Optimisation Algorithms for Data Analysis - Assignment 2

CS7DS2

Maurice Kiely - 23350601

03/03/2024

Question a)

I began by defining 2 classes to represent my 2 functions.

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

```
class func1:
    def __init__(self):
        self.x_sym, self.y_sym = sympy.symbols('x y', real=True)
    def f(self, x, y):
        return 5 * (x - 3)**4 + 7 * (y - 9)**2
    def dx(self, x, y):
        df_dx = sympy.diff(self.f(self.x_sym, self.y_sym),
            self.x_sym)
        return df_dx.subs({self.x_sym: x, self.y_sym: y})
    def dy(self, x, y):
        df_dy = sympy.diff(self.f(self.x_sym, self.y_sym),
            self.y_sym)
        return df_dy.subs({self.x_sym: x, self.y_sym: y})

class func2:
    def __init__(self):
        self.x_sym, self.y_sym = sympy.symbols('x y', real=True)
    def f(self, x, y):
        return sympy.Max(x - 3, 0) + 7 * sympy.Abs(y - 9)
    def dx(self, x, y):
        df_dx = sympy.diff(self.f(self.x_sym, self.y_sym),
            self.x_sym)
        return df_dx.subs({self.x_sym: x, self.y_sym: y})
    def dy(self, x, y):
        df_dy = sympy.diff(self.f(self.x_sym, self.y_sym),
            self.y_sym)
        return df_dy.subs({self.x_sym: x, self.y_sym: y})
```

These classes are used to conveniently find the gradients at each (x, y) point. This obviously saves a lot of time when implementing the different methods. My functions were:

1. $5(x-3)^4 + 7(y-9)^2$

2. $(x-3)^+ + 7|y-9|$

i) Polyak

```
def polyak_GD(func, x0, y0, f_star=0, epsilon=0.0001,
    max_iters=1000, tol=100):
    # Initialize x,y and path and f(x,y) lists
    x, y = x0, y0
    path = [(x, y)]
    f_vals = []

    # Implement formula to find alpha and update at each step.
    for i in range(max_iters):
        df_dx = func.dx(x, y)
        df_dy = func.dy(x, y)
        alpha = (func.f(x, y) - f_star) / (df_dx**2 + df_dy**2 +
            epsilon)
        x -= alpha * df_dx
        y -= alpha * df_dy
        # Append both (x, y) point and f value to lists
        path.append((x, y))
        f_vals.append(func.f(x, y))

        # Check stopping conditions for alpha too high or too low
        if alpha > tol:
            print(f"Stopping after {i} Iterations as alpha too
                large")
            return path, f_vals
        if sympy.sqrt(func.dx(x, y)**2 + func.dy(x, y)**2) <
            epsilon:
            return path, f_vals


    print(f"No sufficient convergence by iteration
        {max_iters}\nFinished at (x, y): ({x:0.4f}, {y:0.4f})")
    return path, f_vals
```

The polyak function begins by initializing the variables for the first pass of calculating alpha, as well as those to store the path taken by f(x,y). The algorithm works by using the formula $\alpha = \frac{f(x)-f^*}{\nabla f(x)^T \nabla f(x)+\epsilon}$. This function calculates the step size by transforming $f(x - \alpha \nabla f(x)) \approx f^*$. We update x iteratively as such.

As stopping criterion, we use $\|\nabla f(x)\| < \epsilon$. Here, if the gradient isn't changing, we stop. This is because, this leads to an extremely large alpha otherwise which can introduce 'noise' around the solution.

ii) RMSProp

```
def rmsprop(func, x0, y0, alpha0, beta, epsilon=0.0001,
   max_iters=1000):
    # Initialize x,y, sums and path,f(x,y) lists
    x, y = x0, y0
    sum_dx = 0
    sum_dy = 0
    path = [(x, y)]
    f_vals = []

    for t in range(1, max_iters + 1):
        # Take gradients for current x, y
        grad_x = func.dx(x, y)
        grad_y = func.dy(x, y)

        # Update denominator for x and y using the following
        sum_dx = beta * sum_dx + (1 - beta) * grad_x**2
        sum_dy = beta * sum_dy + (1 - beta) * grad_y**2

        # Compute the step sizes for x and y
        step_size_x = alpha0 / sympy.sqrt(sum_dx) + epsilon
        step_size_y = alpha0 / sympy.sqrt(sum_dy) + epsilon

        # Update x and y
        x -= step_size_x * grad_x
        y -= step_size_y * grad_y

        path.append((x, y))
        f_vals.append(func.f(x, y))

    return path, f_vals
```

The RMSProp method uses the gradient from each variable to calculate a vector of step sizes. This idea is taken from the Adagard, however, the RMSProp method adds a decaying memory property to put more emphasis on step sizes based on more recent gradients. This algorithm begins by initializing all necessary variables including the sums of gradients for each variable. At each iteration, the gradient is calculated and the sum variable is updated. This is done by taking a β weighted average of current and past gradients. β can be adjusted accordingly to emphasise more recent gradients or not. α is a learning rate that dictates how impactful more recent gradients are. We use this calculated sum to get the step size and update x accordingly.

iii) Heavy Ball

```
def heavy_ball(func, x0, y0, alpha, beta, num_iters=1000,
   tol=1e-5, stopping=True):
    # Initialize x,y, momentum variables and path,f(x,y) lists
    x, y = x0, y0
    ux, uy = 0, 0
    path = [(x, y)]
    f_vals = []

    for i in range(num_iters):
        # Compute the gradients
        grad_x = func.dx(x, y)
        grad_y = func.dy(x, y)
        # Update momentum variable with current and past momentum
        ux = beta * ux + alpha * grad_x
        uy = beta * uy + alpha * grad_y
        # Update x and y
        x -= ux
        y -= uy
        path.append((x, y))  # Store the new values
        f_vals.append(func.f(x, y))

        if func.f(x, y) > 1e20:
            break
        # Check if the norm of the gradient is below the
           tolerance
        if sympy.sqrt(grad_x**2 + grad_y**2) < tol and stopping
           == True:
            break
    return path, f_vals
```

Heavy Ball/Polyak Momentum is an algorithm that uses 1 step size for all variables at each iteration. However, unlike standard polyak, its step size is influenced by values of past gradients. At step t, the step size is a β weighted average of past gradients and learning rate $\alpha\times$gradient. We implement this algorithm by initializing the variables including the momentum variables. We then begin iteration. Each iteration begins by calculating the gradient at each (x,y). These are then added to the momentum variables with weight $0 \leq \beta \leq 1$ which is taken as an argument. These momentum-influenced step sizes are then taken from x and y.

Similar to the Polyak algorithm, we implement a stopping criterion of $\|\nabla f(x)\| < \epsilon$.

This avoids noise around convergence although a good choice of β will minimise this.

iii) Adam

```
def adam(func, x0, y0, alpha, beta1, beta2, epsilon=1e-6,
   num_iters=1000):
    # Initialize x,y, momentum variables and path,f(x,y) lists
    x, y = x0, y0
    m_x, m_y = 0, 0
    v_x, v_y = 0, 0
    path = [(x, y)]
    f_vals = []

    for t in range(1, num_iters + 1):
        # Take gradients for current x, y
        grad_x = func.dx(x, y)
        grad_y = func.dy(x, y)
        # Update momentum variable for gradient
        m_x = beta1 * m_x + (1 - beta1) * grad_x
        m_y = beta1 * m_y + (1 - beta1) * grad_y
       # Update momentum variable for squared gradient
        v_x = beta2 * v_x + (1 - beta2) * grad_x**2
        v_y = beta2 * v_y + (1 - beta2) * grad_y**2
        # Compute bias-corrected estimates
        m_x_hat = m_x / (1 - beta1**t)
        m_y_hat = m_y / (1 - beta1**t)

        v_x_hat = v_x / (1 - beta2**t)
        v_y_hat = v_y / (1 - beta2**t)

        # Update variables
        x -= alpha * m_x_hat / (sympy.sqrt(v_x_hat) + epsilon)
        y -= alpha * m_y_hat / (sympy.sqrt(v_y_hat) + epsilon)
        if func.f(x, y) > 1e20:
            break

        path.append((x, y))
        f_vals.append(func.f(x, y))

    return path, f_vals
```

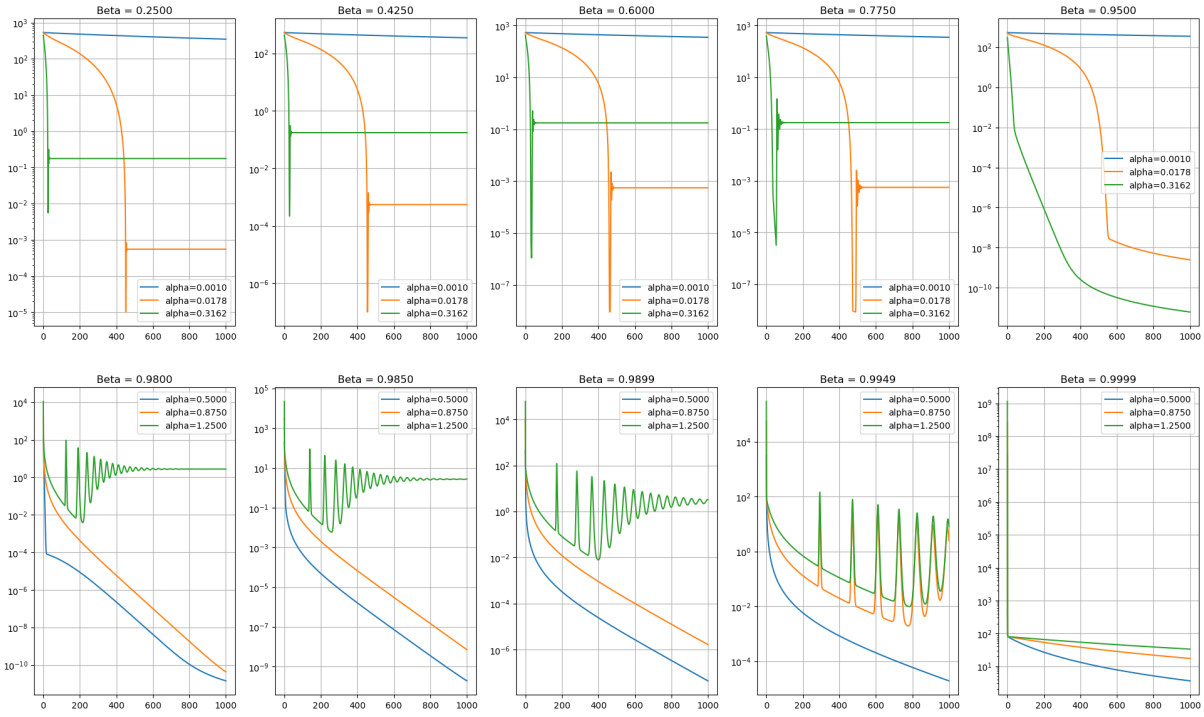
The Adam algorithm is a combination of RMSProp and HeavyBall. It takes the momentum aspect of Heavy ball and applies it to a vector of step sizes corresponding to each variable. This algorithm is implemented by initializing all variables including the momentum variables. Once iteration begins, the first step is to calculate the gradients of x and y. These are used to update the momentum variables based on the weight of β_1. This is also done with respect to the squared gradients using β_2. These updated momentum variables are then bias-corrected and used along with the learning rate α to update x and y.

Question b)

i) RMSProp

To find a rough idea of what values of α and β to use, I used an iterative grid search on a linear scale for β and a log scale for α. This was done twice with the second iteration narrowing around the most accurate values.

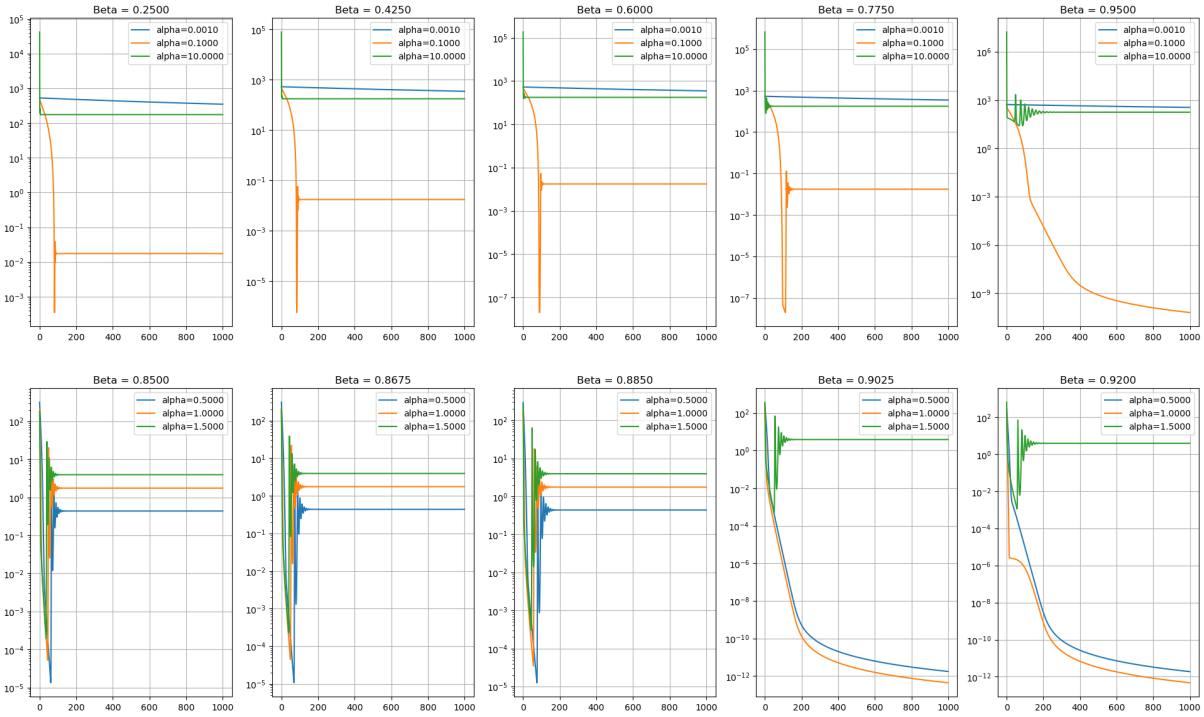
Function 1: alpha vs beta



The top 5 images all show a large spread of β. This gives Adagard-like results. There is a clear difference at later iterations of the chatter being far more prominent in RMSProp than Adagard. This is because the step size isn't reduced enough and too much influence is taken from past gradients. We can see, however, that a higher β does cause a quicker

conversion overall. This is to be expected as the gradient slowing doesn't impact the algorithms with higher β as strongly as those with lower β.

Function 2: alpha vs beta



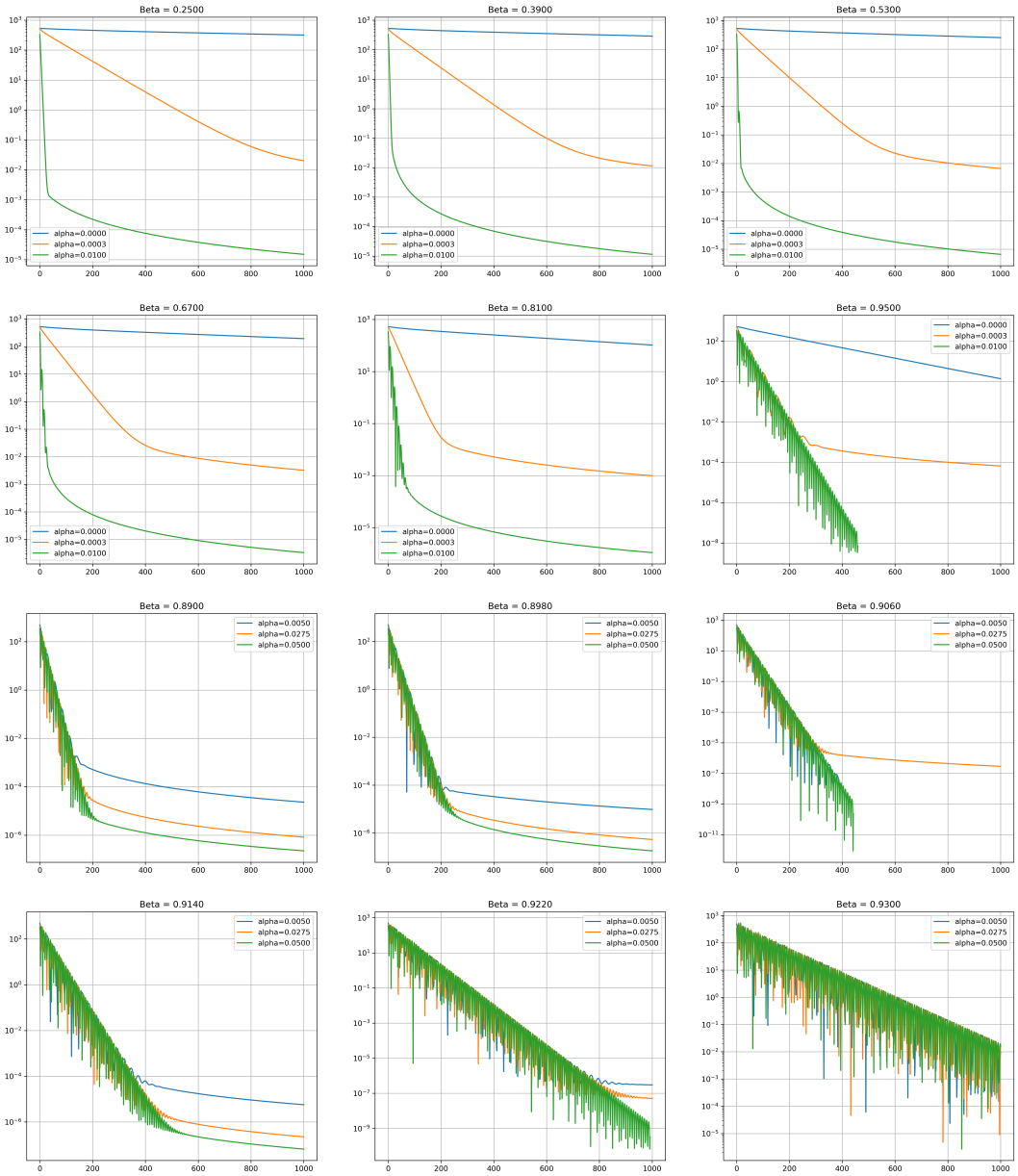
We can also see that lower values of α give a more stable result. We can see that very low value lead to a slow convergence when β is small, however, for higher β, the stability is far more valuable. The same results remain true for the piecewise function.
Also, a contour plot here would show a more direct path to the local minimum than a constant or Polyak.

ii) Heavy Ball

Like the RMSProp implementation I used an iterative grid search to find values of α and β that performed well.

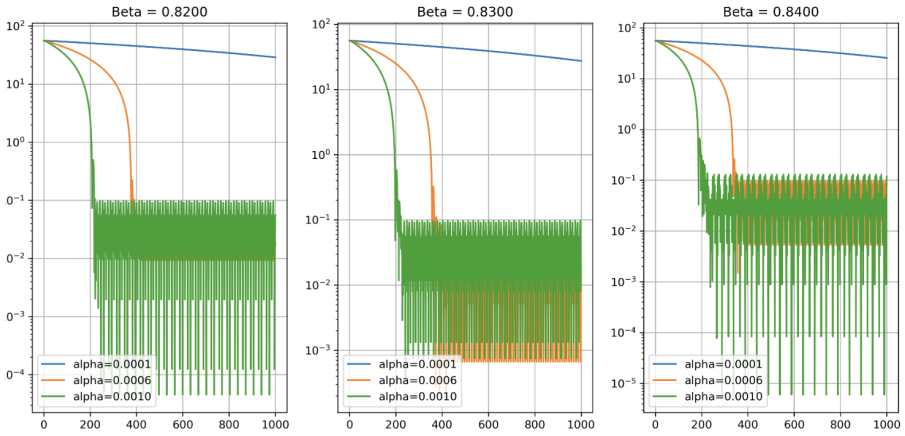
We can see in the initial 6 plots that a lower value of β gives a more stable result. This is because there are going to be less extreme overshooting oscillations of the x and y values. For higher values of β, it takes longer to change the direction of the momentum due to the large impact that a variable will have in the other direction once it hits its minimum for a given variable. We can see that this introduces a lot of oscillation which can be seen by the 'bumps' in the graph. α here is the learning rate which controls rate of convergence to a minimum. However, a larger value of α also introduces a higher likelihood of instability and noise.

Function 1: alpha vs beta



The plot of the second function shows results essentially consistent with those above. In this plot, the bumps/oscillations are more extreme due to the kinks in the function and oscillations don't decay to 0. This is due to the kink around the minimum causing the same gradients to be called repeatedly as (x,y) cross it. These results are all unstable and heavy ball is not a good method for piecewise functions.

A contour plot would show oscillations over and back along the path of the gradient with intensity relative to β, however, I had no room to include these.

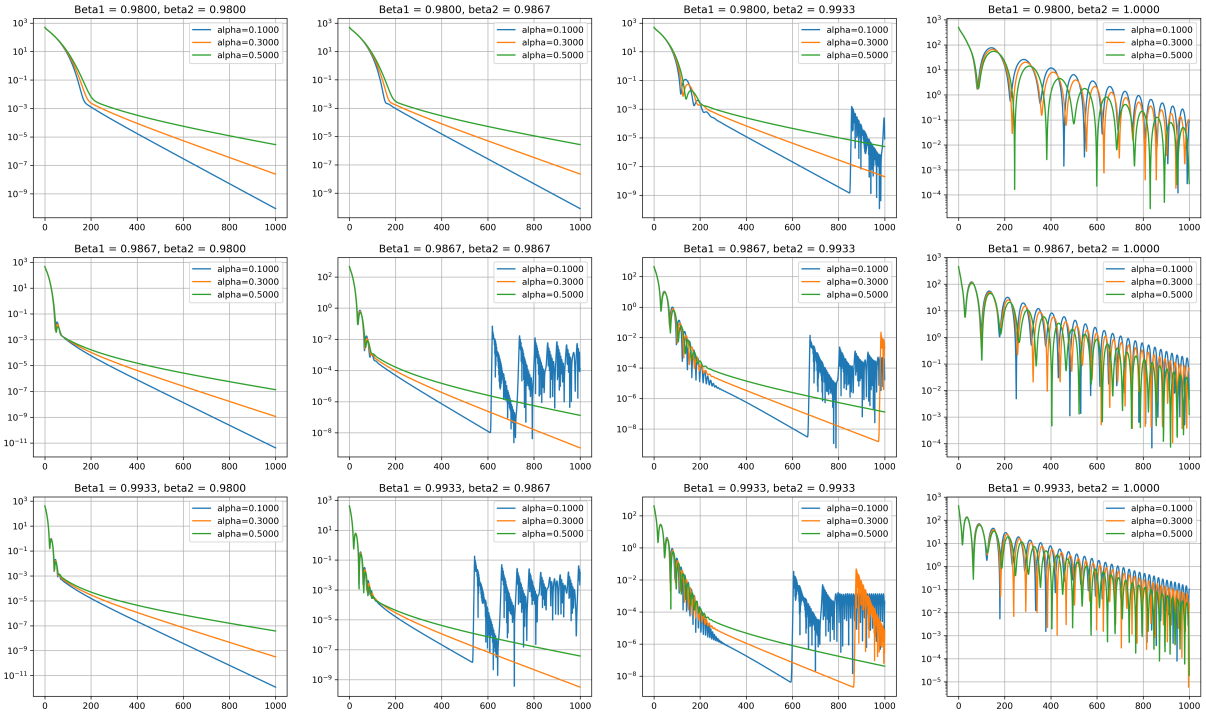


iii) Adam

Like the previous 2 algorithms, we can see that a lower learning rate, α, leads to slower convergence. Although it can't be seen in this example, if α is too big it will cause the solution to diverge.

It can be seen that the general 'bumpy' behaviour only occurs for $\beta_2 \approx 1$. This happens because the increased weight given to the past squared gradients amplifies the oscillations. When $\beta_2 \approx 1$, these values essentially don't decay, or at least decay extremely slowly. The β_1 values also cause more oscillations as they increase but it is far less impactful than β_2.

Function 1: alpha vs beta1 vs beta2



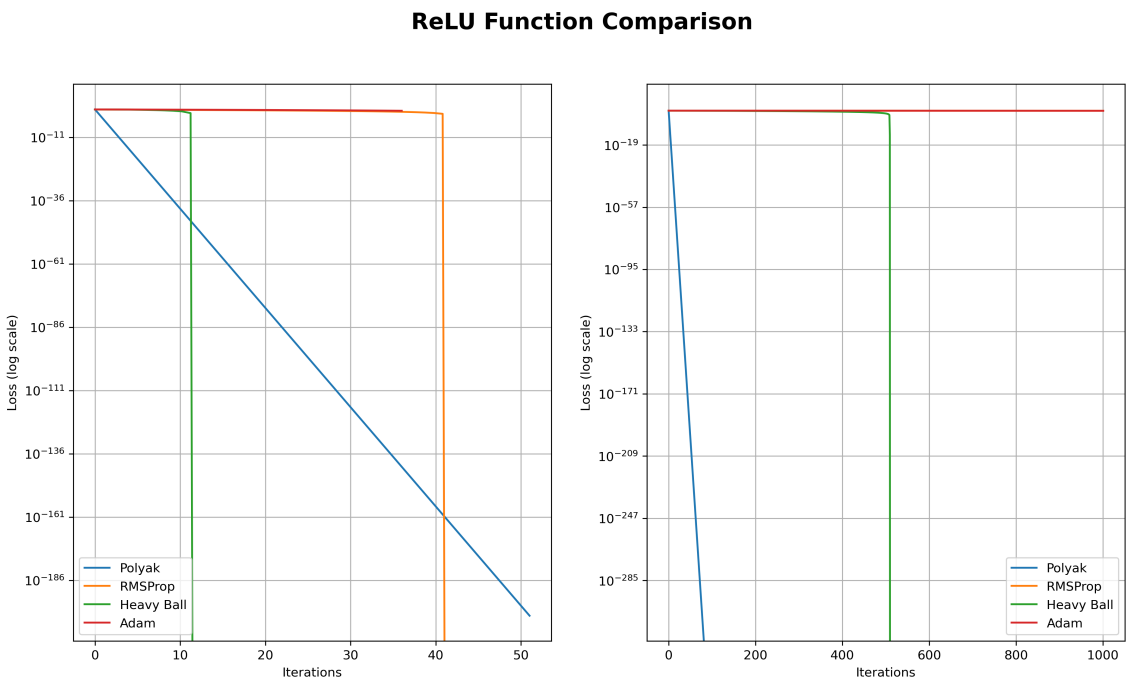
These oscillations would also be very apparent in a contour plot.

Question c)

i) ReLU (-1)

Each of the four algorithms give values of $(x, f(x)) = (-1, 0)$ for all iterations. This is because every one of the functions require the $\frac{df(x)}{dx} \neq 0$ for the x value to change. Each change to the step size is some multiplicative function of the gradient and hence each time the step size changes, it changes by 0.

ii,iii) ReLU (+1, +100)



x0 = +1 (left), x0 = +100 (right)

On the left image, we would expect polyak and heavy ball to fall to 0 essentially after the first iteration. $1 - (\frac{1-0}{1^2})(1) = 0$. However, the reason it takes longer is due to the ϵ that makes it logarithmic. This could be omitted. The adam and RMSProp functions would require a higher value of α to increase the stepsize and decrease to 0 quicker.

Code

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

```

```
5
6  class func1:
7      def __init__(self):
8          self.x_sym, self.y_sym = sympy.symbols('x y', real=True)
9
10     def f(self, x, y):
11         return 5 * (x - 3)**4 + 7 * (y - 9)**2
12
13     def dx(self, x, y):
14         df_dx = sympy.diff(self.f(self.x_sym, self.y_sym),
               self.x_sym)
15         return df_dx.subs({self.x_sym: x, self.y_sym: y})
16
17     def dy(self, x, y):
18         df_dy = sympy.diff(self.f(self.x_sym, self.y_sym),
               self.y_sym)
19         return df_dy.subs({self.x_sym: x, self.y_sym: y})
20
21
22 class func2:
23     def __init__(self):
24         self.x_sym, self.y_sym = sympy.symbols('x y', real=True)
25
26     def f(self, x, y):
27         return sympy.Max(x - 3, 0) + 7 * sympy.Abs(y - 9)
28
29     def dx(self, x, y):
30         df_dx = sympy.diff(self.f(self.x_sym, self.y_sym),
               self.x_sym)
31         return df_dx.subs({self.x_sym: x, self.y_sym: y})
32
33     def dy(self, x, y):
34         df_dy = sympy.diff(self.f(self.x_sym, self.y_sym),
               self.y_sym)
35         return df_dy.subs({self.x_sym: x, self.y_sym: y})
```

```
1  def polyak_GD(func, x0, y0, f_star=0, epsilon=0.0001,
       max_iters=1000, tol=100):
2      # Initialize x,y and path and f(x,y) lists
3      x, y = x0, y0
4      path = [(x, y)]
5      f_vals = []
```

```
6
7       # Implement formula to find alpha and update at each step.
8       for i in range(max_iters):
9           df_dx = func.dx(x, y)
10          df_dy = func.dy(x, y)
11          alpha = (func.f(x, y) - f_star) / (df_dx**2 + df_dy**2 +
                epsilon)
12          # Update x and y based on polyak step - optimal alpha
13          x -= alpha * df_dx
14          y -= alpha * df_dy
15          # Append both (x, y) point and f value to lists
16          path.append((x, y))
17          f_vals.append(func.f(x, y))
18
19          # Check stopping conditions for alpha too high or too low
20          if alpha > tol:
21              print(f"Stopping after {i} Iterations as alpha too
                    large")
22              return path, f_vals
23          if sympy.sqrt(func.dx(x, y)**2 + func.dy(x, y)**2) <
                epsilon:
24              return path, f_vals
25
26
27      print(f"No sufficient convergence by iteration
            {max_iters}\nFinished at (x, y): ({x:0.4f}, {y:0.4f})")
28      return path, f_vals
29
30
31  def rmsprop(func, x0, y0, alpha0, beta, epsilon=0.0001,
        max_iters=1000):
32      # Initialize x,y, sums and path,f(x,y) lists
33      x, y = x0, y0
34      sum_dx = 0
35      sum_dy = 0
36      path = [(x, y)]
37      f_vals = []
38
39      for t in range(1, max_iters + 1):
40          # Take gradients for current x, y
41          grad_x = func.dx(x, y)
```

```
        grad_y = func.dy(x, y)

        # Update denominator for x and y using the following
        sum_dx = beta * sum_dx + (1 - beta) * grad_x**2
        sum_dy = beta * sum_dy + (1 - beta) * grad_y**2

        # Compute the step sizes for x and y
        step_size_x = alpha0 / sympy.sqrt(sum_dx + epsilon)
        step_size_y = alpha0 / sympy.sqrt(sum_dy + epsilon)

        # Update x and y
        x -= step_size_x * grad_x
        y -= step_size_y * grad_y

        path.append((x, y))
        f_vals.append(func.f(x, y))

    return path, f_vals


def heavy_ball(func, x0, y0, alpha, beta, num_iters=1000,
    tol=1e-5, stopping=True):
    # Initialize x,y, momentum variables and path,f(x,y) lists
    x, y = x0, y0
    ux, uy = 0, 0
    path = [(x, y)]
    f_vals = []

    for i in range(num_iters):
        # Take gradients for current x, y
        grad_x = func.dx(x, y)
        grad_y = func.dy(x, y)

        # Update momentum variable with current and past momentum
        ux = beta * ux + alpha * grad_x
        uy = beta * uy + alpha * grad_y

        # Update x and y
        x -= ux
        y -= uy

```

```
82          path.append((x, y))  # Store the new values
83          f_vals.append(func.f(x, y))
84
85          if func.f(x, y) > 1e20:
86              break
87
88          # Check if the norm of the gradient is below the
               tolerance
89          if sympy.sqrt(grad_x**2 + grad_y**2) < tol and stopping
               == True:
90              break
91
92      return path, f_vals
93
94
95
96  def adam(func, x0, y0, alpha, beta1, beta2, epsilon=1e-6,
       num_iters=1000):
97      # Initialize x,y, momentum variables and path,f(x,y) lists
98      x, y = x0, y0
99      m_x, m_y = 0, 0
100     v_x, v_y = 0, 0
101     path = [(x, y)]
102     f_vals = []
103
104     for t in range(1, num_iters + 1):
105         # Take gradients for current x, y
106         grad_x = func.dx(x, y)
107         grad_y = func.dy(x, y)
108         # Update momentum variable for gradient
109         m_x = beta1 * m_x + (1 - beta1) * grad_x
110         m_y = beta1 * m_y + (1 - beta1) * grad_y
111        # Update momentum variable for squared gradient
112         v_x = beta2 * v_x + (1 - beta2) * grad_x**2
113         v_y = beta2 * v_y + (1 - beta2) * grad_y**2
114         # Compute bias-corrected estimates
115         m_x_hat = m_x / (1 - beta1**t)
116         m_y_hat = m_y / (1 - beta1**t)
117
118         v_x_hat = v_x / (1 - beta2**t)
119         v_y_hat = v_y / (1 - beta2**t)
```

```
        # Update variables
        x -= alpha * m_x_hat / (sympy.sqrt(v_x_hat) + epsilon)
        y -= alpha * m_y_hat / (sympy.sqrt(v_y_hat) + epsilon)
        if func.f(x, y) > 1e20:
            break

        path.append((x, y))
        f_vals.append(func.f(x, y))

    return path, f_vals
```

```
    def RMS_grid_search(func, x0, y0, alphas, betas,
       epsilon=1e-4, num_iters=1000):
    best_alpha = None
    best_beta = None
    best_f_value = 1e9

    for alpha in alphas:
        for beta in betas:
            _, f_vals = rmsprop(func, x0, y0, alpha, beta,
               epsilon, num_iters)
            f_value = f_vals[-1]
            if f_value < best_f_value:
                best_f_value = f_value
                best_alpha = alpha
                best_beta = beta

    return best_alpha, best_beta, best_f_value

alphas = np.logspace(-3, 2, 5)
betas = np.linspace(0.25, 0.99, 5)

# Perform the grid search
best_alpha, best_beta, best_f_value = RMS_grid_search(function1,
   x0, y0, alphas, betas)

print(f"Best alpha: {best_alpha:0.4f}")
print(f"Best beta: {best_beta:0.4f}")

alphas = np.linspace(0.5, 2, 5)
betas = np.linspace(0.98, 0.999, 5)
```

```
28
29  # Perform the grid search
30  best_alpha, best_beta, best_f_value = RMS_grid_search(function1,
        x0, y0, alphas, betas)
31
32  print(f"Best alpha: {best_alpha:0.4f}")
33  print(f"Best beta: {best_beta:0.4f}")
34
35  alphas = np.logspace(-3, 2, 5)
36  betas = np.linspace(0.25, 0.95, 5)
37  paths_1 = [[rmsprop(function1, x0, y0, alpha0=alpha,
        beta=beta)[0] for beta in betas] for alpha in alphas]
38  f_vals_1 = [[rmsprop(function1, x0, y0, alpha0=alpha,
        beta=beta)[1] for beta in betas] for alpha in alphas]
39
40  alphas = np.linspace(0.5, 2, 5)
41  betas = np.linspace(0.98, 0.9999, 5)
42  paths_2 = [[rmsprop(function1, x0, y0, alpha0=alpha,
        beta=beta)[0] for beta in betas] for alpha in alphas]
43  f_vals_2 = [[rmsprop(function1, x0, y0, alpha0=alpha,
        beta=beta)[1] for beta in betas] for alpha in alphas]
44
45  x = np.linspace(0, 1000, 1000)
46  alphas = np.logspace(-3, 2, 5)
47  betas = np.linspace(0.25, 0.95, 5)
48
49  plt.figure(figsize=(24, 14))
50  plt.suptitle("Function 1: alpha vs beta", fontsize=24,
        fontweight='bold')
51
52  for i in range(5):
53      plt.subplot(2, 5, i+1)
54      plt.semilogy(x, f_vals_1[0][i],
            label=f'alpha={alphas[0]:0.4f}')
55      plt.semilogy(x, f_vals_1[1][i],
            label=f'alpha={alphas[1]:0.4f}')
56      plt.semilogy(x, f_vals_1[2][i],
            label=f'alpha={alphas[2]:0.4f}')
57      plt.title(f'Beta = {betas[i]:0.4f}')
58      plt.legend()
59      plt.grid(visible=True)
```

```
alphas = np.linspace(0.5, 2, 5)
betas = np.linspace(0.98, 0.9999, 5)
for i in range(5):
    plt.subplot(2, 5, i+6)
    plt.semilogy(x, f_vals_2[0][i],
       label=f'alpha={alphas[0]:0.4f}')
    plt.semilogy(x, f_vals_2[1][i],
       label=f'alpha={alphas[1]:0.4f}')
    plt.semilogy(x, f_vals_2[2][i],
       label=f'alpha={alphas[2]:0.4f}')
    plt.title(f'Beta = {betas[i]:0.4f}')
    plt.legend()
    plt.grid(visible=True)

plt.show()
```

```
alphas = np.logspace(-3, 1, 5)
betas = np.linspace(0.25, 0.95, 5)

# Perform the grid search
best_alpha, best_beta, best_f_value = RMS_grid_search(function2,
   x0, y0, alphas, betas)

print(f"Best alpha: {best_alpha:0.4f}")
print(f"Best beta: {best_beta:0.4f}")

alphas = np.linspace(0.5, 1.5, 5)
betas = np.linspace(0.85, 0.92, 5)

# Perform the grid search
best_alpha, best_beta, best_f_value = RMS_grid_search(function2,
   x0, y0, alphas, betas)

print(f"Best alpha: {best_alpha:0.4f}")
print(f"Best beta: {best_beta:0.4f}")

alphas = np.logspace(-3, 1,3)
betas = np.linspace(0.25, 0.95, 5)
paths_1 = [[rmsprop(function1, x0, y0, alpha0=alpha,
   beta=beta)[0] for beta in betas] for alpha in alphas]
```

```
f_vals_1 = [[rmsprop(function1, x0, y0, alpha0=alpha,
    beta=beta)[1] for beta in betas] for alpha in alphas]

alphas = np.linspace(0.5, 1.5, 3)
betas = np.linspace(0.85, 0.92, 5)
paths_2 = [[rmsprop(function1, x0, y0, alpha0=alpha,
    beta=beta)[0] for beta in betas] for alpha in alphas]
f_vals_2 = [[rmsprop(function1, x0, y0, alpha0=alpha,
    beta=beta)[1] for beta in betas] for alpha in alphas]

x = np.linspace(0, 1000, 1000)
alphas = np.logspace(-3, 1, 3)
betas = np.linspace(0.25, 0.95, 5)
plt.figure(figsize=(24, 14))
plt.suptitle("Function 2: alpha vs beta", fontsize=24,
    fontweight='bold')

for i in range(5):
    plt.subplot(2, 5, i+1)
    plt.semilogy(x, f_vals_1[0][i],
        label=f'alpha={alphas[0]:0.4f}')
    plt.semilogy(x, f_vals_1[1][i],
        label=f'alpha={alphas[1]:0.4f}')
    plt.semilogy(x, f_vals_1[2][i],
        label=f'alpha={alphas[2]:0.4f}')
    plt.title(f'Beta = {betas[i]:0.4f}')
    plt.legend()
    plt.grid(visible=True)


alphas = np.linspace(0.5, 1.5, 3)
betas = np.linspace(0.85, 0.92, 5)
for i in range(5):
    plt.subplot(2, 5, i+6)
    plt.semilogy(x, f_vals_2[0][i],
        label=f'alpha={alphas[0]:0.4f}')
    plt.semilogy(x, f_vals_2[1][i],
        label=f'alpha={alphas[1]:0.4f}')
    plt.semilogy(x, f_vals_2[2][i],
        label=f'alpha={alphas[2]:0.4f}')
    plt.title(f'Beta = {betas[i]:0.4f}')
```

```
    plt.legend()
    plt.grid(visible=True)

plt.show()
```

```
def HB_grid_search(func, x0, y0, alphas, betas, tol=1e-4,
   num_iters=1000):
    best_alpha = None
    best_beta = None
    best_f_value = 1e9

    for alpha in alphas:
        for beta in betas:
            _, f_vals = heavy_ball(func, x0, y0, alpha, beta,
               num_iters=num_iters, tol=tol, stopping=False)
            f_value = f_vals[-1]
            if f_value < best_f_value:
                best_f_value = f_value
                best_alpha = alpha
                best_beta = beta

    return best_alpha, best_beta, best_f_value

alphas = np.logspace(-5, -2, 6)
betas = np.linspace(0.25, 0.95, 6)

# Perform the grid search
best_alpha, best_beta, best_f_value = HB_grid_search(function1,
   x0, y0, alphas, betas)

print(f"Best alpha: {best_alpha:0.4f}")
print(f"Best beta: {best_beta:0.4f}")

alphas = np.linspace(0.001, 0.02, 6)
betas = np.linspace(0.91, 0.99, 6)

# Perform the grid search
best_alpha, best_beta, best_f_value = HB_grid_search(function1,
   x0, y0, alphas, betas)

print(f"Best alpha: {best_alpha:0.4f}")
print(f"Best beta: {best_beta:0.4f}")
```

```
34
35 alphas = np.logspace(-5, -2, 3)
36 betas = np.linspace(0.25, 0.95, 6)
37 paths_1 = [[heavy_ball(function1, x0, y0, alpha=alpha,
      beta=beta, stopping=True)[0] for beta in betas] for alpha in
      alphas]
38 f_vals_1 = [[heavy_ball(function1, x0, y0, alpha=alpha,
      beta=beta, stopping=True)[1] for beta in betas] for alpha in
      alphas]
39
40 alphas = np.linspace(0.001, 0.01, 3)
41 betas = np.linspace(0.91, 0.99, 6)
42 paths_2 = [[heavy_ball(function1, x0, y0, alpha=alpha,
      beta=beta, stopping=True)[0] for beta in betas] for alpha in
      alphas]
43 f_vals_2 = [[heavy_ball(function1, x0, y0, alpha=alpha,
      beta=beta, stopping=True)[1] for beta in betas] for alpha in
      alphas]
44
45 alphas = np.logspace(-5, -2, 3)
46 betas = np.linspace(0.25, 0.95, 6)
47
48 plt.figure(figsize=(24, 28), dpi=300)
49 plt.suptitle("Function 1: alpha vs beta", fontsize=24,
      fontweight='bold')
50
51 for i in range(6):
52     plt.subplot(4, 3, i+1)
53     plt.semilogy(np.linspace(0, len(f_vals_1[0][i]),
          len(f_vals_1[0][i])), f_vals_1[0][i],
          label=f'alpha={alphas[0]:0.4f}')
54     plt.semilogy(np.linspace(0, len(f_vals_1[1][i]),
          len(f_vals_1[1][i])), f_vals_1[1][i],
          label=f'alpha={alphas[1]:0.4f}')
55     plt.semilogy(np.linspace(0, len(f_vals_1[2][i]),
          len(f_vals_1[2][i])), f_vals_1[2][i],
          label=f'alpha={alphas[2]:0.4f}')
56     plt.title(f'Beta = {betas[i]:0.4f}')
57     plt.legend()
58     plt.grid(visible=True)
59
```

```
60
61
62 alphas = np.linspace(0.005, 0.05, 3)
63 betas = np.linspace(0.89, 0.93, 6)
64 for i in range(6):
65     plt.subplot(4, 3, i+7)
66     plt.semilogy(np.linspace(0, len(f_vals_2[0][i]),
          len(f_vals_2[0][i])), f_vals_2[0][i],
          label=f'alpha={alphas[0]:0.4f}')
67     plt.semilogy(np.linspace(0, len(f_vals_2[1][i]),
          len(f_vals_2[1][i])), f_vals_2[1][i],
          label=f'alpha={alphas[1]:0.4f}')
68     plt.semilogy(np.linspace(0, len(f_vals_2[2][i]),
          len(f_vals_2[2][i])), f_vals_2[2][i],
          label=f'alpha={alphas[2]:0.4f}')
69     plt.title(f'Beta = {betas[i]:0.4f}')
70     plt.legend()
71     plt.grid(visible=True)
72
73 plt.show()
```

```
1     alphas = np.logspace(-5, -2, 6)
2 betas = np.linspace(0.25, 0.95, 6)
3
4 # Perform the grid search
5 best_alpha, best_beta, best_f_value = HB_grid_search(function2,
     x0, y0, alphas, betas)
6
7 print(f"Best alpha: {best_alpha:0.4f}")
8 print(f"Best beta: {best_beta:0.4f}")
9
10 alphas = np.linspace(0.0001, 0.001, 6)
11 betas = np.linspace(0.82, 0.86, 6)
12
13 # Perform the grid search
14 best_alpha, best_beta, best_f_value = HB_grid_search(function2,
     x0, y0, alphas, betas)
15
16 print(f"Best alpha: {best_alpha:0.4f}")
17 print(f"Best beta: {best_beta:0.4f}")
18
19 alphas = np.logspace(-5, -2, 3)
```

```
betas = np.linspace(0.25, 0.95, 5)
paths_1 = [[heavy_ball(function2, x0, y0, alpha=alpha,
    beta=beta, stopping=False)[0] for beta in betas] for alpha in
    alphas]
f_vals_1 = [[heavy_ball(function2, x0, y0, alpha=alpha,
    beta=beta, stopping=False)[1] for beta in betas] for alpha in
    alphas]

alphas = np.linspace(0.0001, 0.001, 3)
betas = np.linspace(0.82, 0.86, 5)
paths_2 = [[heavy_ball(function2, x0, y0, alpha=alpha,
    beta=beta, stopping=False)[0] for beta in betas] for alpha in
    alphas]
f_vals_2 = [[heavy_ball(function2, x0, y0, alpha=alpha,
    beta=beta, stopping=False)[1] for beta in betas] for alpha in
    alphas]

alphas = np.logspace(-5, -2, 3)
betas = np.linspace(0.25, 0.95, 5)

plt.figure(figsize=(24, 14), dpi=300)
plt.suptitle("Function 2: alpha vs beta", fontsize=24,
    fontweight='bold')

for i in range(5):
    plt.subplot(2, 5, i+1)
    plt.semilogy(np.linspace(0, len(f_vals_1[0][i]),
        len(f_vals_1[0][i])), f_vals_1[0][i],
        label=f'alpha={alphas[0]:0.4f}')
    plt.semilogy(np.linspace(0, len(f_vals_1[1][i]),
        len(f_vals_1[1][i])), f_vals_1[1][i],
        label=f'alpha={alphas[1]:0.4f}')
    plt.semilogy(np.linspace(0, len(f_vals_1[2][i]),
        len(f_vals_1[2][i])), f_vals_1[2][i],
        label=f'alpha={alphas[2]:0.4f}')
    plt.title(f'Beta = {betas[i]:0.4f}')
    plt.legend()
    plt.grid(visible=True)


alphas = np.linspace(0.0001, 0.001, 3)
```

```
betas = np.linspace(0.82, 0.86, 5)
for i in range(5):
    plt.subplot(2, 5, i+6)
    plt.semilogy(np.linspace(0, len(f_vals_2[0][i]),
       len(f_vals_2[0][i])), f_vals_2[0][i],
       label=f'alpha={alphas[0]:0.4f}')
    plt.semilogy(np.linspace(0, len(f_vals_2[1][i]),
       len(f_vals_2[1][i])), f_vals_2[1][i],
       label=f'alpha={alphas[1]:0.4f}')
    plt.semilogy(np.linspace(0, len(f_vals_2[2][i]),
       len(f_vals_2[2][i])), f_vals_2[2][i],
       label=f'alpha={alphas[2]:0.4f}')
    plt.title(f'Beta = {betas[i]:0.4f}')
    plt.legend()
    plt.grid(visible=True)

plt.show()
```

```
    def adam_grid_search(func, x0, y0, alphas, beta1s, beta2s,
       tol=1e-4, num_iters=1000):
    best_alpha = None
    best_beta1 = None
    best_best2 = None
    best_f_value = 1e9

    for alpha in alphas:
        for beta1 in beta1s:
            for beta2 in beta2s:
                _, f_vals =  adam(func, x0, y0, alpha, beta1,
                   beta2, epsilon=1e-6, num_iters=1000)
                f_value = f_vals[-1]
                if f_value < best_f_value:
                    best_f_value = f_value
                    best_alpha = alpha
                    best_beta1 = beta1
                    best_beta2 = beta2


    return best_alpha, best_beta1, best_beta2, best_f_value

alphas = np.logspace(-5, -2, 4)
beta1s = np.linspace(0.25, 0.9999, 4)
```

```
beta2s = np.linspace(0.25, 0.9999, 4)

# Perform the grid search
best_alpha, best_beta1, best_beta2, best_f_value =
    adam_grid_search(function1, x0, y0, alphas, beta1s=beta1s,
    beta2s=beta2s)

print(f"Best alpha: {best_alpha:0.4f}")
print(f"Best beta1: {best_beta1:0.4f}")
print(f"Best beta2: {best_beta2:0.4f}")

alphas = np.logspace(-5, -2, 4)
beta1s = np.linspace(0.8, 92, 4)
beta2s = np.linspace(0.58, 0.68, 4)

# Perform the grid search
best_alpha, best_beta1, best_beta2, best_f_value =
    adam_grid_search(function1, x0, y0, alphas, beta1s=beta1s,
    beta2s=beta2s)

print(f"Best alpha: {best_alpha:0.4f}")
print(f"Best beta1: {best_beta1:0.4f}")
print(f"Best beta2: {best_beta2:0.4f}")

alphas = np.linspace(0.1, 0.5, 3)
beta1s = np.linspace(0.8, 0.99, 4)
beta2s = np.linspace(0.98, 0.99999, 4)
paths_1 = [[[adam(function1, x0, y0, alpha, beta1, beta2,
    epsilon=1e-6, num_iters=1000)[0] for beta1 in beta1s] for
    beta2 in beta2s] for alpha in alphas]
f_vals_1 = [[[adam(function1, x0, y0, alpha, beta1, beta2,
    epsilon=1e-6, num_iters=1000)[1] for beta1 in beta1s] for
    beta2 in beta2s] for alpha in alphas]


plt.figure(figsize=(24, 14), dpi=300)
plt.suptitle("Function 1: alpha vs beta1 vs beta2", fontsize=24,
    fontweight='bold')

for i in range(4):
    plt.subplot(3, 4, i+1)
```

```
55      plt.semilogy(np.linspace(0, len(f_vals_1[0][0][i]),
            len(f_vals_1[0][0][i])), f_vals_1[0][0][i],
            label=f'alpha={alphas[0]:0.4f}')
56      plt.semilogy(np.linspace(0, len(f_vals_1[0][1][i]),
            len(f_vals_1[0][1][i])), f_vals_1[0][1][i],
            label=f'alpha={alphas[1]:0.4f}')
57      plt.semilogy(np.linspace(0, len(f_vals_1[0][2][i]),
            len(f_vals_1[0][2][i])), f_vals_1[0][2][i],
            label=f'alpha={alphas[2]:0.4f}')
58      plt.title(f'Beta1 = {beta2s[0]:0.4f}, beta2 =
            {beta2s[i]:0.4f}')
59      plt.legend()
60      plt.grid(visible=True)
61
62  for i in range(4):
63      plt.subplot(3, 4, i+5)
64      plt.semilogy(np.linspace(0, len(f_vals_1[1][0][i]),
            len(f_vals_1[1][0][i])), f_vals_1[1][0][i],
            label=f'alpha={alphas[0]:0.4f}')
65      plt.semilogy(np.linspace(0, len(f_vals_1[1][1][i]),
            len(f_vals_1[1][1][i])), f_vals_1[1][1][i],
            label=f'alpha={alphas[1]:0.4f}')
66      plt.semilogy(np.linspace(0, len(f_vals_1[1][2][i]),
            len(f_vals_1[1][2][i])), f_vals_1[1][2][i],
            label=f'alpha={alphas[2]:0.4f}')
67      plt.title(f'Beta1 = {beta2s[1]:0.4f}, beta2 =
            {beta2s[i]:0.4f}')
68      plt.legend()
69      plt.grid(visible=True)
70
71  for i in range(4):
72      plt.subplot(3, 4, i+9)
73      plt.semilogy(np.linspace(0, len(f_vals_1[2][0][i]),
            len(f_vals_1[2][0][i])), f_vals_1[2][0][i],
            label=f'alpha={alphas[0]:0.4f}')
74      plt.semilogy(np.linspace(0, len(f_vals_1[2][1][i]),
            len(f_vals_1[2][1][i])), f_vals_1[2][1][i],
            label=f'alpha={alphas[1]:0.4f}')
75      plt.semilogy(np.linspace(0, len(f_vals_1[2][2][i]),
            len(f_vals_1[2][2][i])), f_vals_1[2][2][i],
            label=f'alpha={alphas[2]:0.4f}')
```

```
    plt.title(f'Beta1 = {beta2s[2]:0.4f}, beta2 = {beta2s[i]:0.4f}')
    plt.legend()
    plt.grid(visible=True)

plt.show()
```

```
    class ReLU:
    def f(self, x):
        return max(0, x)
    def dx(self, x):
        return 1 if x > 0 else 0

def polyak_GD_relu(func, x0, f_star=0, epsilon=0.0001, max_iters=50, tol=100):
    # Initialize x,y and path and f(x,y) lists
    x = x0
    path = [x]
    f_vals = []

    # Implement formula to find alpha and update at each step.
    for i in range(max_iters):
        df_dx = func.dx(x)
        alpha = (func.f(x) - f_star) / (df_dx**2 + epsilon)
        # Update x and y based on polyak step - optimal alpha
        x -= alpha * df_dx
        # Append both (x, y) point and f value to lists
        path.append(x)
        f_vals.append(func.f(x))

    print(f"No sufficient convergence by iteration {max_iters}\nFinished at x: ({x:0.4f}, f(x) = {f_vals[-1]:0.4f})")
    return path, f_vals

def rmsprop_relu(func, x0, alpha0, beta, epsilon=0.0001, max_iters=50):
    # Initialize x,y, sums and path,f(x,y) lists
    x = x0
    sum_dx = 0
    path = [x]
    f_vals = []
```

```
32
33      for t in range(1, max_iters + 1):
34          # Take gradients for current x, y
35          grad_x = func.dx(x)
36
37          # Update denominator for x and y using the following
38          sum_dx = beta * sum_dx + (1 - beta) * grad_x**2
39
40          # Compute the step sizes for x and y
41          step_size_x = alpha0 / sympy.sqrt(sum_dx + epsilon)
42
43          # Update x and y
44          x -= step_size_x * grad_x
45
46          path.append(x)
47          f_vals.append(func.f(x))
48
49      return path, f_vals
50
51  def heavy_ball_relu(func, x0, alpha, beta, num_iters=50,
        tol=1e-5, stopping=True):
52      # Initialize x,y, momentum variables and path,f(x,y) lists
53      x = x0
54      ux = 0
55      path = [x]
56      f_vals = []
57
58      for i in range(num_iters):
59          # Take gradients for current x, y
60          grad_x = func.dx(x)
61
62          # Update momentum variable with current and past momentum
63          ux = beta * ux + alpha * grad_x
64
65          # Update x and y
66          x -= ux
67
68          path.append(x)  # Store the new values
69          f_vals.append(func.f(x))
70
71      print(f"Finished at x: ({x:0.4f}, f(x) = {f_vals[-1]:0.4f})")
```

```
    return path, f_vals

def adam_relu(func, x0, alpha, beta1, beta2, epsilon=1e-6,
    num_iters=35):
    # Initialize x,y, momentum variables and path,f(x,y) lists
    x = x0
    m_x = 0
    v_x = 0
    path = [x]
    f_vals = []

    for t in range(1, num_iters + 1):
        grad_x = func.dx(x)

        m_x = beta1 * m_x + (1 - beta1) * grad_x
        v_x = beta2 * v_x + (1 - beta2) * grad_x**2

        m_x_hat = m_x / (1 - beta1**t)
        v_x_hat = v_x / (1 - beta2**t)

        x -= alpha * m_x_hat / (sympy.sqrt(v_x_hat) + epsilon)

        path.append(x)
        f_vals.append(func.f(x))

    return path, f_vals

relu_func = ReLU()
x0 = 1
path, f_vals = polyak_GD_relu(relu_func, x0, max_iters=30)
f_vals
path, f_vals = rmsprop_relu(relu_func, x0, 0.02, 0.9,
    max_iters=30)
f_vals
path, f_vals = heavy_ball_relu(relu_func, x0, 0.02, 0.9,
    num_iters=30)
f_vals
path, f_vals = adam_relu(relu_func, x0, 0.02, 0.9, 0.999,
    num_iters=30)
path

```

```
plt.figure(figsize=(15, 8), dpi=300)
plt.suptitle("ReLU Function Comparison", fontsize=18,
    fontweight='bold')

x0 = 1
plt.subplot(1,2,1)
plt.semilogy(np.linspace(0, len(polyak_GD_relu(relu_func,
    x0)[0]), len(polyak_GD_relu(relu_func, x0)[0])),
    polyak_GD_relu(relu_func, x0)[0], label='Polyak')
plt.semilogy(np.linspace(0, len(rmsprop_relu(relu_func, x0,
    0.02, 0.9)[0]), len(rmsprop_relu(relu_func, x0, 0.02,
    0.9)[0])), rmsprop_relu(relu_func, x0, 0.02, 0.9)[0],
    label='RMSProp')
plt.semilogy(np.linspace(0, len(heavy_ball_relu(relu_func, x0,
    0.02, 0.9)[0]), len(heavy_ball_relu(relu_func, x0, 0.02,
    0.9)[0])), heavy_ball_relu(relu_func, x0, 0.02, 0.9)[0],
    label='Heavy Ball')
plt.semilogy(np.linspace(0, len(adam_relu(relu_func, x0, 0.02,
    0.9, 0.999)[0]), len(adam_relu(relu_func, x0, 0.02, 0.9,
    0.999)[0])), adam_relu(relu_func, x0, 0.02, 0.9, 0.999)[0],
    label='Adam')
plt.legend()
plt.grid(visible=True)
plt.xlabel('Iterations')
plt.ylabel('Loss (log scale)')

x0 = 100

plt.subplot(1,2, 2)
plt.semilogy(np.linspace(0, len(polyak_GD_relu(relu_func, x0,
    max_iters=1000)[0]), len(polyak_GD_relu(relu_func, x0,
    max_iters=1000)[0])), polyak_GD_relu(relu_func, x0,
    max_iters=1000)[0], label='Polyak')
plt.semilogy(np.linspace(0, len(rmsprop_relu(relu_func, x0,
    0.02, 0.9,max_iters=1000)[0]), len(rmsprop_relu(relu_func,
    x0, 0.02, 0.9,max_iters=1000)[0])), rmsprop_relu(relu_func,
    x0, 0.02, 0.9,max_iters=1000)[0], label='RMSProp')
plt.semilogy(np.linspace(0, len(heavy_ball_relu(relu_func, x0,
    0.02, 0.9, num_iters=1000)[0]),
    len(heavy_ball_relu(relu_func, x0, 0.02, 0.9,
    num_iters=1000)[0])), heavy_ball_relu(relu_func, x0, 0.02,
```

```
    0.9, num_iters=1000)[0], label='Heavy Ball')
plt.semilogy(np.linspace(0, len(adam_relu(relu_func, x0, 0.02,
    0.9, 0.999, num_iters=1000)[0]), len(adam_relu(relu_func, x0,
    0.02, 0.9, 0.999, num_iters=1000)[0])), adam_relu(relu_func,
    x0, 0.02, 0.9, 0.999, num_iters=1000)[0], label='Adam')
plt.legend()
plt.grid(visible=True)
plt.xlabel('Iterations')
plt.ylabel('Loss (log scale)')
plt.show()
```