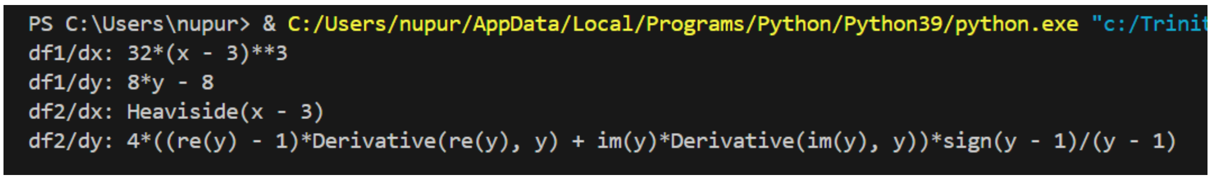
Functions:
function1: 8*(x-3) ^4+4*(y-1)^2

Function2: Max(x-3,0) +4*|y-1|

Derivative using sympy,



```
PS C:\Users\nupur> & C:/Users/nupur/AppData/Local/Programs/Python/Python39/python.exe "c:/Trinit
df1/dx: 32*(x - 3)**3
df1/dy: 8*y - 8
df2/dx: Heaviside(x - 3)
df2/dy: 4*((re(y) - 1)*Derivative(re(y), y) + im(y)*Derivative(im(y), y))*sign(y - 1)/(y - 1)
```

1)

1. Polyak

In polyak step size, the update for the parameter (param_update) is calculated using both the current gradient (grad) and a momentum. The momentum term is derived from the difference between the previous parameter value (prev_param) and the previous gradient (prev_grad). This difference is scaled by the momentum coefficient beta. The purpose of this momentum term is to smoothen the optimization path and potentially accelerate convergence, especially in scenarios with ravines or saddle points where gradients can drastically change direction. The learning rate alpha controls the size of the step in the direction of the negative gradient.

```python
def polyak_update(grad, prev_param, prev_grad, alpha=0.01, beta=0.9):
    param_update = -alpha * grad + beta * (prev_param - prev_grad)
    return param_update
```

2. RMSProp

RMSProp starts by initializing an array, s, to zero for each parameter to store the moving average of squared gradients. It then computes the square of each current gradient and merges it with s, weighted by a factor β which acts as momentum. This process creates a weighted average focusing on recent gradients. The parameters are updated using param_update, which adjusts them towards minimizing the function. The update's magnitude is inversely proportional to the square root of this weighted average, ensuring more stable and adaptive parameter adjustments.

```python
def rmsprop_update(grad, s, alpha=0.01, beta=0.9, epsilon=1e-8):
    s = beta * s + (1 - beta) * (grad ** 2)
    param_update = -alpha / (np.sqrt(s) + epsilon) * grad
    return param_update, s
```

3. Heavy Ball Method

.The method initializes an array to zero for tracking previous update steps. During each iteration, the update is determined by combining the momentum coefficient beta with the previous update and subtracting the product of the learning rate alpha and the current gradient. This approach allows the optimization to gain speed in directions with consistent gradients while smoothing out oscillations in variable terrains, by retaining a part of the previous step's momentum.

```
def heavy_ball_update(grad, prev_update, alpha=0.01, beta=0.9):
    param_update = -alpha * grad + beta * prev_update
    return param_update
```

4. Adam

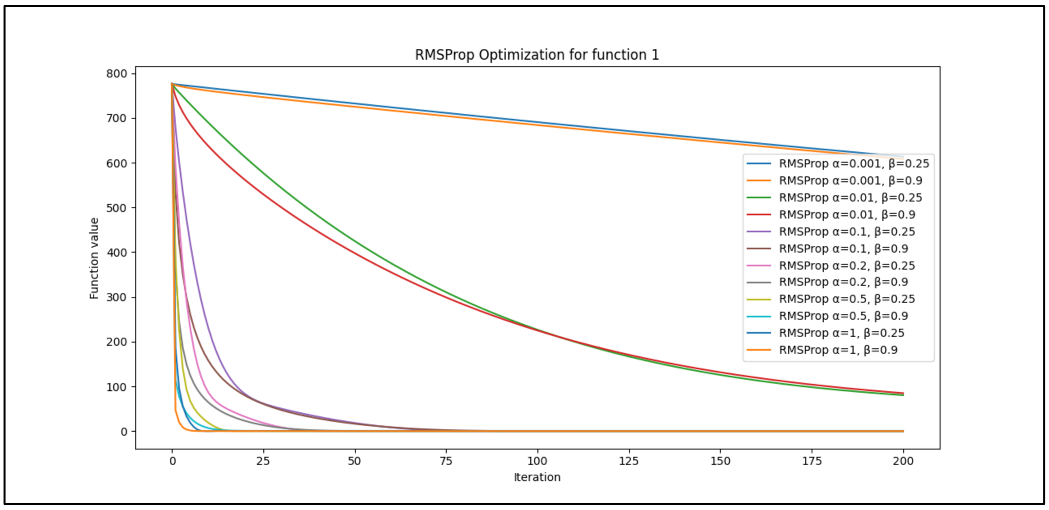
Adam optimization algorithm uses three vectors: m first moment, v second moment, and a timestep counter t, all starting with m and v as zero vectors and t at 1. In each iteration, Adam updates m and v by integrating new gradient information, scaled by factors beta1 and beta2, thus merging historical data with recent gradients. A key feature is the correction of these moments to mitigate initial bias, particularly vital early in training when the moments significantly deviate from their long-term averages. The parameter update step size is derived from these adjusted moments, where the learning rate alpha is adapted based on the square root of the second moment, with a small constant epsilon for numerical stability. This approach allows Adam to adjust step sizes adaptively, taking into account both the strength and direction of gradients, which enhances the optimization process's efficiency and robustness.

```
# Adam Function
def adam_update(grad, m, v, t, alpha=0.01, beta1=0.9, beta2=0.999, epsilon=1e-
8):
    m = beta1 * m + (1 - beta1) * grad
    v = beta2 * v + (1 - beta2) * (grad ** 2)
    m_hat = m / (1 - beta1 ** t)
    v_hat = v / (1 - beta2 ** t)
    param_update = -alpha * m_hat / (np.sqrt(v_hat) + epsilon)
    return param_update, m, v
```

2)

a)

The given plot is for function 1 for RMSProp.

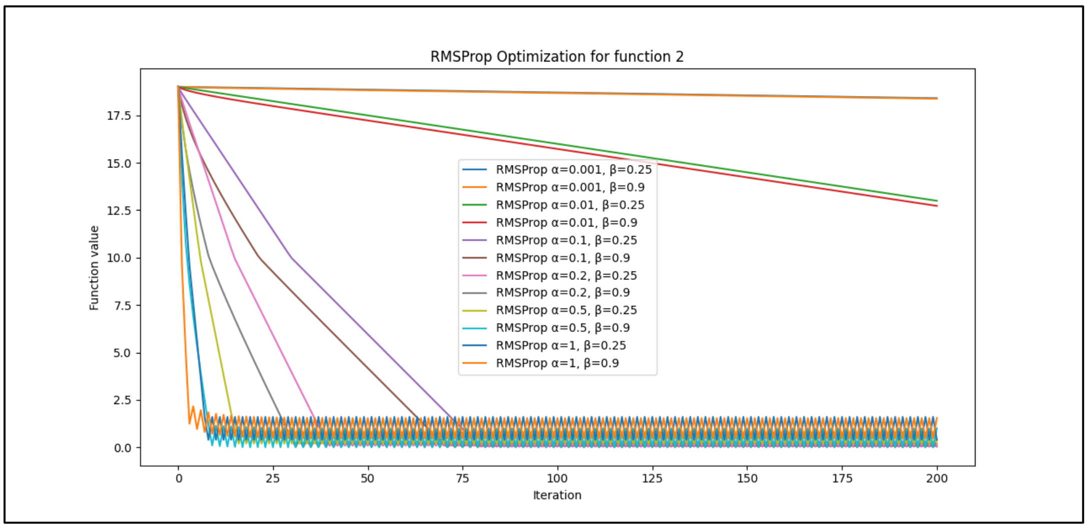


From the plot it can be seen that with a lower learning rate for example α=0.001, the convergence is generally slower. This is indicated by the more gradual decline of the function value over iterations, irrespective of the β value. As α increases, the convergence speed increases, which is evident from the steeper descent in the function value. However, when α becomes too high for example α=1, there is a risk of overshooting the minimum, which can be seen as the curves begin to flatten out or diverge after a rapid initial descent.

A higher β like 0.9 adds more smoothing to the update, which can help in scenarios where the gradient might be noisy or changing rapidly. However, too much smoothing can potentially slow down the convergence or lead to suboptimal paths. With a lower β like β=0.25, the algorithm reacts more strongly to the recent gradients, which can be beneficial for more consistent and well-behaved gradients.

From the plot we can say that an intermediate α value paired with a moderate β tends to achieve a good balance between convergence speed and stability. For example, α=0.1 and β=0.9 seem to reach a low function value quickly without the instability observed at higher α values.

The below given plot is for function 2 for RMSProp.



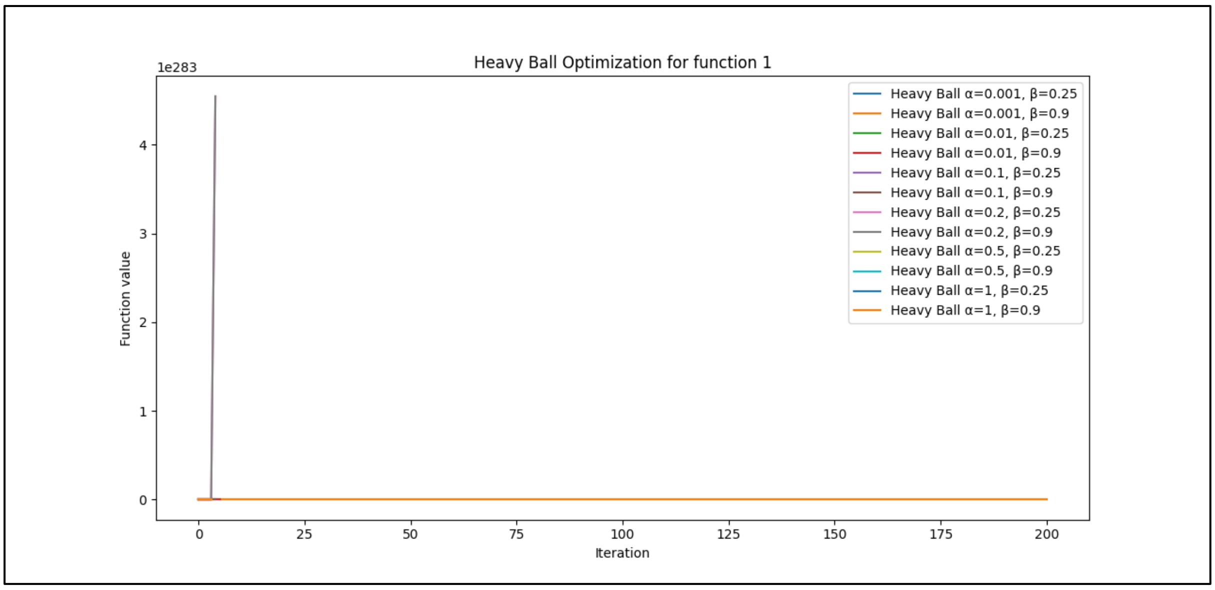
Similar to Function 1, lower α values result in slower convergence. However, for Function 2, even low α values eventually reach close to the optimal value, suggesting that Function 2 may be easier to optimize or less sensitive to the learning rate.

Increases in α show faster convergence, but for Function 2, even high learning rates do not seem to cause instability or divergence within the plotted range of iterations. This may indicate that Function 2 has a smoother landscape compared to Function 1.

The β parameter's impact is less pronounced in Function 2's optimization plot. Since Function 2 is possibly less complex or has a smoother optimization landscape, the benefits of a momentum term are not as clear-cut as in Function 1.

For Function 2, a wide range of α and β combinations appear to reach the optimal region efficiently. Lower β values do not seem to cause instability, likely due to the smoother nature of the function's gradient.

b) Plot for function 1 for heavyball



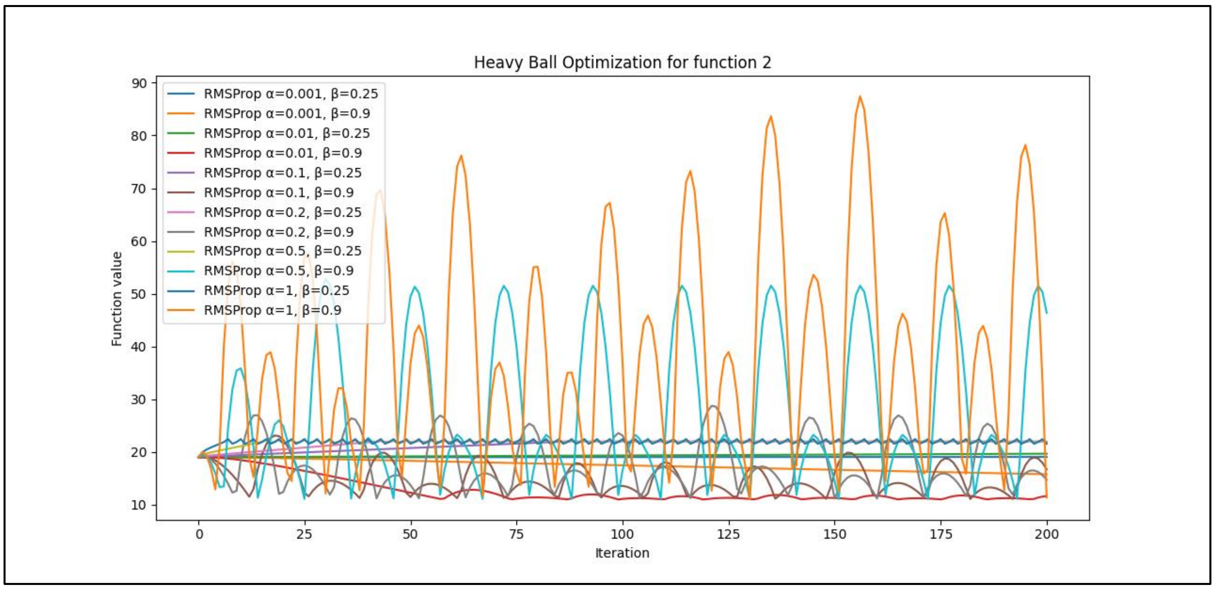
For smaller values of α like α=0.001, we see a slower descent, showing of cautious steps towards the function's minimum. The slower convergence is due to the smaller increments in parameter updates with each iteration.

Increasing α leads to a faster initial descent. However, at extremely high values like α=1, the function value dramatically increases, showing instability and divergence. This is likely due to the step size being too large, causing the algorithm to overshoot the minimum and fail to converge.

The β parameter appears to play a significant role in the convergence behavior. With a high β value , the algorithm experiences dramatic oscillations and instability, which are signs of excessive momentum leading to overshooting. In contrast, a lower β value results in a more stable but potentially slower convergence.

The plot suggests that the choice of optimal parameters involves balancing the learning rate and momentum term. Moderate values of α and β might provide a better convergence rate without leading to instability.

This plot is for function 2 of heavyball



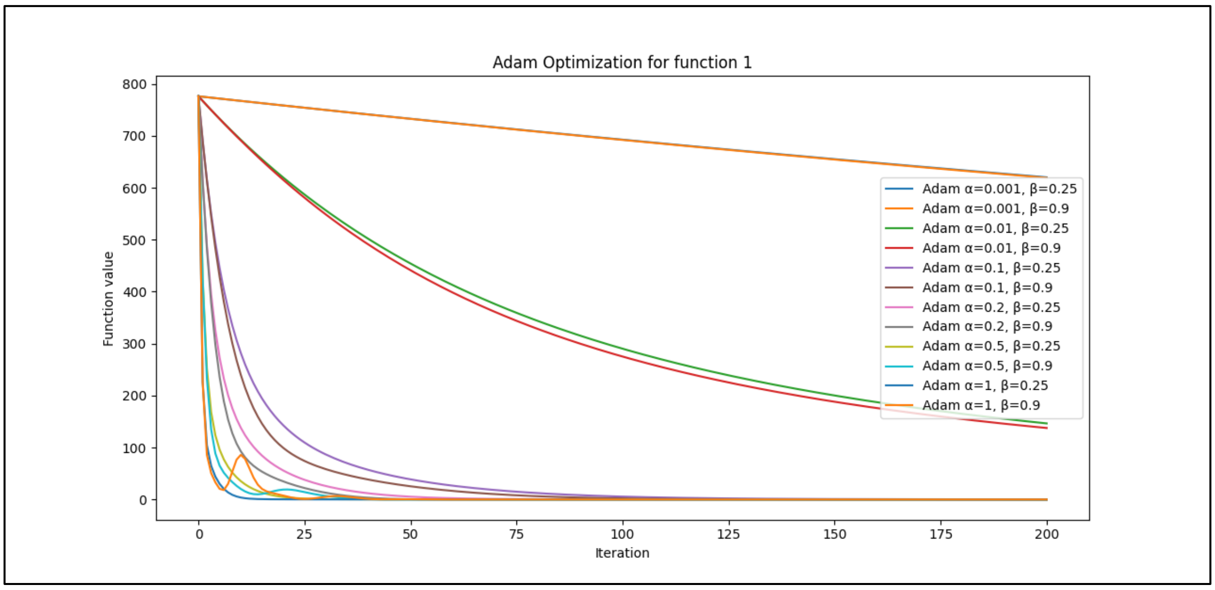
The optimization process shows a more stable and consistent convergence towards the function's minimum. This indicates that Function 2 might have a smoother landscape or a more straightforward path to the minimum.

Even with high learning rates, the algorithm doesn't exhibit the same divergence as with Function 1, but rather oscillates around the minimum. This suggests that while the step sizes are too large, leading to oscillations, they are not large enough to cause complete divergence.

The impact of the momentum term is still present; however, the function seems to be more forgiving of the higher momentum settings compared to Function 1. The oscillations in function value are prominent but do not result in the dramatic increase observed in Function 1.

For Function 2, the algorithm appears to be less sensitive to parameter settings. Moderate to high values of α with either low or high β values seem to find the function minimum without excessive instability.

c)The plot is for function 1 for Adam.



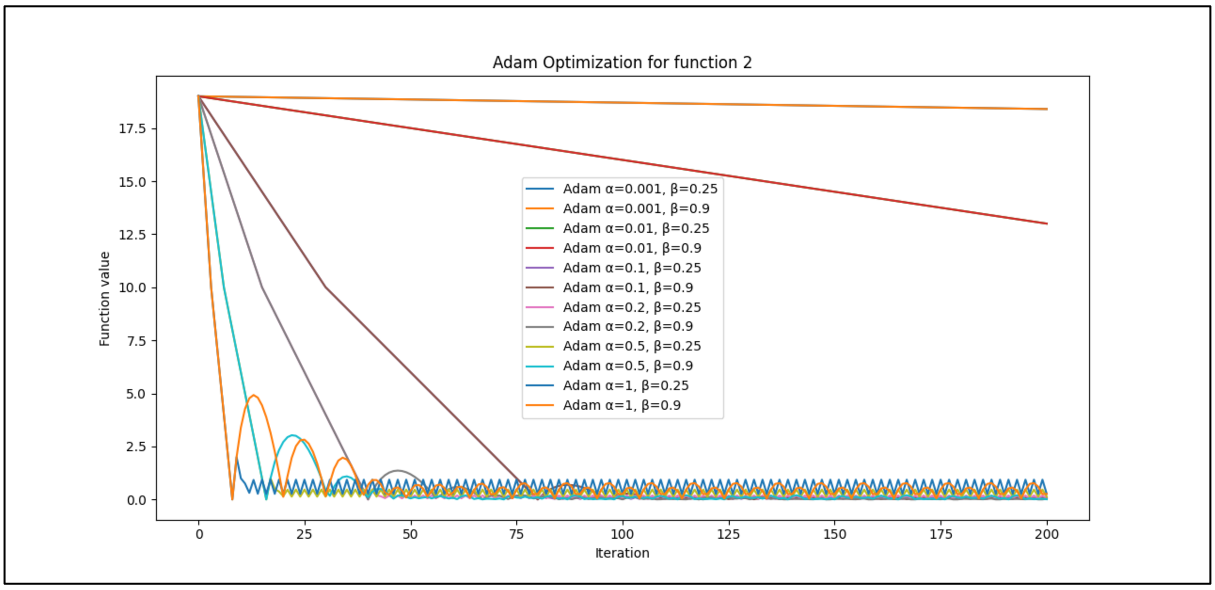
The plot shows that with a smaller α value for example α=0.001, the descent in function value is gradual, showing a careful approach that reduces the risk of overshooting the minimum. However, this way may require more iterations to converge to the minimum value, as seen in the slow downward trend.

As the learning rate α is increased, we observe a faster initial descent. This is because larger steps are taken in the direction of the gradient. The moderate learning rates like α=0.01 or α=0.1 appear to offer a good compromise between convergence speed and stability, rapidly reducing the function value without causing instability in the optimization trajectory.

When α is too high like α=1, there is noticeable instability. Even though the optimization makes quick progress initially, the function value tends to plateau or even increase slightly in later iterations, suggesting that the step sizes are too large to finely converge to the optimal value.

The plots suggest that varying β has a less dramatic impact on convergence compared to α. At higher β like β=0.9 tends to smooth out fluctuations and can lead to a more stable convergence.

The plot is for function 2 for Adam.

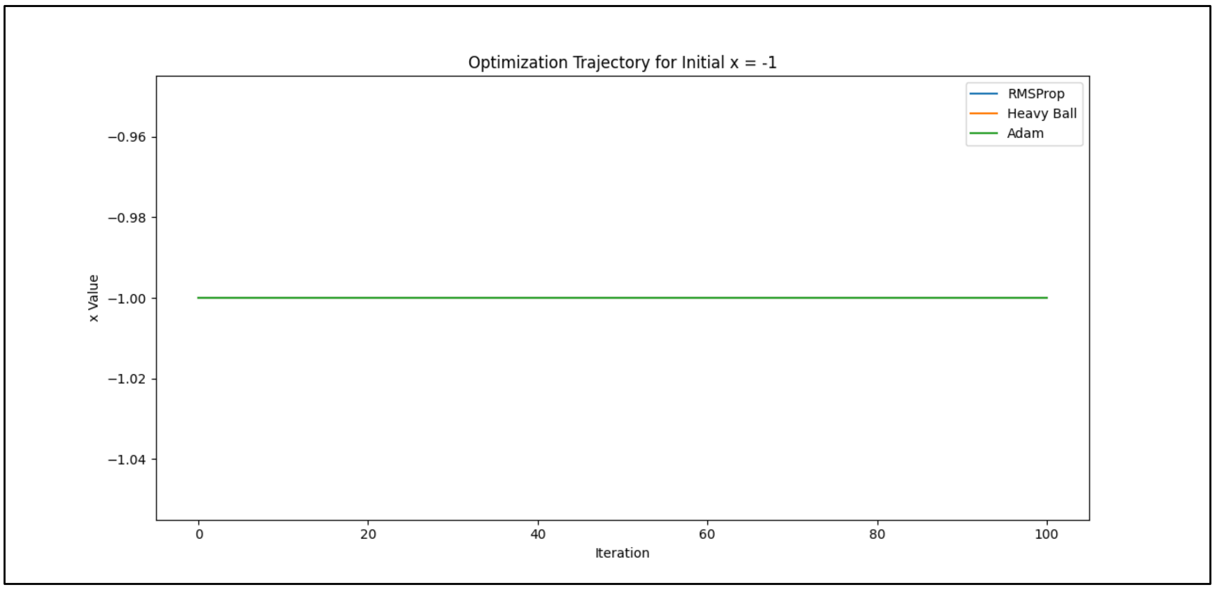


The lower learning rates shows a steady but slow convergence for Function 2. The impact of the learning rate is consistent with the behavior observed for Function 1. As the learning rate increases, the optimization trajectory for Function 2 shows quicker convergence. However, unlike Function 1, even higher values of α do not lead to the same level of instability. This suggests that Function 2 might have a smoother landscape or is less complex, making it less sensitive to larger step sizes.

For Function 2, the combination of high learning rates and the β value does not appear to show the same instability as seen with Function 1. This indicates that the chosen function and its characteristics significantly influence the optimization process's sensitivity to the algorithm's parameters.

For both functions, selecting an intermediate value of α with a relatively high β seems to offer a balance between rapid convergence and algorithmic stability. This combination allows the optimizer to quickly move towards the minimum while leveraging momentum to avoid getting stuck in suboptimal areas.

3) Initial Condition x = -1



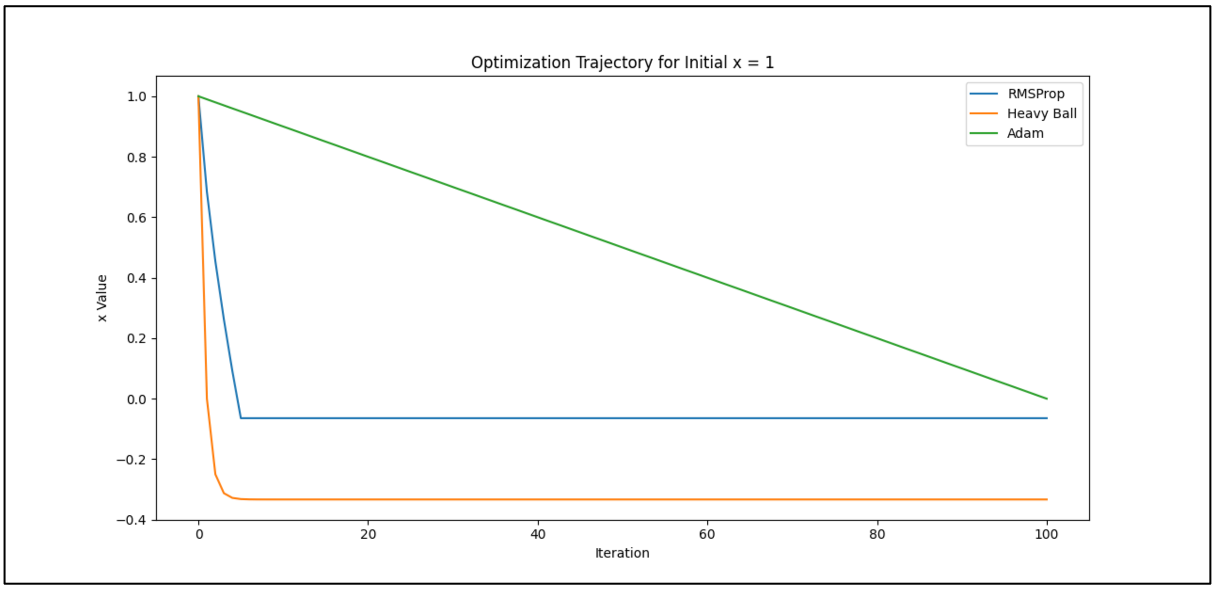
Foe RMSProp the trajectory stays constant, suggesting that the algorithm perceives it is at a local minimum due to the flat gradient (ReLU's gradient is 0 for negative values). RMSProp does not make any progress because the gradient information it uses to update the parameters is effectively zero.

Similar to RMSProp, the Heavy Ball method does not move far from the initial condition. Since the momentum term depends on previous gradients, and since the initial gradients are zero, no significant updates accumulate, resulting in minimal progress.

Adam remains stationary as well. This is due to the same reason that affects RMSProp; Adam relies on the gradients to compute the moment estimates, which are zero in this case. Hence, no substantial updates to the variable occur.

This behavior for x = -1 is primarily because, for the ReLU function, the gradient is zero when, x<0 meaning there is no direction for optimization to follow.

Initial Condition x = +1



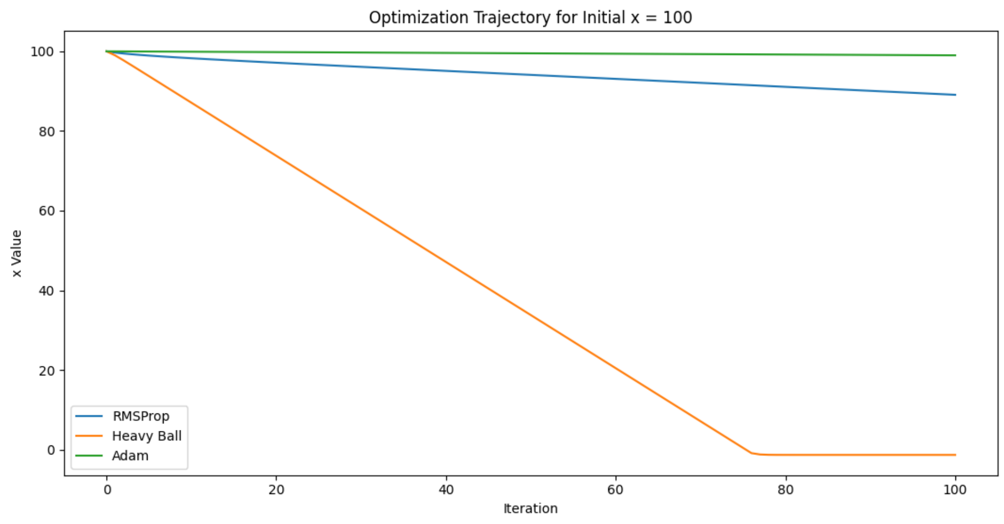
Here for RMSProp, the variable quickly converges to the optimum. RMSProp efficiently uses the non-zero gradient to update the variable towards zero, which is the minimum for the ReLU function when x>0.

The variable also converges for heavy ball but exhibits some oscillation due to the momentum term, which can overshoot the minimum and then correct itself in subsequent iterations.

Converges smoothly to the optimum without the oscillations seen in Heavy Ball, benefiting from its adaptive learning rate, which scales the updates based on the estimated moments.

In this scenario, since the gradient of the ReLU function is positive for x>0, all algorithms are able to effectively reduce the variable towards the minimum.

Initial Condition x = +100



RMSProp quickly reduces the variable value but then slows down as it approaches the minimum, reflecting the adaptive nature of RMSProp, which scales down the step size as the gradient diminishes.

Heavy Ball on other hand shows a rapid descent initially, then stabilizes around the minimum, again with minor oscillations due to the momentum term carrying the updates past the minimum before settling.

Smoothly approaches and hovers around the minimum without significant overshoot, demonstrating the effectiveness of its adaptive learning rates even with a large initial condition.

For x = +100, the steep initial gradient due to the high initial condition results in large updates initially. However, as the variable approaches the minimum, the gradient decreases, leading to smaller updates and thus a slowdown in convergence.

Appendix:

```python
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp


x, y = sp.symbols('x y')
f1 = 8*(x - 3)**4 + 4*(y - 1)**2
f2 = sp.Max(x - 3, 0) + 4 * sp.Abs(y - 1)

df1_dx = sp.diff(f1, x)
df1_dy = sp.diff(f1, y)
df2_dx = sp.diff(f2, x)
df2_dy = sp.diff(f2, y)

print("df1/dx:", df1_dx)
print("df1/dy:", df1_dy)
print("df2/dx:", df2_dx)
print("df2/dy:", df2_dy)

def compute_gradients(x, y):
    return np.array([32*x**3, 4*y - 32])

def f_new(x, y):
    return np.maximum(x - 0, 0) + 2 * np.abs(y - 8)

def f_relu(x):
    return np.maximum(0, x)

def compute_gradient_relu(x):
    return np.array([1 if x > 0 else 0])
```

```
def compute_gradients_new(x, y):
    grad_x = 1 if x > 0 else 0
    grad_y = 2 if y > 8 else (-2 if y < 8 else 0)
    return np.array([grad_x, grad_y])

def polyak(grad, prev_param, prev_grad, alpha=0.01, beta=0.9):
    param_update = -alpha * grad + beta * (prev_param - prev_grad)
    return param_update

def rmsprop(grad, s, alpha=0.01, beta=0.9, epsilon=1e-8):
    s = beta * s + (1 - beta) * (grad ** 2)
    param_update = -alpha / (np.sqrt(s) + epsilon) * grad
    return param_update, s


def heavy_ball(grad, prev_update, alpha=0.01, beta=0.9):
    param_update = -alpha * grad + beta * prev_update
    return param_update

# 4)
# Adam Function
def adam(grad, m, v, t, alpha=0.01, beta1=0.9, beta2=0.999, epsilon=1e-8):
    m = beta1 * m + (1 - beta1) * grad
    v = beta2 * v + (1 - beta2) * (grad ** 2)
    m_hat = m / (1 - beta1 ** t)
    v_hat = v / (1 - beta2 ** t)
    param_update = -alpha * m_hat / (np.sqrt(v_hat) + epsilon)
    return param_update, m, v

def optimize_and_plot(function, gradient_function, update_function, params,
alpha, beta, beta2=None, algorithm_name='RMSProp', iterations=200):

    values = [function(*params)]
    if algorithm_name == 'Adam':
        m, v = np.zeros_like(params), np.zeros_like(params)
    elif algorithm_name == 'RMSProp':
        s = np.zeros_like(params)
    prev_update = np.zeros_like(params)

    for t in range(1, iterations + 1):
        grad = gradient_function(*params)
        if algorithm_name == 'RMSProp':
            param_update, s = update_function(grad, s, alpha, beta)
        elif algorithm_name == 'Heavy Ball':
            param_update = update_function(grad, prev_update, alpha, beta)
        elif algorithm_name == 'Adam':
            param_update, m, v = update_function(grad, m, v, t, alpha, beta,
beta2)
```

```
        params += param_update
        prev_update = param_update
        values.append(function(*params))

    # Plotting
    plt.plot(values, label=f'{algorithm_name} α={alpha}, β={beta}')


def f(x, y):
    return 8*(x-0)**4 + 2*(y-8)**2

initial_params = [3,0]
alphas = [0.001,0.01, 0.1,0.2,0.5, 1]
betas = [0.25, 0.9]

# Optimization and plot for RMSProp for function 1
plt.figure(figsize=(12, 8))
for alpha in alphas:
    for beta in betas:
        optimize_and_plot(f, compute_gradients, rmsprop,
initial_params.copy(), alpha, beta, algorithm_name='RMSProp')

plt.title('RMSProp Optimization for function 1')
plt.xlabel('Iteration')
plt.ylabel('Function value')
plt.legend()
plt.show()
# Optimization and plot for RMSProp with function 2
plt.figure(figsize=(12, 8))
for alpha in alphas:
    for beta in betas:
        optimize_and_plot(f_new, compute_gradients_new, rmsprop,
initial_params.copy(), alpha, beta, algorithm_name='RMSProp')

plt.title('RMSProp Optimization for function 2')
plt.xlabel('Iteration')
plt.ylabel('Function value')
plt.legend()
plt.show()



# Optimization and plot for Heavy Ball with function 1
plt.figure(figsize=(12, 8))
for alpha in alphas:
    for beta in betas:
        optimize_and_plot(f, compute_gradients, heavy_ball,
initial_params.copy(), alpha, beta, algorithm_name='Heavy Ball')

plt.title('Heavy Ball Optimization for function 1')
plt.xlabel('Iteration')
plt.ylabel('Function value')
plt.legend()
plt.show()

# Optimization and plot for RMSProp with function 2
plt.figure(figsize=(12, 8))
for alpha in alphas:
    for beta in betas:
        optimize_and_plot(f_new, compute_gradients_new, heavy_ball,
initial_params.copy(), alpha, beta, algorithm_name='RMSProp')

plt.title('Heavy Ball Optimization for function 2')
plt.xlabel('Iteration')
plt.ylabel('Function value')
plt.legend()
plt.show()


# Optimization and plot for Adam with function 1
plt.figure(figsize=(12, 8))
beta2 = 0.999
for alpha in alphas:
    for beta in betas:
        optimize_and_plot(f, compute_gradients, adam, initial_params.copy(),
alpha, beta, beta2=beta2, algorithm_name='Adam')

plt.title('Adam Optimization for function 1')
plt.xlabel('Iteration')
plt.ylabel('Function value')
plt.legend()
plt.show()

# Optimization and plot for Adam with function 2
plt.figure(figsize=(12, 8))
beta2 = 0.999
for alpha in alphas:
    for beta in betas:
        optimize_and_plot(f_new, compute_gradients_new, adam,
initial_params.copy(), alpha, beta, beta2=beta2, algorithm_name='Adam')

plt.title('Adam Optimization for function 2')
plt.xlabel('Iteration')
plt.ylabel('Function value')
plt.legend()
plt.show()

def f_relu(x):
    return np.maximum(x, 0)

def compute_gradient_relu(x):
    return np.heaviside(x,0)


def rmsprop_single(x, grad, s, alpha, beta, epsilon=1e-8):
    s = beta * s + (1 - beta) * grad**2
    x_update = -alpha / (np.sqrt(s) + epsilon) * grad
    return x + x_update, s

def heavy_ball_single(x, grad, prev_update, alpha, beta):
    x_update = -alpha * grad + beta * prev_update
    return x + x_update, x_update

def adam_single(x, grad, m, v, t, alpha, beta1, beta2, epsilon=1e-8):
    m = beta1 * m + (1 - beta1) * grad
    v = beta2 * v + (1 - beta2) * grad**2
    m_hat = m / (1 - beta1**t)
    v_hat = v / (1 - beta2**t)
    x_update = -alpha * m_hat / (np.sqrt(v_hat) + epsilon)
    return x + x_update, m, v


def optimize(x_init, grad_func, update_func, update_params, iterations=100):
    x = x_init
    history = [x]

    if update_func.__name__.startswith('adam'):
        m, v = 0, 0
    else:
        prev_update = 0
    for t in range(1, iterations + 1):
        grad = grad_func(x)
        if update_func.__name__.startswith('adam'):
            x, m, v = update_func(x, grad, m, v, t, *update_params)
        else:
            x, prev_update = update_func(x, grad, prev_update, *update_params)
        history.append(x)
    return history
# Parameters
iterations = 100
x_inits = [-1, 1, 100]
alpha = 0.1
alphaadam = 0.01
alphaheavy = 1
```

```
betaheavy = 0.25
beta = 0.9
beta2 = 0.99

for x_init in x_inits:
    plt.figure(figsize=(12, 6))

    # RMSProp
    history = optimize(x_init, compute_gradient_relu, rmsprop_single, [alpha,
beta], iterations)
    plt.plot(history, label='RMSProp')

    # Heavy Ball
    history = optimize(x_init, compute_gradient_relu, heavy_ball_single,
[alphaheavy, betaheavy], iterations)
    plt.plot(history, label='Heavy Ball')

    # Adam
    history = optimize(x_init, compute_gradient_relu, adam_single, [alphaadam,
beta, beta2], iterations)
    plt.plot(history, label='Adam')

    plt.title(f'Optimization Trajectory for Initial x = {x_init}')
    plt.xlabel('Iteration')
    plt.ylabel('x Value')
    plt.legend()
    plt.show()
```