Name: Karan Dua Course Code: CS7DS2-202223 OPTIMISATION ALGORITHMS FOR DATA ANALYSIS
Student Id: 21331391

Question 1: Download functions from link and obtain function expression and derivative for using Sympy

I have downloaded the functions from the link provided. I have used Sympy library to implement these functions and obtain its derivatives with respect to **x** and **y**. Following are the two functions that I have to implement

Function 1: 1*(x-2) ^4 + 6*(y-9) ^2

```
class Function1 :
    def get_quadratic_equation_symbolic(self) :
        x = sm.symbols('x', real = True)
        y = sm.symbols('y', real = True)
        f = (1 * ((x - 2) ** 4)) + (6 * ((y - 9) ** 2))
        return f

    def get_quadratic_equation_derivative_wrt_x_symbolic(self) :
        x = sm.symbols('x', real = True)
        f = self.get_quadratic_equation_symbolic()
        derivative_x = sm.diff(f, x)
        return derivative_x

    def get_quadratic_equation_derivative_wrt_y_symbolic(self) :
        y = sm.symbols('y', real = True)
        f = self.get_quadratic_equation_symbolic()
        derivative_y = sm.diff(f, y)
        return derivative_y
```

Input:

```
function1 = Function1()
f = function1.get_quadratic_equation_symbolic()
derivative_df_dx = function1.get_quadratic_equation_derivative_wrt_x_symbolic()
derivative_df_dy = function1.get_quadratic_equation_derivative_wrt_y_symbolic()
```

Output:

```
Quadratic equation for Function 1 is: (x - 2)**4 + 6*(y - 9)**2
Derivative for f = (x - 2)**4 + 6*(y - 9)**2 with respect to x is: 4*(x - 2)**3
Derivative for f = (x - 2)**4 + 6*(y - 9)**2 with respect to y is: 12*y - 108
```

Function 2: Max(x-2,0) + 6*|y-9|

```
class Function2 :
    def get_quadratic_equation_symbolic(self) :
        x = sm.symbols('x', real = True)
        y = sm.symbols('y', real = True)
        f = (sm.Max(x - 2, 0)) + (6 * (sm.Abs(y - 9)))
        return f

    def get_quadratic_equation_derivative_wrt_x_symbolic(self) :
        x = sm.symbols('x', real = True)
        f = self.get_quadratic_equation_symbolic()
        derivative_x = sm.diff(f, x)
        return derivative_x

    def get_quadratic_equation_derivative_wrt_y_symbolic(self) :
        y = sm.symbols('y', real = True)
        f = self.get_quadratic_equation_symbolic()
        derivative_y = sm.diff(f, y)
        return derivative_y
```

Input:

```
function2 = Function2()
f = function2.get_quadratic_equation_symbolic()
derivative_df_dx = function2.get_quadratic_equation_derivative_wrt_x_symbolic()
derivative_df_dy = function2.get_quadratic_equation_derivative_wrt_y_symbolic()
```

Output:

```
Quadratic equation for Function 2 is: 6*Abs(y - 9) + Max(0, x - 2)
Derivative for f = 6*Abs(y - 9) + Max(0, x - 2) with respect to x is: Heaviside(x - 2)
Derivative for f = 6*Abs(y - 9) + Max(0, x - 2) with respect to y is: 6*sign(y - 9)
```

Name: Karan Dua
Student Id: 21331391

Question (a): Implement various Gradient Descent techniques

1. Polyak Gradient Descent

I have used below code to implement Polyak Gradient Descent technique.

```
for _ in range(num_iterations) :

    numerator = function_object.get_quadratic_equation_value(x_point, y_point)
    derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
    denominator = ((derivative_x_value + derivative_y_value) ** 2) + epsilon
    step = numerator / denominator
    x_point = x_point - step * derivative_x_value
    y_point = y_point - step * derivative_y_value
    x_values = np.append(x_values, [x_point], axis = 0)
    y_values = np.append(y_values, [y_point], axis = 0)
    function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])
```

Figure 1: Implementation of Polyak Gradient Descent Step Updation Algorithm

Explanation of code:

Polyak Gradient Descent uses the running average of the gradient descent from previous steps to update the current step. Gradient is calculated using running average which incorporates the momentum using the contribution of past gradients. Step size is calculated using the below formula:

Step Size = f(x, y) / ((df/dx + df/dy)² + ε)

Here, **f(x, y)** is the value of the function for given **x and y** variable, **df/dx** is the derivative of the function w.r.t **x** , **df/dy** is the derivative of the function w.r.t **y,** and **ε** is the small value added in denominator to prevent division by zero **(default set to 1e-8)**.

Furthermore, this function takes **object of the function to be implemented, starting points x and y, and number of iterations** as **input**. In each iteration, I **calculate step size** using the above formula and **update parameters x and y** by subtracting its current value from product of step size and its gradient(derivative).

I have implemented the above function for both the functions provided to me and following are the graphs generated:

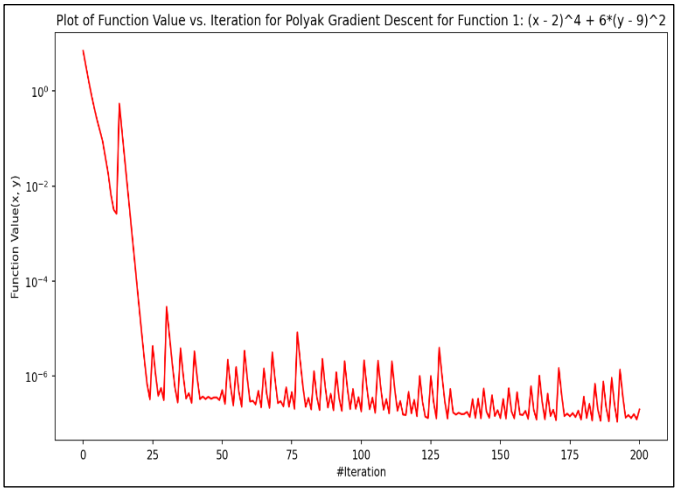

Figure 2a: Plot of Function 1 for PGD convergence

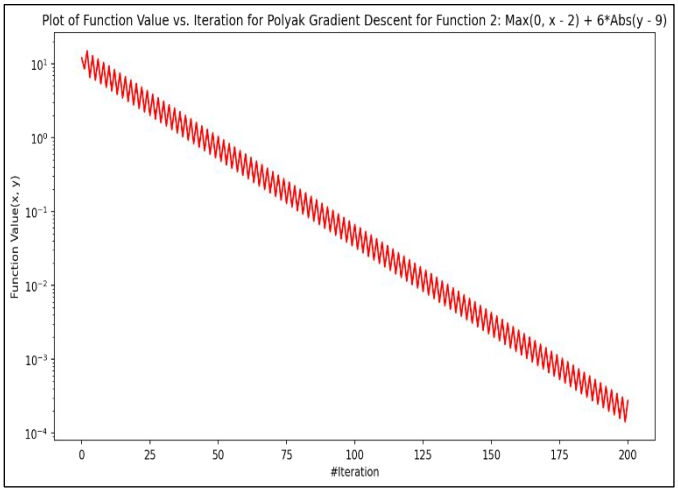

Figure 2b: Plot of Function 2 for PGD convergence

From both the plots we can analyse the Polyak Gradient Descent is able to converge the values to minimum as number of iteration increases. Function1 has converged after just 25 iterations while Function 2 consistently moving towards the minima with each iteration. I can also analyse the impact of momentum factor in these figures as function value is constantly oscillating.

2. RMSProp (Root Mean Square Propagation) Gradient Descent

I have used below code to implement RMSProp Gradient Descent technique.

```
x_alpha = alpha
y_alpha = alpha
x_sum = 0
y_sum = 0

for _ in range(num_iterations) :

    derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
    x_sum = (x_sum * beta) + ((1 - beta) * ((derivative_x_value) ** 2))
    y_sum = (y_sum * beta) + ((1 - beta) * ((derivative_y_value) ** 2))

    x_alpha = alpha / (math.sqrt(x_sum) + epsilon)
    y_alpha = alpha / (math.sqrt(y_sum) + epsilon)

    x_point = x_point - (x_alpha * derivative_x_value)
    y_point = y_point - (y_alpha * derivative_y_value)

    x_values = np.append(x_values, [x_point], axis = 0)
    y_values = np.append(y_values, [y_point], axis = 0)
    function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])
```

Figure 3: Implementation of RMSProp Gradient Descent Step Updation Algorithm

Explanation of code:
RMSProp Gradient Descent is used to update the learning rate of parameters on the basis of the **root mean square value of the gradients** of those parameters. First, we calculate **square of the gradient** and then **take mean** of them. Finally, we **divide the learning rate with square root of the mean square value of the gradients**. Then, we use this learning rate to update the x and y parameters. We also use a **decaying factor called β** which is used to determine the significance of previous gradients.
Step Size:
sum = sum * β + (1 – β) * gradient²
learning_rate = learning_rate / (√sum + ε)
parameter = parameter – learing_rate * gradient
Here, **β** is the decaying rate for gradients, **leaning_rate** is the initial learning rate which is updated in each iteration on the basis of gradients, and **ε** is the small value added in denominator to prevent division by zero **(default set to 1e-8)**.
Furthermore, this function takes **object of the function to be implemented, starting points x and y, learning rate alpha, decaying rate beta, and number of iterations** as **input**. In each iteration, I update **learning rate** using the above formula and **update parameters x and y** by subtracting its current value from product of learning rate and its gradient(derivative).
I have implemented the above function for both the functions provided to me and following are the graphs generated:

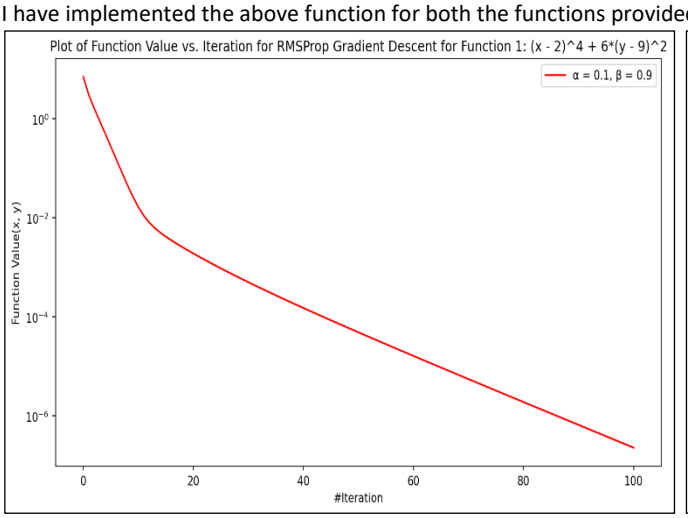

Figure 4a: Plot of Function 1 for RMSProp convergence

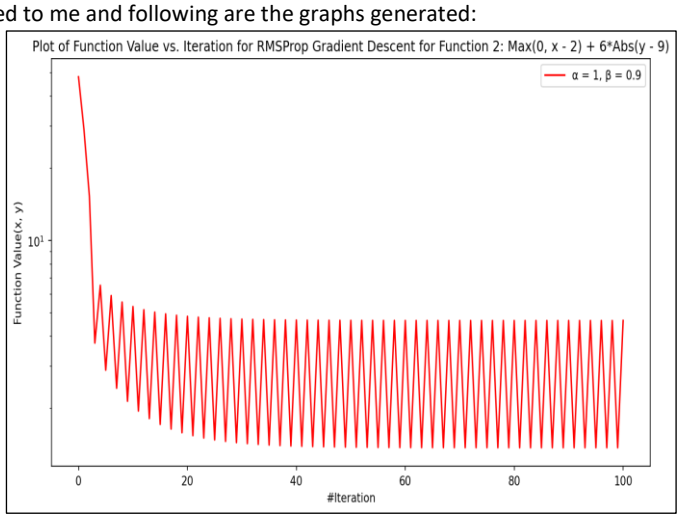

Figure 4b: Plot of Function 2 for RMSProp convergence

From **Figure 4a** I can analyse that Function 1 converged fast for first 15 iterations and then it slowed a little, but it is constantly moving towards the minima. From **Figure 4b** I can analyse that for Function 2, RMSProp was not able to converge the value to minima and it started oscillating after a very few iterations.

3. Heavy Ball Gradient Descent

I have used below code to implement Heavy Ball Gradient Descent technique.

```
    x_z = 0
    y_z = 0

    for _ in range(num_iterations) :

        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
        x_z = (x_z * beta) + (alpha * derivative_x_value)
        y_z = (y_z * beta) + (alpha * derivative_y_value)

        x_point = x_point - x_z
        y_point = y_point - y_z

        x_values = np.append(x_values, [x_point], axis = 0)
        y_values = np.append(y_values, [y_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])
```

Figure 5: Implementation of Heavy Ball Gradient Descent Step Updation Algorithm

Explanation of code:
Heavy Ball gradient descent uses the momentum to update the current parameters in an iteration but tries to reduce the impact of oscillations during convergence. To achieve this, it uses additional weight parameter to control the significance of gradients and momentum. The weight parameter beta controls the weight of previous sum and alpha controls the significance of current derivatives. The key idea is to use the velocity of convergence rather than momentum of convergence for optimisation and reduce the overall oscillations.
Step Size:
velocity = velocity * β + α * gradient
parameter = parameter – velocity
Here, **β** is the weight of previous sum, and **α** is the weight of current gradients.
Furthermore, this function takes **object of the function to be implemented, starting points x and y, weights alpha and beta, and number of iterations** as **input**. In each iteration, I update **velocity** using the above formula and **update parameters x and y** by subtracting its current value from velocity.
I have implemented the above function for both the functions provided to me and following are the graphs generated:

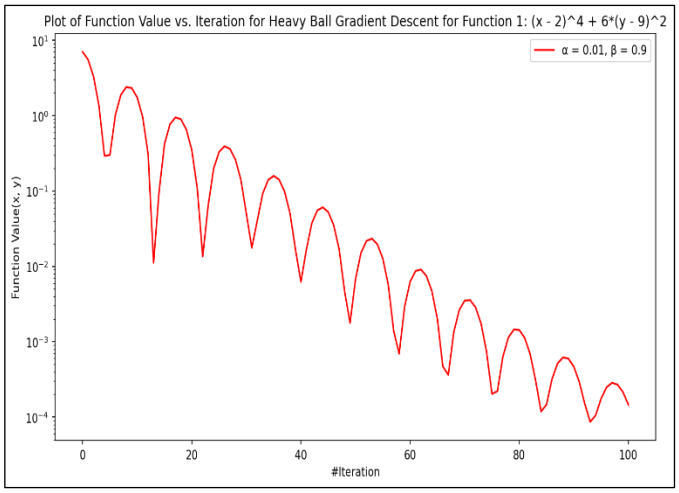

Figure 6a: Plot of Function 1 for Heavy Ball convergence

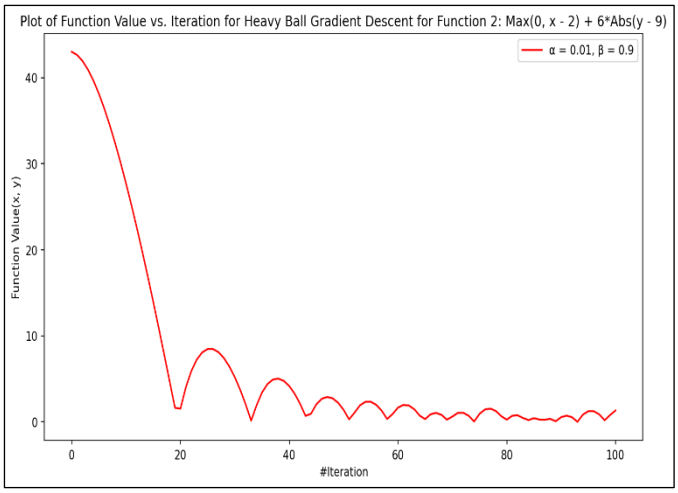

Figure 6b: Plot of Function 2 for Heavy Ball convergence

From **Figure 6a and 6b** I can analyse that both Function 1 and Function 2 have fewer oscillations when compared to Polyak Gradient Descent in Figure 2a and 2b. Both the functions are converging towards minima. Function 1 is constantly converging while Function 2 converges at faster rate first and then it slows with each iteration. However, there is significant decrease in the number of oscillations.

4. Adam Gradient Descent

I have used below code to implement Heavy Ball Gradient Descent technique.

```
    x_v = 0
    x_m = 0

    y_v = 0
    y_m = 0

    for i in range(num_iterations) :

        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)

        x_m = (x_m * beta_1) + ((1 - beta_1) * derivative_x_value)
        y_m = (y_m * beta_1) + ((1 - beta_1) * derivative_y_value)

        x_v = (x_v * beta_2) + ((1 - beta_2) * ((derivative_x_value) ** 2))
        y_v = (y_v * beta_2) + ((1 - beta_2) * ((derivative_y_value) ** 2))

        x_m_hat = x_m / (1 - beta_1 ** i + 1)
        y_m_hat = y_m / (1 - beta_1 ** i + 1)

        x_v_hat = x_v / (1 - beta_2 ** i + 1)
        y_v_hat = y_v / (1 - beta_2 ** i + 1)

        x_alpha = x_m_hat / (math.sqrt(x_v_hat) + epsilon)
        y_alpha = y_m_hat / (math.sqrt(y_v_hat) + epsilon)

        x_point = x_point - (x_alpha * alpha)
        y_point = y_point - (y_alpha * alpha)

        x_values = np.append(x_values, [x_point], axis = 0)
        y_values = np.append(y_values, [y_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])
```

Figure 7: Implementation of Adam Gradient Descent Step Updation Algorithm

Explanation of code:

Adam (Adaptive Moment Estimation) combines the momentum-based algorithms like Heavy Ball with RMSProp algorithms to take advantage of both algorithms. It takes into account **α**, which is learning rate, and 2 weight parameters **β1 and β2.** β1 control the weight of current gradients representing the first moment and β2 controls the weight square of gradients representing the second moment. Now the values calculated for both these moments are scaled in each iteration. Scaling is done by diving these values by $(1 - \beta1)^i$ for the first moment (current gradient) and $(1 - \beta2)^i$ for second moment (squared gradient), where **'i'** is i^{th} iteration. Finally, step size is calculated by dividing the scaled first moment values (current gradient) with square root of second moment (squared gradient) values. **ε** is added in denominator to prevent division by zero **(default set to 1e-8)**.

This function takes **object of the function to be implemented, starting points x and y, learning rate alpha, weights beta1 and beta 2, and number of iterations** as **input**. In each iteration, I calculate **step size** using the above formula and **update parameters x and y** by subtracting its current value from the product of step size with learning rate alpha.

I have implemented the above function for both the functions provided to me and following are the graphs generated:

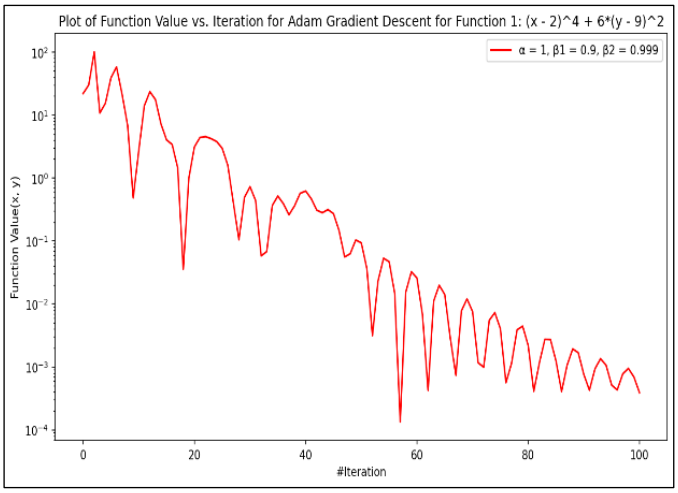

Figure 8a: Plot of Function 1 for Adam convergence

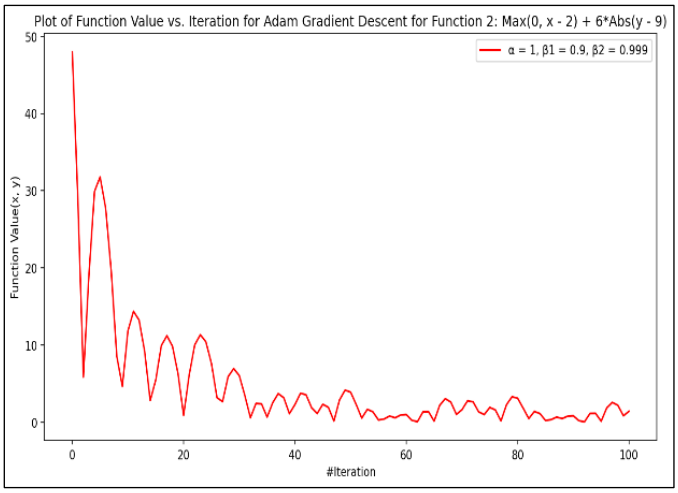

Figure 8b: Plot of Function 2 for Adam convergence

From **Figure 8a and 8b** I can analyse that both Function 1 and Function 2 are converging towards minima. Function 1 is constantly converging while Function 2 converges at faster rate first and then it slows with each iteration. However, I can analyse that oscillations are not of constant magnitude like heavy ball oscillations, but they phase out with iterations which show relationship between momentum and iterations.

Question (b): Optimise various Gradient Descent techniques for hyperparameters α and β for Function 1 and Function 2

1. RMSProp Gradient Descent

I have implemented the above RMSProp Gradient Descent algorithm for a variety of learning rate α in range [0.001, 0.01, 0.1, 1] and decay rate β in range [0.25, 0.9]. The initial conditions for function 1 are x_0 = 1and y_0 = 10 and number of iterations = 200 and for Function 2 x_0 = 3 and y_0 = 1 and number of iterations = 100. For each β parameter, I have varied the α and tried to analyse its impact on overall convergence for both Function1 and Function2. Below are the plots generated.

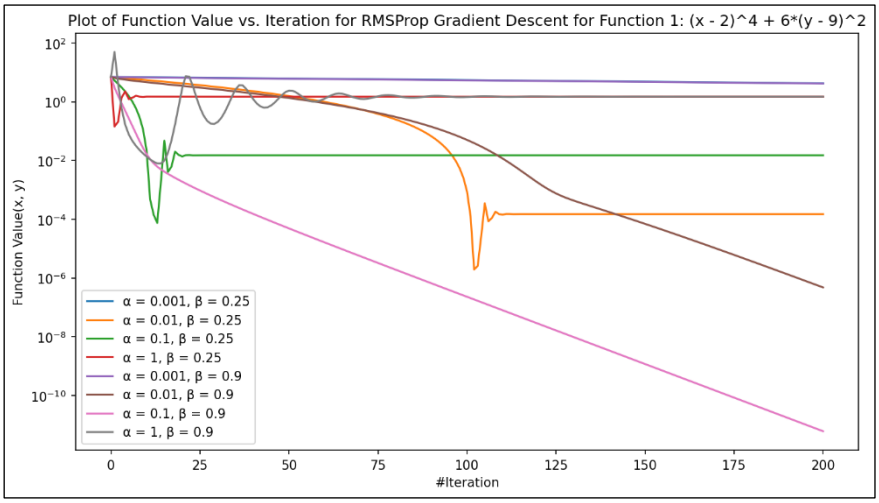

Figure 8a: Plot of Function 1 for RMSProp convergence for range of α and β

From **Figure 8a** I can analyse that for **Function 1**, when value of β is fixed, the learning rate α plays a significant role in overall convergence of the function. The **convergence rate increases significantly as the α increase from 0.001 to 0.1**, but for **α = 1,** we see that after a first few iterations this algorithm jumps over the global minima and starts to **diverge** and become constant. I can analyse the impact of decay factor β. As **β increase from 0.25 to 9**, I can see that **convergence has improved significantly** for higher value of α like 0.1 and was very fast for first few iterations then decreased slowly but it was still accelerating towards minima at a good pace. Thus, we can conclude that for **Function 1 learning rate α should be 0.1 and decay rate β should be 0.9 for optimal convergence**.

From **Figure 8b below** I can analyse that for **Function 2**, when value of β is fixed, the learning rate α plays a significant role in overall convergence of the function. For small values of **α like 0.001 this algorithm shows very poor convergence** and learning cost is significantly high. However, for **α = 0.1**, Function 2 converges smoothly for approximately 70 iterations and the it starts to oscillate over the minima. When α is further **increase to 1**, Function 2 **converges very** fast in first few iterations for both decay rates and then it **starts oscillating**. I can also analyse the impact of decay factor β. **As β increase from 0.25 to 0.9**, I see the **rate of convergence has increased**. Thus, we can conclude that for **Function 2 learning rate α should be 0.1 and decay rate β should be 0.9 for optimal convergence**.

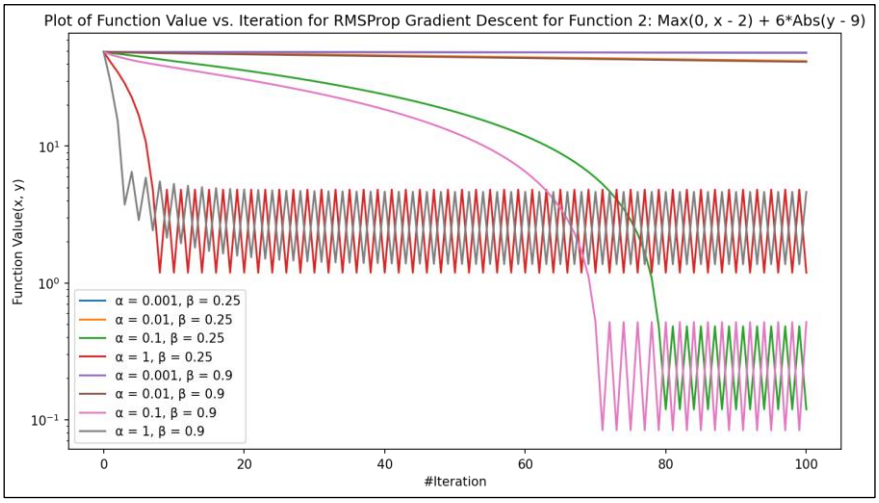

Figure 8b: Plot of Function 2 for RMSProp convergence for range of α and β

From both figures we can analyse that **α and β plays significant role in overall convergence** of these functions. **Decay rate β** plays a more significant role in overall convergence with **high values being more appropriate**. The primary reason for this behaviour is that β decides the magnitude of current gradients in overall momentum and higher β means more significance is given to current gradients. However, the **choice of α** depends on the Function for which this algorithm is optimised but **higher value of α gives better results overall**.

For appropriate values of α and β, **Function 1 has a smoother convergence** and it never oscillates like Function 2. The primary reason for this behaviour is that **Function 1 has smooth convex functions** involving **quadratic** expressions, therefore this algorithm performs well on these convex functions. **Function 2 has kink like surface** because of presence of non-convex functions like **max and abs.** The initial convergence is fast but due to the presence of a kink in this graph, gradient descent is never able to find the minima even though it has descended towards it but it keeps on **oscillating** over it.

2. Heavy Ball Gradient Descent

I have implemented the Heavy Ball Gradient Descent algorithm for a variety of α, which controls the significance of current derivatives, in range [0.001, 0.01, 0.1] and β, which controls the weight of previous momentum, in range [0.25, 0.9]. The initial conditions for function 1 are x_0 = 1and y_0 = 10 and number of iterations = 100 and for Function 2 x_0 = 3 and y_0 = 2 and number of iterations = 100. For each β parameter, I have varied the α and tried to analyse its impact on overall convergence for both Function1 and Function2. Below are the plots generated

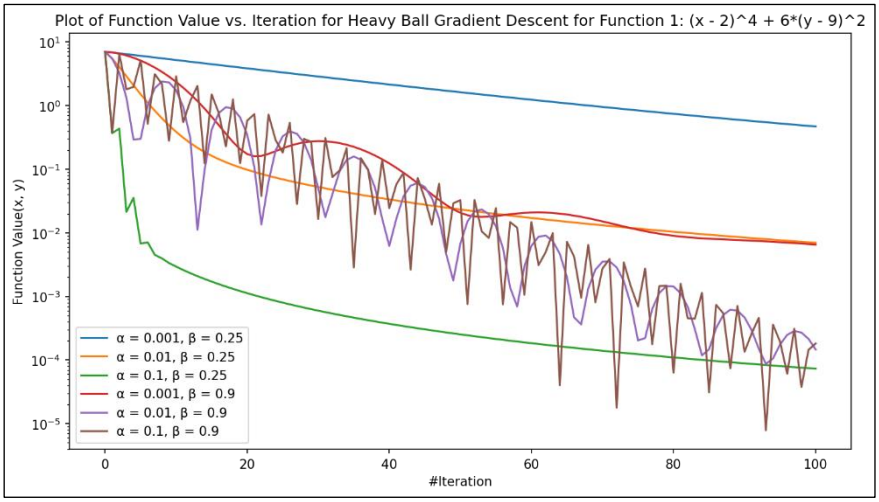

Figure 9a: Plot of Function 1 for Heavy Ball convergence for range of α and β

From **Figure 9a** I can analyse that for **Function 1,** when value of β is fixed, the learning rate α plays a significant role in overall convergence of the function. The **convergence rate increases significantly as the α increase from 0.001 to 0.1.** When α = 0.1, we get best convergence with initial rate of convergence being fast and then it decays slowly. I can also analyse that as **β increase from 0.25 to 9**, I can see that convergence has improved**,** but **increasing β has also increased the rate of oscillations** when α = 0.1in the plot. The primary reason for this behaviour is that **higher β values leads to more weight being given to previous gradients** sum and therefore leading to higher momentum. Thus, we can conclude that for **Function 1 α should be 0.1 and β should be 0.9 for optimal convergence**.

From **Figure 9b below** I can analyse that for **Function 2**, when value of β is fixed, the learning rate α plays a significant role in overall convergence of the function. For small values of **α like 0.001,** for any given β**, this algorithm shows very poor convergence** and learning cost is significantly high. However, for **α = 0.01 or 0.1**, Function 2 converges smoothly for approximately across iterations. **As β increase from 0.25 to 0.9**, I see the **rate of convergence has increased and function converges in first few iterations**. Thus, we can conclude that for **Function 2 learning rate α should be 0.01 and decay rate β should be 0.9 for optimal convergence**.

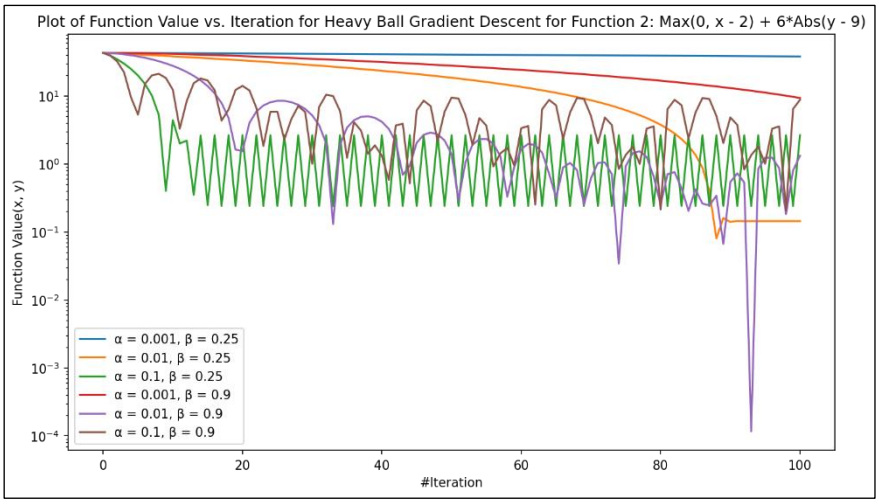

Figure 9b: Plot of Function 2 Heavy Ball convergence for range of α and β

From both figures we can analyse that **α and β plays significant role in overall convergence** of these functions. **β** plays a more significant role in overall convergence with **high values being more appropriate**. The primary reason for this behaviour is that **β** decides the magnitude previous gradients sum in the step size and thus increasing the overall momentum. I can also conclude that **higher value of α** also helps both functions converge faster as more significance is given to current gradients and which results in bigger step size towards minima.

For appropriate values of α and β, **Function 1 has a smoother convergence with good momentum** and it never oscillates like Function 2. The primary reason for this behaviour is that **Function 1 has smooth convex functions** involving **quadratic** expressions, therefore this algorithm performs well on these convex functions. **Function 2 has kink like surface** because of presence of non-convex functions like **max and abs.** The initial convergence is fast due to steep curve but due to the presence of a kink in this graph, gradient descent is never able to find the minima even though it has descended towards it but it keeps on **oscillating** even though there is still momentum in convergence.

3. Adam Gradient Descent

I have implemented the Adam Gradient Descent algorithm for a variety of α, which is the learning rate, in range [0.001, 0.01, 0.1]; β1, which controls the weight of current gradients representing the first moment, in range [0.25, 0.9]; and β2, controls the weight square of gradients representing the second moment, in range [0.9, 0.999]. The initial conditions for function 1 are $x_0 = 4$ and $y_0 = 10$ and number of iterations = 100 and for Function 2 $x_0 = 1$ and $y_0 = 5$ and number of iterations = 100. For each β2 parameter, I have varied the α and β1 and tried to analyse its impact on overall convergence for both Function1 and Function2. Below are the plots generated

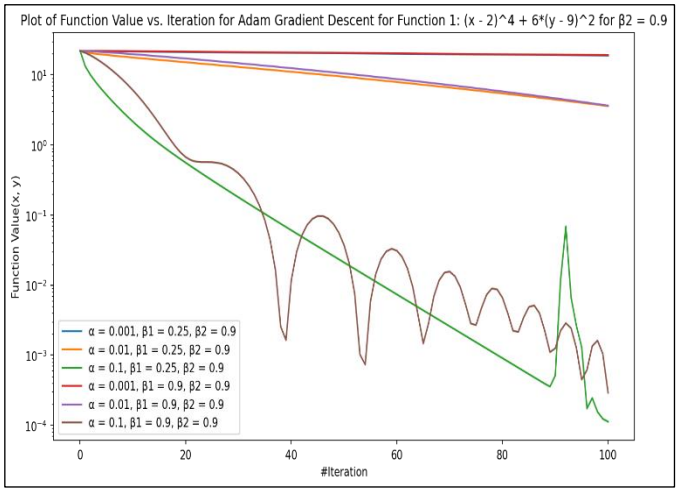

Figure 10a: Plot of Function 1 for Adam convergence for range of α, β1 and β2 = 0.9

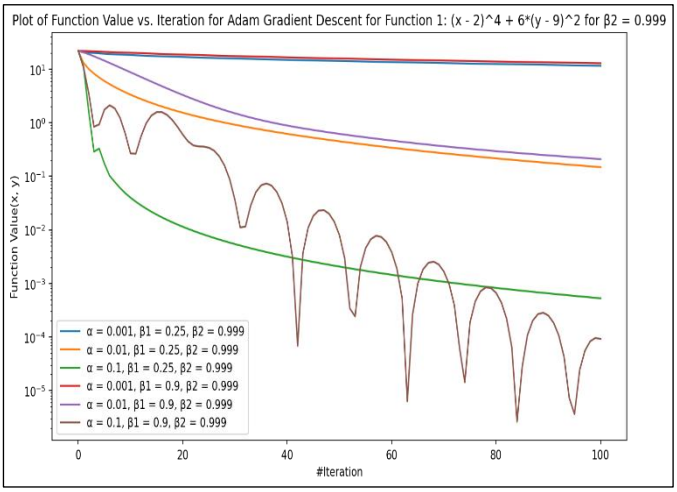

Figure 10b: Plot of Function 1 for Adam convergence for range of α, β1 and β2 = 0.999

From **Figure 10a and 10b** I can analyse that for **Function 1,** when value of β2 is fixed, the learning rate α and β1 plays a significant role in overall convergence of the function. The **convergence rate increases significantly as the α increase from 0.001 to 0.1.** When α = 0.1, we get fastest convergence and rate of convergence is also maintained throughout the iterations. I can also analyse that as **β1 increase from 0.25 to 9**, I can see that convergence has improved**,** but **increasing β1 has also increased the rate of oscillations** in the plot when α = 0.1. I can also see that when **β1** is increased 0.9, the rate of convergence was very fast in the initial iteration and then is get decomposed slightly. As **β2** is increased from **0.9 to 0.999,** the **rate of convergence was slow initially but then it gained momentum** as more and more significance is given to second momentum which results in late bloom with increase in iterations. Due to this **we see better convergence when β2 is 0.999 as algorithm gains momentum when curve is getting flatter near minima**. Thus, we can conclude that for **Function 1 α should be 0.1, β1 should be 0.9 and β2 should be 0.999 for optimal convergence**.

From **Figure 11a and 11b below** I can analyse that for **Function 1,** when value of β2 is fixed, the learning rate α and β1 plays a significant role in overall convergence of the function. The **convergence rate increases significantly as the α increase from 0.001 to 0.1.** When α = 0.1, we get fastest convergence in just 20 iterations and then it starts to oscillate over the minima. I can also analyse that as **β1 increase from 0.25 to 9**, I can see that convergence has slows down a bit**,** but **increasing β1 has also increased the rate of oscillations** in the plot when α = 0.1. As **β2** is increased from **0.9 to 0.999,** the **rate of convergence was has improved significantly with function converging first 10 iterations** when α = 0.1 and then it starts oscillating, but it keeps maintaining momentum during oscillations due to increase in β2. Thus, we can conclude that for **Function 2 α should be 0.1, β1 should be 0.9 and β2 should be 0.999 for optimal convergence**.

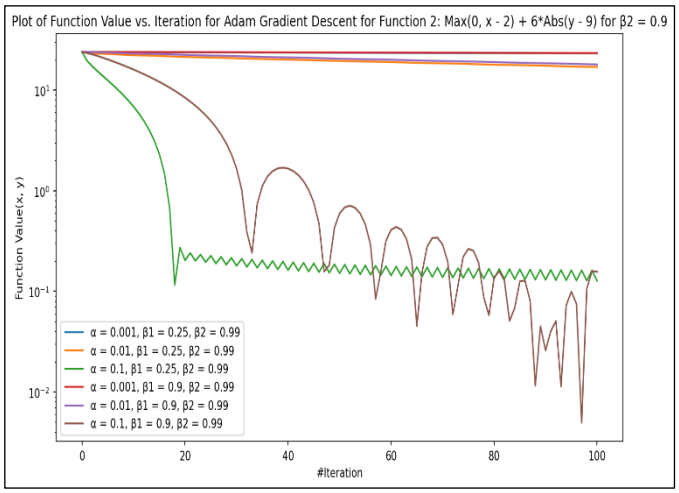

Figure 10a: Plot of Function 2 for Adam convergence for range of α, β1 and β2 = 0.9

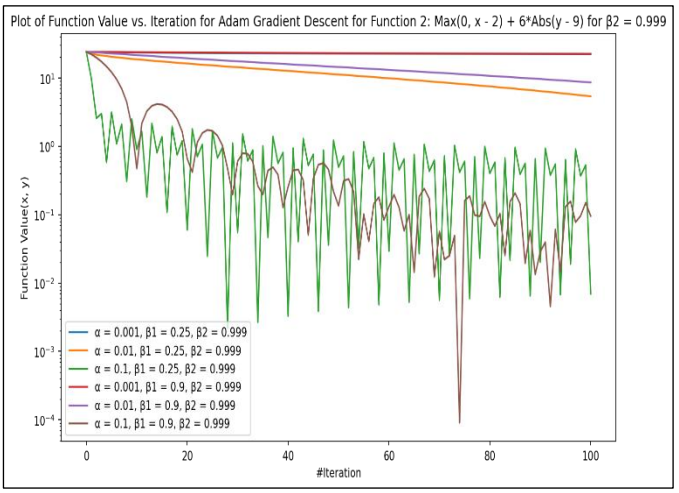

Figure 10b: Plot of Function 2 for Adam convergence for range of α, β1 and β2 = 0.999

From both figures we can analyse that **α, β1 and β2 plays significant role in overall convergence** of these functions. I can also conclude that **higher value of α** helps both functions converge faster as higher learning rate results in bigger step size towards minima. **β1** plays a more significant role in overall convergence with **high values being more appropriate**. The primary reason for this behaviour is that **β1** decides the magnitude current gradients and gives velocity to the graph to move towards minima. **β2** provide momentum to graph to move towards minima when function approaches in flatter regions of the curve by giving more wight to sum pf previous gradients thus higher value should be selected.

For appropriate values of α, β1 and β2, **Function 1 has a smoother convergence with good momentum** and it never oscillates like Function 2 due to presence of **smooth convex functions** involving **quadratic** expressions, therefore this algorithm performs well on these convex functions. **Function 2 has kink like surface** because of presence of non-convex functions like **max and abs.** resulting in fast initial convergence due to steep curve but due to the presence of a kink in this graph, gradient descent is never able to find the minima even though it has descended towards it but it keeps on **oscillating** even though there is still momentum in convergence.

Question (c): Implement Gradient Descent techniques for Relu Function

I have implemented these 3 Gradient Descent technique for Relu Function. As this function accepts only one parameter **x** I have modified these functions to accept only this parameter. I have set α to 0.5, β1 to 0.9 and β2 to 0.999 (only for Adam) based on the hyperparameter tuning performed earlier. Then I varied the starting point x_0 from -1, 1 and 100 and generated below plots.

Condition 1: x_0 = -1

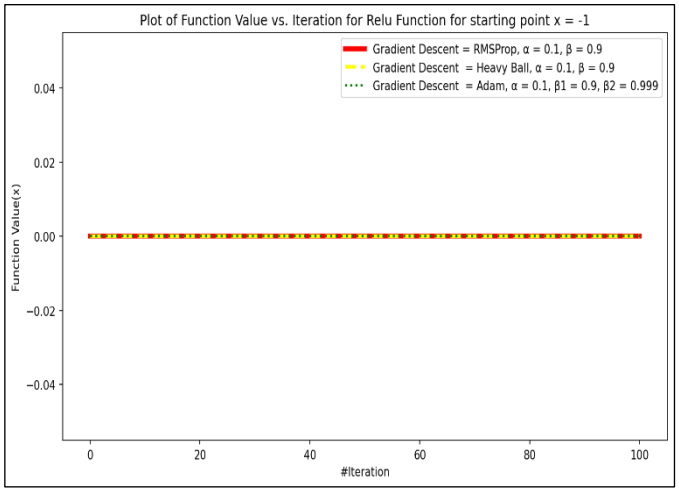

Figure 11a: Plot of Relu Function Value convergence for x_0 = -1

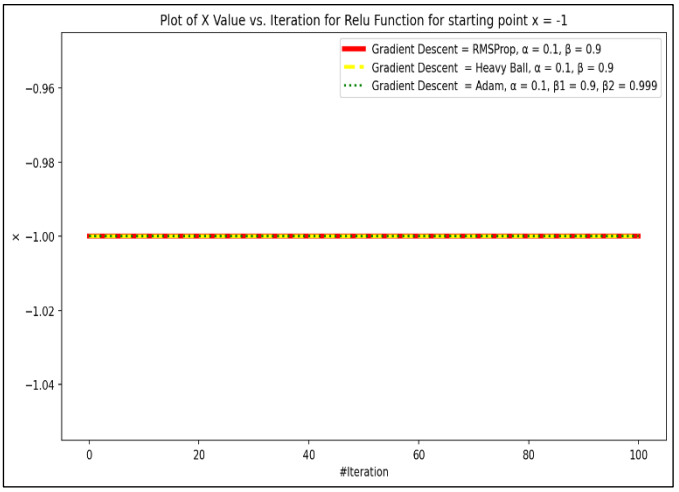

Figure 11b: Plot of X Value convergence for x_0 = -1

From **Figure 11a** I can analyse that all three algorithms do not converge at all but rather remain constant at value 0. The primary reason for this behaviour is that for x = -1, the Relu function i.e., max (0,-1) will always return 0 value which is the minimum value of this function. Furthermore, gradient function at this point Heaviside (-1) will also be 0. Now during the calculation of step size in these algorithms these gradients are multiplied with factors like α, β1 and β2, which results in step size being 0. Therefore, we see no convergence from this point x = -1 for any of these 3 algorithms for any number of iterations.

From **Figure 11b** I can analyse that there is no change in the initial value of x and it remains constant at x = -1 for all the iterations. The primary reason is again that gradient is 0 so x does not deviate from x_0.

Condition 1: x_0 = +1

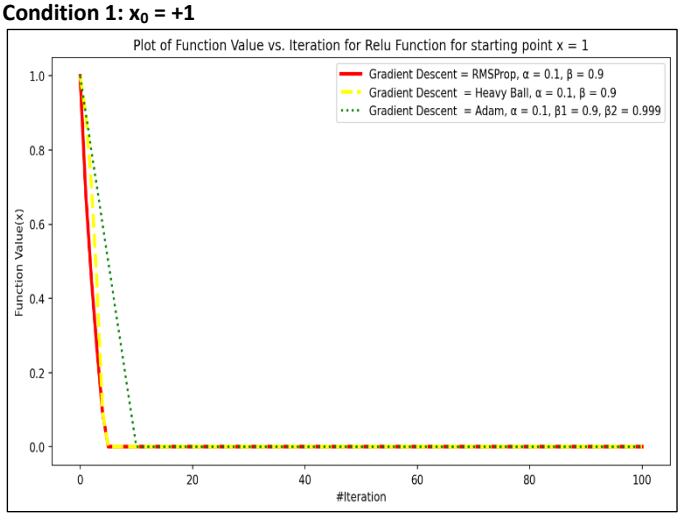

Figure 12a: Plot of Relu Function convergence for x_0 = 1

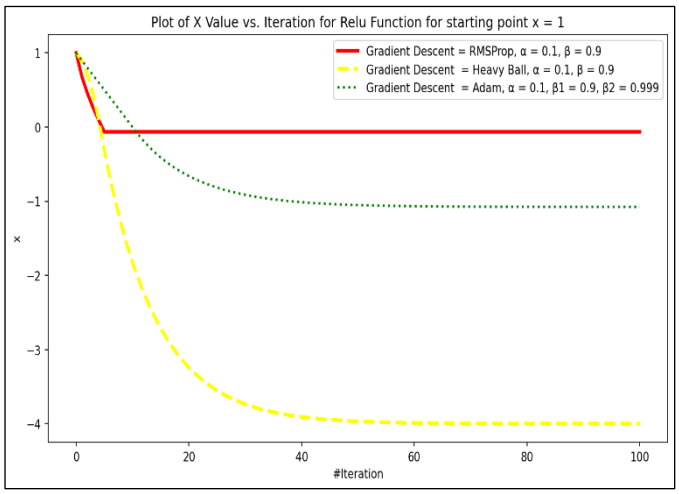

Figure 12b: Plot of X Value convergence for x_0 = 1

From **Figure 12a**, I can analyse that both RMSProp and Heavy Ball algorithm performs slightly better than Adam gradient descent. These two algorithms converge to 0 in about 5 iterations and Adam takes 10 iteration. The primary reason for this behaviour is that Both RMSProp and Heavy Ball algorithms uses concept of velocity and gradient. Now gradient of Relu when x_0 is set to 1 converges to 0 very fast in just first 5 iterations and therefore we see these graph converge very fast. On the other hand, Adam uses adaptive learning and take some more iterations to converge to 0.

From **Figure 12b** I can analyse that x values for all the 3 algorithms goes below 0. The value of Heavy Ball gradient descent converges initially and then it remains constant at -4, while for Adam it settles for about -1. However, I can also analyse that as soon as x value turn negative the value of Relu function stops changing as derivative becomes 0 and therefore step size becomes 0.

Condition 1: x_0 = 100

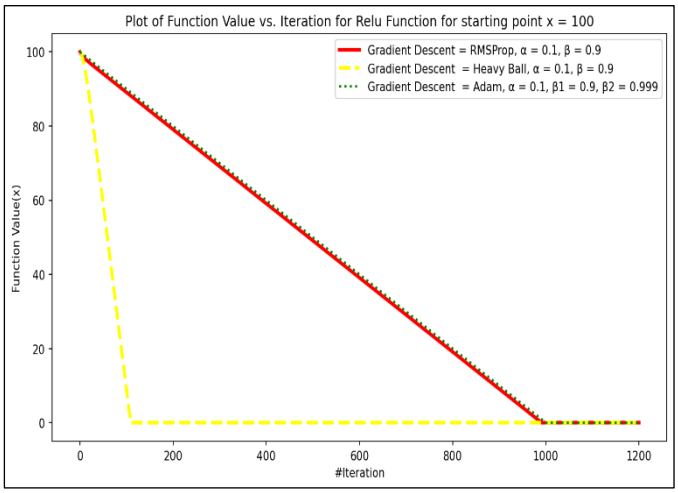

Figure 13a: Plot of Relu Function convergence for x_0 = 100

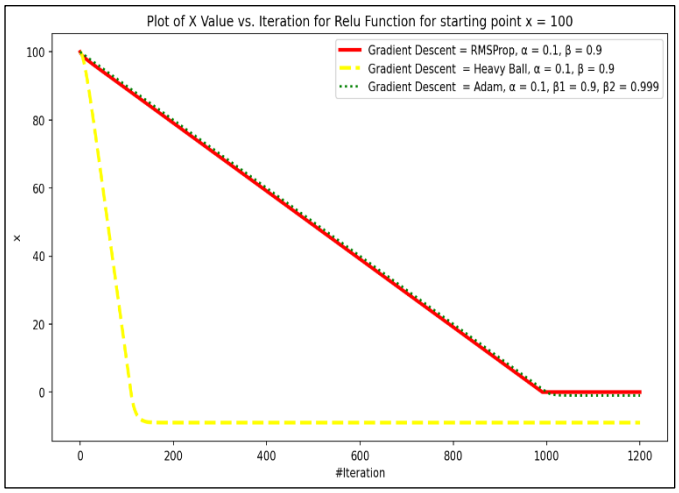

Figure 13b: Plot of X Value convergence for x_0 = 100

From **Figure 13a,** I can clearly analyse that as x increases from 1 to 100, the performance of algorithm also decrease and these take more iterations now to converge as compared earlier. I can also analyse that Heavy ball performs better than Adam and RMSProp and it converges completely in around 170 iterations while Adam and RMSProp take about 1000 iterations. The primary reason for Heavy Ball converging faster than Adam and RMSProp is that β is set to 0.9 and it helps Heavy Ball Algorithm to converge quickly than others due to increase momentum.

From **Figure 13b** I can analyse that x values for all the Heavy Ball algorithm goes below 0. The value of Heavy Ball gradient descent converges initially and then it remains constant at -1. For Adam and RMSProp, it takes from than 1000 iterations to reach negative value and then it remains constant for remaining iterations. I can also analyse that as soon as x value turn negative the value of Relu function stops changing as derivative becomes 0 and therefore step size becomes 0.

Name: Karan Dua
Student Id: 21331391

Appendix: Code for Question 1

```
import numpy as np
import matplotlib.pyplot as plot
import sympy as sm

class Function1 :

    def get_quadratic_equation_symbolic(self) :
        x = sm.symbols('x', real = True)
        y = sm.symbols('y', real = True)
        f = (1 * ((x - 2) ** 4)) + (6 * ((y - 9) ** 2))
        return f

    def get_quadratic_equation_derivative_wrt_x_symbolic(self) :
        x = sm.symbols('x', real = True)
        f = self.get_quadratic_equation_symbolic()
        derivative_x = sm.diff(f, x)
        return derivative_x

    def get_quadratic_equation_derivative_wrt_y_symbolic(self) :
        y = sm.symbols('y', real = True)
        f = self.get_quadratic_equation_symbolic()
        derivative_y = sm.diff(f, y)
        return derivative_y

function1 = Function1()
f = function1.get_quadratic_equation_symbolic()
print(f"Quadratic equation for Function 1 is: {f}")

derivative_df_dx = function1.get_quadratic_equation_derivative_wrt_x_symbolic()
print(f"Derivative for f = {f} with respect to x is: {derivative_df_dx}")

derivative_df_dy = function1.get_quadratic_equation_derivative_wrt_y_symbolic()
print(f"Derivative for f = {f} with respect to y is: {derivative_df_dy}")

class Function2 :

    def get_quadratic_equation_symbolic(self) :
        x = sm.symbols('x', real = True)
        y = sm.symbols('y', real = True)
        f = (sm.Max(x - 2, 0)) + (6 * (sm.Abs(y - 9)))
        return f

    def get_quadratic_equation_derivative_wrt_x_symbolic(self) :
        x = sm.symbols('x', real = True)
        f = self.get_quadratic_equation_symbolic()
        derivative_x = sm.diff(f, x)
        return derivative_x

    def get_quadratic_equation_derivative_wrt_y_symbolic(self) :
        y = sm.symbols('y', real = True)
        f = self.get_quadratic_equation_symbolic()
        derivative_y = sm.diff(f, y)
        return derivative_y

function2 = Function2()
f = function2.get_quadratic_equation_symbolic()
print(f"Quadratic equation for Function 2 is: {f}")

derivative_df_dx = function2.get_quadratic_equation_derivative_wrt_x_symbolic()
print(f"Derivative for f = {f} with respect to x is: {derivative_df_dx}")

derivative_df_dy = function2.get_quadratic_equation_derivative_wrt_y_symbolic()
print(f"Derivative for f = {f} with respect to y is: {derivative_df_dy}")
```

Appendix: Code for Question a – All Parts

```
import numpy as np
import matplotlib.pyplot as plot
import sympy as sm
import math

class Function1 :

    def get_quadratic_equation_value(self, x, y) :
        value = ((x - 2) ** 4) + (6 * ((y - 9) ** 2))
        return value

    def get_quadratic_equation_derivative_value(self, x, y) :
        derivative_x_value = 4 * ((x - 2) ** 3)
        derivative_y_value = (12 * y) - 108
        return derivative_x_value, derivative_y_value

function1 = Function1()
test_function_value = function1.get_quadratic_equation_value(3, 3)
print(f'Function 1 value for x = 3 and y = 3 is: {test_function_value}')

test_function_derivative_value_x, test_function_derivative_value_y = function1.get_quadratic_equation_derivative_value(3, 3)
print(f'Function 1 derivative for x = 3 is: {test_function_derivative_value_x} and for y = 3 is: {test_function_derivative_value_y}')

class Function2 :

    def get_quadratic_equation_value(self, x, y) :
        value = (max((x - 2), 0)) + (6 * np.abs(y - 9))
        return value

    def get_quadratic_equation_derivative_value(self, x, y) :
        derivative_x_value = np.heaviside((x - 2), 1)
        derivative_y_value = 6 * np.sign(y - 9)
        return derivative_x_value, derivative_y_value

function2 = Function2()
test_function_value = function2.get_quadratic_equation_value(-1, 3)
print(f'Function 2 value for x = 3 and y = 3 is: {test_function_value}')

test_function_derivative_value_x, test_function_derivative_value_y = function2.get_quadratic_equation_derivative_value(3, 3)
print(f'Function 2 derivative for x = 3 is: {test_function_derivative_value_x} and for y = 3 is: {test_function_derivative_value_y}')
```

Question a(i): Polyak step size Gradient Descent

```
def execute_polyak_gradient_descent(function_object, starting_point_x = 1, starting_point_y = 1, num_iterations = 50) :

    x_point = starting_point_x
    y_point = starting_point_y
    x_values = np.array([starting_point_x])
    y_values = np.array([starting_point_y])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x, starting_point_y)])
    epsilon = 1e-8

    for _ in range(num_iterations) :

        numerator = function_object.get_quadratic_equation_value(x_point, y_point)
        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
        denominator = ((derivative_x_value + derivative_y_value) ** 2) + epsilon
        step = numerator / denominator
        x_point = x_point - step * derivative_x_value
        y_point = y_point - step * derivative_y_value
        x_values = np.append(x_values, [x_point], axis = 0)
        y_values = np.append(y_values, [y_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])
```

```
    return x_values, y_values, function_values

function1 = Function1()
x_values, y_values, function_values = execute_polyak_gradient_descent(function1, starting_point_x = 1, starting_point_y = 8,
num_iterations = 200)

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(function_values, color = "Red")
plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Polyak Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.show()

function2 = Function2()
x_values, y_values, function_values = execute_polyak_gradient_descent(function2, starting_point_x = 8, starting_point_y = 10,
num_iterations = 200)

plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(function_values, color = "Red")
plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Polyak Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')
plot.yscale("log")
plot.show()
```

Question a(ii): RMSProp Gradient Descent

```
def execute_rmsprop_gradient_descent(function_object, starting_point_x = 1, starting_point_y = 1, alpha = 1, beta = 1, num_iterations =
50) :

    x_point = starting_point_x
    y_point = starting_point_y
    x_values = np.array([starting_point_x])
    y_values = np.array([starting_point_y])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x, starting_point_y)])
    epsilon = 1e-8

    x_alpha = alpha
    y_alpha = alpha
    x_sum = 0
    y_sum = 0

    for _ in range(num_iterations) :

        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
        x_sum = (x_sum * beta) + ((1 - beta) * ((derivative_x_value) ** 2))
        y_sum = (y_sum * beta) + ((1 - beta) * ((derivative_y_value) ** 2))

        x_alpha = alpha / (math.sqrt(x_sum) + epsilon)
        y_alpha = alpha / (math.sqrt(y_sum) + epsilon)

        x_point = x_point - (x_alpha * derivative_x_value)
        y_point = y_point - (y_alpha * derivative_y_value)

        x_values = np.append(x_values, [x_point], axis = 0)
        y_values = np.append(y_values, [y_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])

    return x_values, y_values, function_values

function1 = Function1()
x_values, y_values, function_values = execute_rmsprop_gradient_descent(function1, starting_point_x = 1, starting_point_y = 10, alpha =
0.1, beta = 0.9, num_iterations = 100)
plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(function_values, color = "Red", label = 'α = 0.1, β = 0.9')
```

```
plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for RMSProp Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()

function2 = Function2()
x_values, y_values, function_values = execute_rmsprop_gradient_descent(function2, starting_point_x = 1, starting_point_y = 1, alpha = 1,
beta = 0.9, num_iterations = 100)
plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(function_values, color = "Red", label = 'α = 1, β = 0.9')
plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for RMSProp Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')
plot.yscale("log")
plot.legend()
plot.show()
```

Question a(iii): Heavy Ball Gradient Descent

```
def execute_heavyball_gradient_descent(function_object, starting_point_x = 1, starting_point_y = 1, alpha = 1, beta = 1, num_iterations =
50) :

    x_point = starting_point_x
    y_point = starting_point_y
    x_values = np.array([starting_point_x])
    y_values = np.array([starting_point_y])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x, starting_point_y)])
    epsilon = 1e-8

    x_z = 0
    y_z = 0

    for _ in range(num_iterations) :

        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
        x_z = (x_z * beta) + (alpha * derivative_x_value)
        y_z = (y_z * beta) + (alpha * derivative_y_value)

        x_point = x_point - x_z
        y_point = y_point - y_z

        x_values = np.append(x_values, [x_point], axis = 0)
        y_values = np.append(y_values, [y_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])

    return x_values, y_values, function_values

function1 = Function1()
x_values, y_values, function_values = execute_heavyball_gradient_descent(function1, starting_point_x = 1, starting_point_y = 10, alpha =
0.01, beta = 0.9, num_iterations = 100)
plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(function_values, color = "Red", label = 'α = 0.01, β = 0.9')
plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Heavy Ball Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()

function2 = Function2()
x_values, y_values, function_values = execute_heavyball_gradient_descent(function2, starting_point_x = 3, starting_point_y = 2, alpha =
0.01, beta = 0.9, num_iterations = 100)
plot.figure(figsize=(11, 6), dpi=150)
plot.plot(function_values, color = "Red", label = 'α = 0.01, β = 0.9')
```

```
plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Heavy Ball Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')
plot.legend()
plot.show()
```

Question a(iv): Adam Gradient Descent

```
def execute_adam_gradient_descent(function_object, starting_point_x = 1, starting_point_y = 1, alpha = 1, beta_1 = 1, beta_2 = 1,
num_iterations = 50) :

    x_point = starting_point_x
    y_point = starting_point_y
    x_values = np.array([starting_point_x])
    y_values = np.array([starting_point_y])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x, starting_point_y)])
    epsilon = 1e-8

    x_v = 0
    x_m = 0

    y_v = 0
    y_m = 0

    for i in range(num_iterations) :

        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)

        x_m = (x_m * beta_1) + ((1 - beta_1) * derivative_x_value)
        y_m = (y_m * beta_1) + ((1 - beta_1) * derivative_y_value)

        x_v = (x_v * beta_2) + ((1 - beta_2) * ((derivative_x_value) ** 2))
        y_v = (y_v * beta_2) + ((1 - beta_2) * ((derivative_y_value) ** 2))

        x_m_hat = x_m / (1 - beta_1 ** i + 1)
        y_m_hat = y_m / (1 - beta_1 ** i + 1)

        x_v_hat = x_v / (1 - beta_2 ** i + 1)
        y_v_hat = y_v / (1 - beta_2 ** i + 1)

        x_alpha = x_m_hat / (math.sqrt(x_v_hat) + epsilon)
        y_alpha = y_m_hat / (math.sqrt(y_v_hat) + epsilon)

        x_point = x_point - (x_alpha * alpha)
        y_point = y_point - (y_alpha * alpha)

        x_values = np.append(x_values, [x_point], axis = 0)
        y_values = np.append(y_values, [y_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])

    return x_values, y_values, function_values

function1 = Function1()
x_values, y_values, function_values = execute_adam_gradient_descent(function1, starting_point_x = 4, starting_point_y = 10, alpha = 1,
beta_1 = 0.9, beta_2 = 0.999, num_iterations = 100)
plot.figure(figsize=(11, 6), dpi=150)
plot.semilogy(function_values, color = "Red", label = 'α = 1, β1 = 0.9, β2 = 0.999')
plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Adam Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()

function2 = Function2()
```

```
x_values, y_values, function_values = execute_adam_gradient_descent(function2, starting_point_x = 1, starting_point_y = 1, alpha = 1,
beta_1 = 0.9, beta_2 = 0.999, num_iterations = 100)
plot.figure(figsize=(11, 6), dpi=150)
plot.plot(function_values, color = "Red", label = 'α = 1, β1 = 0.9, β2 = 0.999' )
plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Adam Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')

plot.legend()
plot.show()
```

Appendix: Code for Question b – All Parts

```
import numpy as np
import matplotlib.pyplot as plot
import sympy as sm
import math

class Function1 :

    def get_quadratic_equation_value(self, x, y) :
        value = ((x - 2) ** 4) + (6 * ((y - 9) ** 2))
        return value

    def get_quadratic_equation_derivative_value(self, x, y) :
        derivative_x_value = 4 * ((x - 2) ** 3)
        derivative_y_value = (12 * y) - 108
        return derivative_x_value, derivative_y_value

function1 = Function1()
test_function_value = function1.get_quadratic_equation_value(3, 3)
print(f'Function 1 value for x = 3 and y = 3 is: {test_function_value}')

test_function_derivative_value_x, test_function_derivative_value_y = function1.get_quadratic_equation_derivative_value(3, 3)
print(f'Function 1 derivative for x = 3 is: {test_function_derivative_value_x} and for y = 3 is: {test_function_derivative_value_y}')

class Function2 :

    def get_quadratic_equation_value(self, x, y) :
        value = (max((x - 2), 0)) + (6 * np.abs(y - 9))
        return value

    def get_quadratic_equation_derivative_value(self, x, y) :
        derivative_x_value = np.heaviside((x - 2), 1)
        derivative_y_value = 6 * np.sign(y - 9)
        return derivative_x_value, derivative_y_value

function2 = Function2()
test_function_value = function2.get_quadratic_equation_value(3, 3)
print(f'Function 2 value for x = 3 and y = 3 is: {test_function_value}')

test_function_derivative_value_x, test_function_derivative_value_y = function2.get_quadratic_equation_derivative_value(3, 3)
print(f'Function 2 derivative for x = 3 is: {test_function_derivative_value_x} and for y = 3 is: {test_function_derivative_value_y}')
```

Question b(i): RMSProp Gradient Descent for a variety of α and β

```
def execute_rmsprop_gradient_descent(function_object, starting_point_x = 1, starting_point_y = 1, alpha = 1, beta = 1, num_iterations =
50) :

    x_point = starting_point_x
    y_point = starting_point_y
    x_values = np.array([starting_point_x])
    y_values = np.array([starting_point_y])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x, starting_point_y)])
    epsilon = 1e-8

    x_alpha = alpha
    y_alpha = alpha
    x_sum = 0
    y_sum = 0

    for _ in range(num_iterations) :

        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
        x_sum = (x_sum * beta) + ((1 - beta) * ((derivative_x_value) ** 2))
        y_sum = (y_sum * beta) + ((1 - beta) * ((derivative_y_value) ** 2))
```

```
        x_alpha = alpha / (math.sqrt(x_sum) + epsilon)
        y_alpha = alpha / (math.sqrt(y_sum) + epsilon)

        x_point = x_point - (x_alpha * derivative_x_value)
        y_point = y_point - (y_alpha * derivative_y_value)

        x_values = np.append(x_values, [x_point], axis = 0)
        y_values = np.append(y_values, [y_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])

    return x_values, y_values, function_values

function1 = Function1()

alpha_range = [0.001, 0.01, 0.1, 1]
beta_range = [0.25, 0.9]
plot.figure(figsize=(11, 6), dpi=150)
for beta in beta_range :

    for alpha in alpha_range :

        try :
            x_values, y_values, function_values = execute_rmsprop_gradient_descent(function1, starting_point_x = 1, starting_point_y = 10,
alpha = alpha, beta = beta, num_iterations = 200)
            plot.semilogy(function_values, label = f'α = {alpha}, β = {beta}')
        except:
            print(f'Exception Occured for α = {alpha}, β = {beta}')

plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for RMSProp Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()

function2 = Function2()

alpha_range = [0.001, 0.01, 0.1, 1]
beta_range = [0.25, 0.9]
plot.figure(figsize=(11, 6), dpi=150)
for beta in beta_range :

    for alpha in alpha_range :

        try :
            x_values, y_values, function_values = execute_rmsprop_gradient_descent(function2, starting_point_x = 3, starting_point_y = 1,
alpha = alpha, beta = beta, num_iterations = 100)
            plot.plot(function_values, label = f'α = {alpha}, β = {beta}')
        except:
            print(f'Exception Occured for α = {alpha}, β = {beta}')

plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for RMSProp Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')
plot.legend()
plot.yscale("log")
plot.show()
```

Question b(ii): Heavy Ball Gradient Descent for a variety of α and β

```
def execute_heavyball_gradient_descent(function_object, starting_point_x = 1, starting_point_y = 1, alpha = 1, beta = 1, num_iterations =
50) :

    x_point = starting_point_x
    y_point = starting_point_y
    x_values = np.array([starting_point_x])
```

```
    y_values = np.array([starting_point_y])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x, starting_point_y)])
    epsilon = 1e-8

    x_z = 0
    y_z = 0

    for _ in range(num_iterations) :

        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)
        x_z = (x_z * beta) + (alpha * derivative_x_value)
        y_z = (y_z * beta) + (alpha * derivative_y_value)

        x_point = x_point - x_z
        y_point = y_point - y_z

        x_values = np.append(x_values, [x_point], axis = 0)
        y_values = np.append(y_values, [y_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])

    return x_values, y_values, function_values

function1 = Function1()

alpha_range = [0.001, 0.01, 0.1]
beta_range = [0.25, 0.9]
plot.figure(figsize=(11, 6), dpi=150)
for beta in beta_range :

    for alpha in alpha_range :

        try :
            x_values, y_values, function_values = execute_heavyball_gradient_descent(function1, starting_point_x = 1, starting_point_y = 10,
alpha = alpha, beta = beta, num_iterations = 100)
            plot.semilogy(function_values, label = f'α = {alpha}, β = {beta}')
        except:
            print(f'Exception Occured for α = {alpha}, β = {beta}')
plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Heavy Ball Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2')
plot.yscale("log")
plot.legend()
plot.show()

function2 = Function2()

alpha_range = [0.001, 0.01, 0.1]
beta_range = [0.25, 0.9]
plot.figure(figsize=(11, 6), dpi=150)
for beta in beta_range :

    for alpha in alpha_range :

        try :
            x_values, y_values, function_values = execute_heavyball_gradient_descent(function2, starting_point_x = 3, starting_point_y = 2,
alpha = alpha, beta = beta, num_iterations = 100)
            plot.plot(function_values, label = f'α = {alpha}, β = {beta}')
        except:
            print(f'Exception Occured for α = {alpha}, β = {beta}')

plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Heavy Ball Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9)')
plot.legend()
plot.yscale("log")
plot.show()
```

Question b(iii): Adam Gradient Descent for a variety of α, β1 and β2

```
def execute_adam_gradient_descent(function_object, starting_point_x = 1, starting_point_y = 1, alpha = 1, beta_1 = 1, beta_2 = 1,
num_iterations = 50) :

    x_point = starting_point_x
    y_point = starting_point_y
    x_values = np.array([starting_point_x])
    y_values = np.array([starting_point_y])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x, starting_point_y)])
    epsilon = 1e-8

    x_v = 0
    x_m = 0

    y_v = 0
    y_m = 0

    for i in range(num_iterations) :

        derivative_x_value, derivative_y_value = function_object.get_quadratic_equation_derivative_value(x_point, y_point)

        x_m = (x_m * beta_1) + ((1 - beta_1) * derivative_x_value)
        y_m = (y_m * beta_1) + ((1 - beta_1) * derivative_y_value)

        x_v = (x_v * beta_2) + ((1 - beta_2) * ((derivative_x_value) ** 2))
        y_v = (y_v * beta_2) + ((1 - beta_2) * ((derivative_y_value) ** 2))

        x_m_hat = x_m / (1 - beta_1 ** i + 1)
        y_m_hat = y_m / (1 - beta_1 ** i + 1)

        x_v_hat = x_v / (1 - beta_2 ** i + 1)
        y_v_hat = y_v / (1 - beta_2 ** i + 1)

        x_alpha = x_m_hat / (math.sqrt(x_v_hat) + epsilon)
        y_alpha = y_m_hat / (math.sqrt(y_v_hat) + epsilon)

        x_point = x_point - (x_alpha * alpha)
        y_point = y_point - (y_alpha * alpha)

        x_values = np.append(x_values, [x_point], axis = 0)
        y_values = np.append(y_values, [y_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point, y_point)])

    return x_values, y_values, function_values

function1 = Function1()

alpha_range = [0.001, 0.01, 0.1]
beta_1_range = [0.25, 0.9]
beta_2 = 0.9
plot.figure(figsize=(11, 6), dpi=150)
for beta_1 in beta_1_range :

    for alpha in alpha_range :

        try :
            x_values, y_values, function_values = execute_adam_gradient_descent(function1, starting_point_x = 4, starting_point_y = 10, alpha
= alpha, beta_1 = beta_1, beta_2 = beta_2, num_iterations = 100)
            plot.semilogy(function_values, label = f'α = {alpha}, β1 = {beta_1}, β2 = {beta_2}')
        except:
            print(f'Exception Occured for α = {alpha}, β1 = {beta_1}, β2 = {beta_2}')

plot.xlabel('#Iteration')
```

```
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Adam Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2 for β2 = 0.9')
plot.yscale("log")
plot.legend()
plot.show()

function1 = Function1()

alpha_range = [0.001, 0.01, 0.1]
beta_1_range = [0.25, 0.9]
beta_2 = 0.999
plot.figure(figsize=(11, 6), dpi=150)

for beta_1 in beta_1_range :

   for alpha in alpha_range :

      try :
         x_values, y_values, function_values = execute_adam_gradient_descent(function1, starting_point_x = 4, starting_point_y = 10, alpha
= alpha, beta_1 = beta_1, beta_2 = beta_2, num_iterations = 100)
         plot.semilogy(function_values, label = f'α = {alpha}, β1 = {beta_1}, β2 = {beta_2}')
      except:
         print(f'Exception Occured for α = {alpha}, β1 = {beta_1}, β2 = {beta_2}')

plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Adam Gradient Descent for Function 1: (x - 2)^4 + 6*(y - 9)^2 for β2 = 0.999')
plot.yscale("log")
plot.legend()
plot.show()

function2 = Function2()

alpha_range = [0.001, 0.01, 0.1]
beta_1_range = [0.25, 0.9]
beta_2 = 0.99

plot.figure(figsize=(11, 6), dpi=150)
for beta_1 in beta_1_range :

   for alpha in alpha_range :

      try :
         x_values, y_values, function_values = execute_adam_gradient_descent(function2, starting_point_x = 1, starting_point_y = 5, alpha =
alpha, beta_1 = beta_1, beta_2 = beta_2, num_iterations = 100)
         plot.plot(function_values, label = f'α = {alpha}, β1 = {beta_1}, β2 = {beta_2}')
      except:
         print(f'Exception Occured for α = {alpha}, β1 = {beta_1}, β2 = {beta_2}')

plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Adam Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9) for β2 = 0.9')
plot.legend()
plot.yscale("log")
plot.show()

function2 = Function2()

alpha_range = [0.001, 0.01, 0.1]
beta_1_range = [0.25, 0.9]
beta_2 = 0.999

plot.figure(figsize=(11, 6), dpi=150)
for beta_1 in beta_1_range :

   for alpha in alpha_range :
```

```
    try :
        x_values, y_values, function_values = execute_adam_gradient_descent(function2, starting_point_x = 1, starting_point_y = 5, alpha =
alpha, beta_1 = beta_1, beta_2 = beta_2, num_iterations = 100)
        plot.plot(function_values, label = f'α = {alpha}, β1 = {beta_1}, β2 = {beta_2}')
    except:
        print(f'Exception Occured for α = {alpha}, β1 = {beta_1}, β2 = {beta_2}')

plot.xlabel('#Iteration')
plot.ylabel('Function Value(x, y)')
plot.title('Plot of Function Value vs. Iteration for Adam Gradient Descent for Function 2: Max(0, x - 2) + 6*Abs(y - 9) for β2 = 0.999')
plot.legend()
plot.yscale("log")
plot.show()
```

Appendix: Code for Question c – All Parts

```
import numpy as np
import matplotlib.pyplot as plot
import sympy as sm
import math

class Function_Relu :

    def get_quadratic_equation_value(self, x) :
        value = max(x,0)
        return value

    def get_quadratic_equation_derivative_value(self, x) :
        derivative_x_value = np.heaviside(x, 0)
        return derivative_x_value

relu_test = Function_Relu()
test_function_value = relu_test.get_quadratic_equation_value(-1)
print(f'Relu Function value for x = -1 is: {test_function_value}')

test_function_derivative_value_x = relu_test.get_quadratic_equation_derivative_value(-1)
print(f'Relu Function derivative for x = -1 is: {test_function_derivative_value_x}')

relu_test = Function_Relu()
test_function_value = relu_test.get_quadratic_equation_value(3)
print(f'Relu Function value for x = 3 is: {test_function_value}')

test_function_derivative_value_x = relu_test.get_quadratic_equation_derivative_value(3)
print(f'Relu Function derivative for x = 3 is: {test_function_derivative_value_x}')

def execute_rmsprop_gradient_descent(function_object, starting_point_x = 1, alpha = 1, beta = 1, num_iterations = 50) :

    x_point = starting_point_x
    x_values = np.array([starting_point_x])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x)])
    epsilon = 1e-8

    x_alpha = alpha
    x_sum = 0

    for _ in range(num_iterations) :

        derivative_x_value = function_object.get_quadratic_equation_derivative_value(x_point)
        x_sum = (x_sum * beta) + ((1 - beta) * ((derivative_x_value) ** 2))

        x_alpha = alpha / (math.sqrt(x_sum) + epsilon)

        x_point = x_point - (x_alpha * derivative_x_value)

        x_values = np.append(x_values, [x_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point)])

    return x_values, function_values

def execute_heavyball_gradient_descent(function_object, starting_point_x = 1, alpha = 1, beta = 1, num_iterations = 50) :

    x_point = starting_point_x
    x_values = np.array([starting_point_x])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x)])
    epsilon = 1e-8

    x_z = 0

    for _ in range(num_iterations) :
```

```
        derivative_x_value = function_object.get_quadratic_equation_derivative_value(x_point)
        x_z = (x_z * beta) + (alpha * derivative_x_value)

        x_point = x_point - x_z

        x_values = np.append(x_values, [x_point], axis = 0)
        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point)])

    return x_values, function_values

def execute_adam_gradient_descent(function_object, starting_point_x = 1, alpha = 1, beta_1 = 1, beta_2 = 1, num_iterations = 50) :

    x_point = starting_point_x
    x_values = np.array([starting_point_x])
    function_values = np.array([function_object.get_quadratic_equation_value(starting_point_x)])
    epsilon = 1e-8

    x_v = 0
    x_m = 0

    for i in range(num_iterations) :

        derivative_x_value = function_object.get_quadratic_equation_derivative_value(x_point)

        x_m = (x_m * beta_1) + ((1 - beta_1) * derivative_x_value)

        x_v = (x_v * beta_2) + ((1 - beta_2) * ((derivative_x_value) ** 2))

        x_m_hat = x_m / (1 - ((beta_1) ** (i + 1)))

        x_v_hat = x_v / (1 - ((beta_2) ** (i + 1)))

        x_alpha = x_m_hat / (math.sqrt(x_v_hat) + epsilon)

        x_point = x_point - (x_alpha * alpha)

        x_values = np.append(x_values, [x_point], axis = 0)

        function_values = np.append(function_values, [function_object.get_quadratic_equation_value(x_point)])

    return x_values, function_values
```

Question c(i): Relu for initial condition x = −1

```
x = -1
plot.figure(figsize=(11, 6), dpi=150)
relu = Function_Relu()

x_values, function_values = execute_rmsprop_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 100)
plot.plot(function_values, lw = 4, color = 'red', label = 'Gradient Descent = RMSProp, α = 0.1, β = 0.9')

x_values, function_values = execute_heavyball_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 100)
plot.plot(function_values, lw = 3, color = 'yellow', linestyle='dashed', label = 'Gradient Descent  = Heavy Ball, α = 0.1, β = 0.9')

x_values, function_values = execute_adam_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta_1 = 0.9, beta_2 = 0.999,
num_iterations = 100)
plot.plot(function_values, lw = 2, color = 'green', linestyle='dotted', label = 'Gradient Descent  = Adam, α = 0.1, β1 = 0.9, β2 = 0.999')


plot.xlabel('#Iteration')
plot.ylabel('Function Value(x)')
plot.title(f'Plot of Function Value vs. Iteration for Relu Function for starting point x = {x}')

plot.legend()
plot.show()
```

```
x = -1
plot.figure(figsize=(11, 6), dpi=150)
relu = Function_Relu()

x_values, function_values = execute_rmsprop_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 100)
plot.plot(x_values, lw = 4, color = 'red', label = 'Gradient Descent = RMSProp, α = 0.1, β = 0.9')

x_values, function_values = execute_heavyball_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 100)
plot.plot(x_values, lw = 3, color = 'yellow', linestyle='dashed', label = 'Gradient Descent  = Heavy Ball, α = 0.1, β = 0.9')

x_values, function_values = execute_adam_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta_1 = 0.9, beta_2 = 0.999,
num_iterations = 100)
plot.plot(x_values, lw = 2, color = 'green', linestyle='dotted', label = 'Gradient Descent  = Adam, α = 0.1, β1 = 0.9, β2 = 0.999')


plot.xlabel('#Iteration')
plot.ylabel('x')
plot.title(f'Plot of X Value vs. Iteration for Relu Function for starting point x = {x}')

plot.legend()
plot.show()
```

Question c(ii): Relu for initial condition x = 1

```
x = 1
plot.figure(figsize=(11, 6), dpi=150)
relu = Function_Relu()

x_values, function_values = execute_rmsprop_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 100)
plot.plot(function_values, lw = 3, color = 'red', label = 'Gradient Descent = RMSProp, α = 0.1, β = 0.9')

x_values, function_values = execute_heavyball_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 100)
plot.plot(function_values, lw = 3, color = 'yellow', linestyle='dashed', label = 'Gradient Descent  = Heavy Ball, α = 0.1, β = 0.9')

x_values, function_values = execute_adam_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta_1 = 0.9, beta_2 = 0.999,
num_iterations = 100)
plot.plot(function_values, lw = 2, color = 'green', linestyle='dotted', label = 'Gradient Descent  = Adam, α = 0.1, β1 = 0.9, β2 = 0.999')


plot.xlabel('#Iteration')
plot.ylabel('Function Value(x)')
plot.title(f'Plot of Function Value vs. Iteration for Relu Function for starting point x = {x}')

plot.legend()
plot.show()

x = 1
plot.figure(figsize=(11, 6), dpi=150)
relu = Function_Relu()

x_values, function_values = execute_rmsprop_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 100)
plot.plot(x_values, lw = 3, color = 'red', label = 'Gradient Descent = RMSProp, α = 0.1, β = 0.9')

x_values, function_values = execute_heavyball_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 100)
plot.plot(x_values, lw = 3, color = 'yellow', linestyle='dashed', label = 'Gradient Descent  = Heavy Ball, α = 0.1, β = 0.9')

x_values, function_values = execute_adam_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta_1 = 0.9, beta_2 = 0.999,
num_iterations = 100)
plot.plot(x_values, lw = 2, color = 'green', linestyle='dotted', label = 'Gradient Descent  = Adam, α = 0.1, β1 = 0.9, β2 = 0.999')


plot.xlabel('#Iteration')
plot.ylabel('x')
plot.title(f'Plot of X Value vs. Iteration for Relu Function for starting point x = {x}')

plot.legend()
```

```
plot.show()
```

Question c(iii): Relu for initial condition x = 100

```
x = 100
plot.figure(figsize=(11, 6), dpi=150)
relu = Function_Relu()

x_values, function_values = execute_rmsprop_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 1200)
plot.plot(function_values, lw = 3, color = 'red', label = 'Gradient Descent = RMSProp, α = 0.1, β = 0.9')

x_values, function_values = execute_heavyball_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 1200)
plot.plot(function_values, lw = 3, color = 'yellow', linestyle='dashed', label = 'Gradient Descent  = Heavy Ball, α = 0.1, β = 0.9')

x_values, function_values = execute_adam_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta_1 = 0.9, beta_2 = 0.999,
num_iterations = 1200)
plot.plot(function_values, lw = 2, color = 'green', linestyle='dotted', label = 'Gradient Descent  = Adam, α = 0.1, β1 = 0.9, β2 = 0.999')


plot.xlabel('#Iteration')
plot.ylabel('Function Value(x)')
plot.title(f'Plot of Function Value vs. Iteration for Relu Function for starting point x = {x}')

plot.legend()
plot.show()

x = 100
plot.figure(figsize=(11, 6), dpi=150)
relu = Function_Relu()

x_values, function_values = execute_rmsprop_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 1200)
plot.plot(x_values, lw = 3, color = 'red', label = 'Gradient Descent = RMSProp, α = 0.1, β = 0.9')

x_values, function_values = execute_heavyball_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta = 0.9, num_iterations = 1200)
plot.plot(x_values, lw = 3, color = 'yellow', linestyle='dashed', label = 'Gradient Descent  = Heavy Ball, α = 0.1, β = 0.9')

x_values, function_values = execute_adam_gradient_descent(relu, starting_point_x = x, alpha = 0.1, beta_1 = 0.9, beta_2 = 0.999,
num_iterations = 1200)
plot.plot(x_values, lw = 2, color = 'green', linestyle='dotted', label = 'Gradient Descent  = Adam, α = 0.1, β1 = 0.9, β2 = 0.999')


plot.xlabel('#Iteration')
plot.ylabel('x')
plot.title(f'Plot of X Value vs. Iteration for Relu Function for starting point x = {x}')

plot.legend()
plot.show()
```