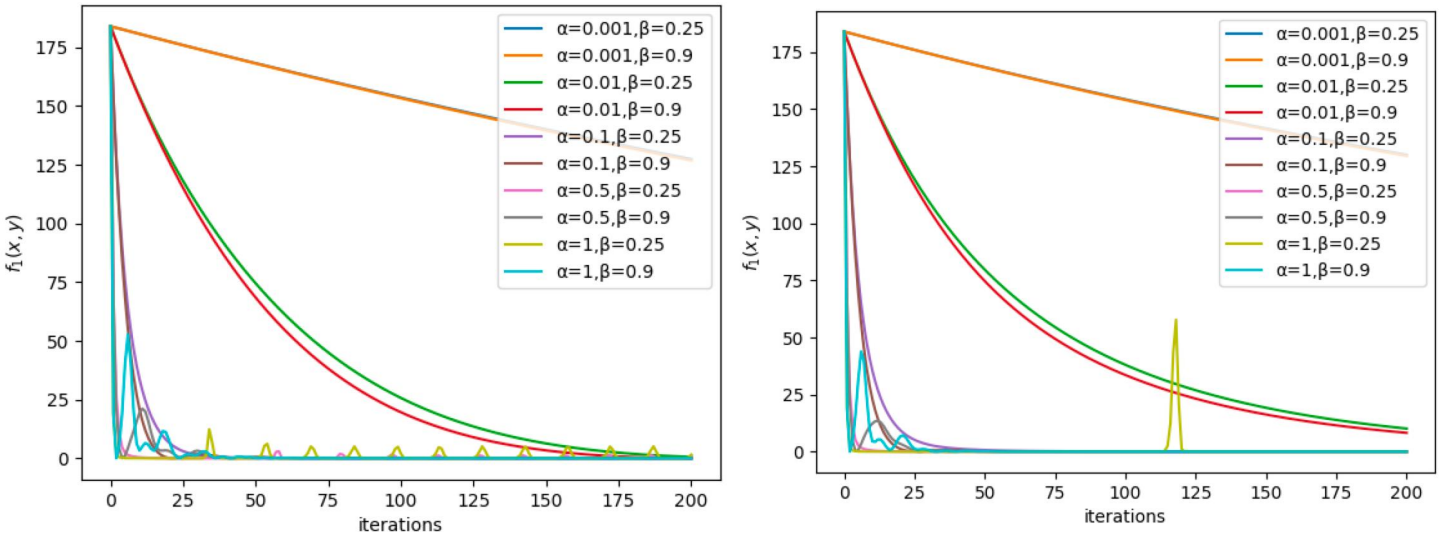
As with function1, the function 2 does not drop to 0 at 200 iterations when alpha is 0.001 and when alpha is 0.01 and beta is 0.25. At alpha of 0.01 and beta of 0.9, the function converges to 0, but there are about 25 iterations. When alpha is 0.1, the function converges in about 75 iterations after oscillation. When alpha is 1 and beta is 0.9, the function falls to 0 fastest, so this is optimal.



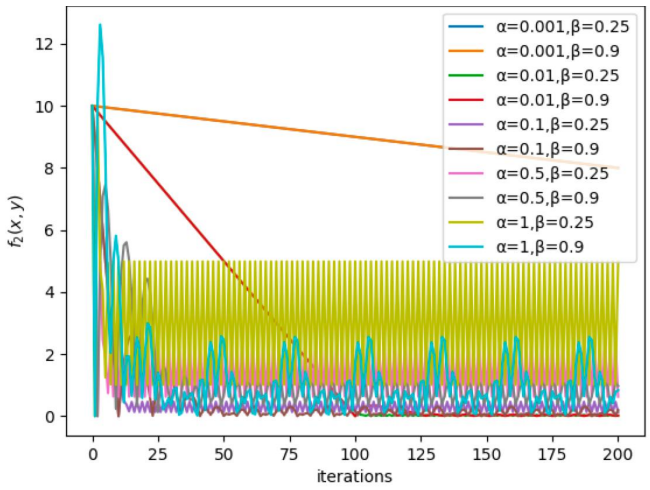
(iii) Adam

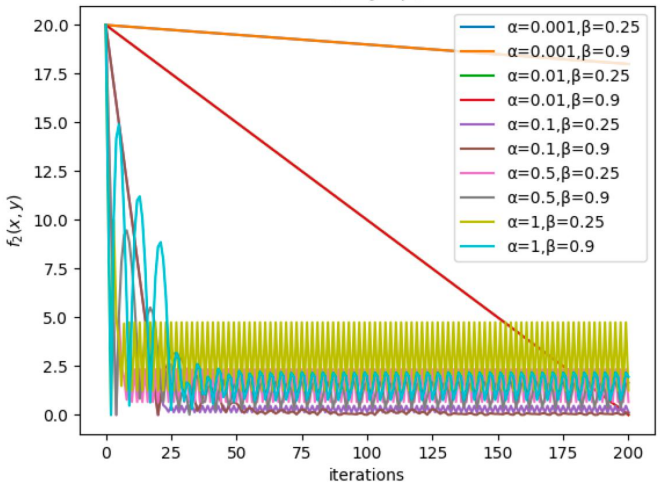
Firstly I set the value of the parameter alpha to [0.001, 0.01, 0.1, 0.5, 1], the value of the parameter beta1 to [0.25, 0.9], beta2 to [0.9, 1]the value of X0 to 4, the value of Y0 to 1 and the number of iterations to 200. The left panel below shows beta2 = 0.9 and the right panel shows beta2 = 1.

For function1, firstly, fixing beta2 at 0.9, the function curve falls very slowly and does not converge until 200 iterations when alpha equals 0.001, and does not converge to 0 until 200 iterations when alpha equals 0.01. The curve falls very slowly when alpha equals 0.1. When alpha equals 0.1, the function converges in all cases, but beta1=0.9 performs better than beta1=0.25 and beta2=0.9 performs better than beta2=1. When alpha is 1, beta1=0.9 or 0.25 behaves the same, but the function fluctuates less with beta2=1.



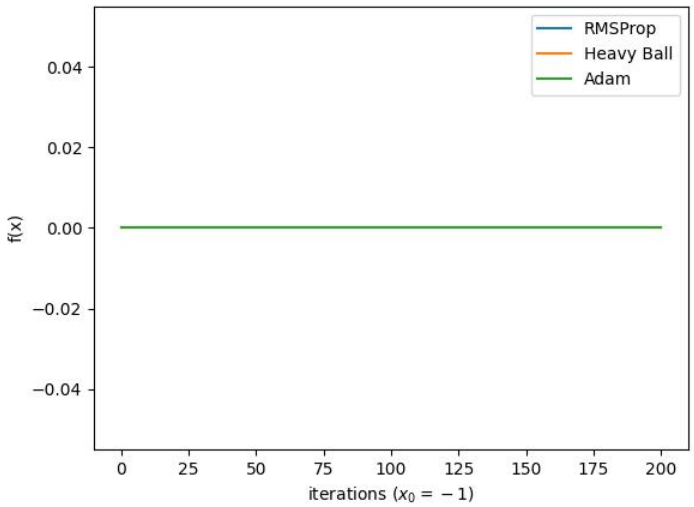
For function2, for beta2 = 0.9 or 1, the images did not show a significant difference. When alpha equals 0.001 and beta1 equals 0.9, the function still does not converge up to 200 iterations. When alpha equals 0.001 and beta1 equals 0.25, and when alpha equals 0.01, the function converges to 0 in less than 200 iterations. when alpha equals 0.1, the function converges to 0 in 25 iterations. in all other cases, the function does not converge, showing constant fluctuations between 0 and 5. All in all when alpha=0.1 and beta1=0.25 and beta2=0.9, the result is the best.





(c) Firstly, based on the results from (b), for function2, the parameter chosen is below. alpha=0.01, beta=0.25 for RMSProp; alpha=1, beta=0.9 for Heavy Ball; alpha=0.1, beta1=0.25, beta2=0.9 for Adam. The iteration for all algorithms is 200 times.

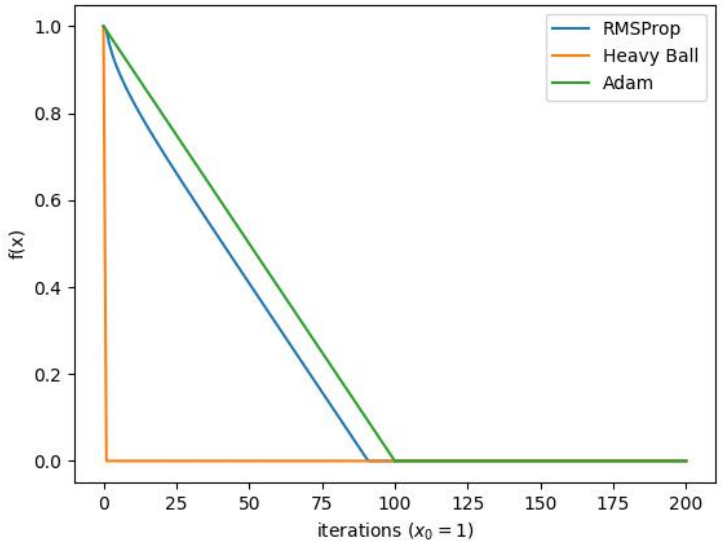
(i) When x_0 = -1, all functions converge at 0 after 1 round of iterations. There are 2 reasons. First, since f(x) = Max(0, x), f(-1) = Max(0,-1) when x = -1. Therefore f(-1) = 0 and the function is already at its minimum value of 0. Second, the derivative of θ(-1) is 0.



(ii) When X0 = +1, all functions received 0. The Heavy Ball algorithm took 3 iterations, the RMSProp algorithm took 90 iterations, and the Adam algorithm took 98 iterations.

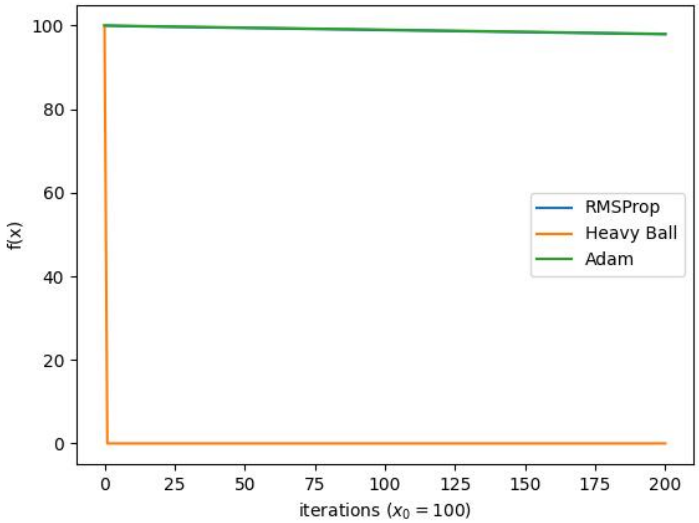
For the Heavy Ball algorithm, the value of z after the first iteration is $\frac{100}{1+\varepsilon}$. This value is multiplied by gradient(θ(1)=1) and then subtracted from x0. Since epsilon is small, it can be seen that z = 1, so the result is 0.

For both the RMSProp and Adam algorithms, the function decreases at a steady rate, since gredient is a constant. RMSProp decreases a little faster than Adam at the beginning, since RMSProp starts with a custom alpha0 and the alpha after that is derived from the calculation.



(iii) When X0 = +100 only the Heavy Ball algorithm converges to 0 after very few iterations. because x = x - sxxx in the first iteration, which is almost directly 0 because epsilon is very small and sxxx is close to 100.

(iv) For the RMSProp and Adam algorithms, since only gradient has an effect on the rate of convergence, and gradient is a fixed value of 1, both algorithms converge, except that a large number of iterations are required when raising x to 100.



Appendix

```
1. import sympy as sp
```

```
from copy import deepcopy
import numpy as np
import matplotlib

matplotlib.use('TkAgg')
import matplotlib.pyplot as plt

def function_derivatives():
    x, y = sp.symbols('x y', real=True)
    f1 = 3 * ((x - 3) ** 4) + 9 * ((y - 8) ** 2)
    df1_x = sp.diff(f1, x)
    df1_y = sp.diff(f1, y)
    print(f1)
    print(df1_x)
    print(df1_y)
    f2 = sp.Max(x - 3, 0) + 9 * sp.Abs(y - 2)
    df2_x = sp.diff(f2, x)
    df2_y = sp.diff(f2, y)
    print(f2)
    print(df2_x)
    print(df2_y)

def polyak(x0, f, df, epsilon=1e-8):
    x = deepcopy(x0)
    n = len(df)
    x_list, f_list, step_list = [deepcopy(x)], [f(*x)], []
    for _ in range(300):
        cnt = 0
        for i in range(n):
            cnt = sum(df[i](x[i]) ** 2)
        step = f(*x) / (cnt + epsilon)
        for i in range(n):
            x[i] -= step * df[i](x[i])
        x_list.append(deepcopy(x))
        f_list.append(f(*x))
        step_list.append(step)
    return x_list, f_list, step_list


def RMSprop(x0, f, df, alpha0, beta):
    x = deepcopy(x0)
    n = len(df)
```

```
    x_list, f_list, step_list = [deepcopy(x)], [f(*x)], []
    epsilon = 1e-8
    sums = [0] * n
    alphas = [alpha0] * n
    for _ in range(300):
        for i in range(n):
            x[i] -= alphas[i] * df[i](x[i])
            sums[i] = (beta * sums[i]) + ((1 - beta) * (df[i](x[i]) ** 2))
            alphas[i] = alpha0 / ((sums[i] ** 0.5) + epsilon)
        x_list.append(deepcopy(x))
        f_list.append(f(*x))
        step_list.append(deepcopy(alphas))
    return x_list, f_list, step_list


def Heavy_Ball(x0, f, df, alpha, beta):
    x = deepcopy(x0)
    n = len(df)
    x_list, f_list, step_list = [deepcopy(x)], [f(*x)], [0]
    epsilon = 1e-8
    z = 0
    for _ in range(300):
        cnt = 0
        for i in range(n):
            cnt = sum(df[i](x[i])**2)
        z = (beta * z) + (alpha * f(*x) / (cnt + epsilon))
        for i in range(n):
            x[i] -= z * df[i](x[i])
        x_list.append(deepcopy(x))
        f_list.append(f(*x))
        step_list.append(z)
    return x_list, f_list, step_list


def adam(x0, f, df, alpha, beta1, beta2):
    x = deepcopy(x0)
    n = len(df)
    x_list, f_list, step_list = [deepcopy(x)], [f(*x)], [[0] * n]
```

```
    epsilon = 1e-8
    ms = [0] * n
    vs = [0] * n
    step = [0] * n
    t = 0
    for _ in range(300):
        t += 1
        for i in range(n):
            ms[i] = (beta1 * ms[i]) + ((1 - beta1) * df[i](x[i]))
            vs[i] = (beta2 * vs[i]) + ((1 - beta2) * (df[i](x[i]) ** 2))
            m_hat = ms[i] / (1 - (beta1 ** t))
            v_hat = vs[i] / (1 - (beta2 ** t))
            step[i] = alpha * (m_hat / ((v_hat ** 0.5) + epsilon))
            x[i] -= step[i]
        x_list.append(deepcopy(x))
        f_list.append(f(*x))
        step_list.append(deepcopy(step))
    return x_list, f_list, step_list
```