Functions

The functions I will use for this assignment are:

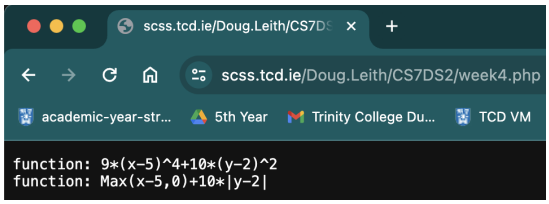

Figure 1: Functions obtained

Function	Expression
$f_1(x, y)$	$9(x-5)^4 + 10(y-2)^2$
$f_2(x, y)$	$max(x-5, 0) + 10 \cdot \|y-2\|$

Table 1: Functions as expressions

By declaring x and y as symbols in Python with SymPy (using the **symbols()** function), we can declare the functions (for f_2 we can use the **Max()** and **Abs()** functions) and differentiate them (using the **diff()** function) in terms of x and y:

Differentiating the 1st function in terms of x: $\frac{\partial f_1}{\partial x} = 36(x-5)^3$

Differentiating the 1st function in terms of y: $\frac{\partial f_1}{\partial y} = 20y - 40$

Differentiating the 2nd function in terms of x: $\frac{\partial f_2}{\partial x} = \begin{cases} 0 & \text{if } x < 5 \\ 1 & \text{if } x > 5 \end{cases} = \theta(x-5)$

Differentiating the 2nd function in terms of y: $\frac{\partial f_2}{\partial y} == \begin{cases} -10 & \text{if } y < 2 \\ +10 & \text{if } y > 2 \end{cases} = 10 \cdot sign(y-2)$

Question (a)

I implemented three gradient descent updates which are discussed below, where:

- `number_of_parameters` is the number of parameters i.e. 2 parameters for $f_1(x, y)$.
- `array_of_lambda_functions` are lamda functions in an array to compute the partial derivatives of the primary function i.e. $[\frac{\partial f_1}{\partial x}, \frac{\partial f_1}{\partial y}]$.
- `x` are parameters in an array i.e $[x, y]$.

Part (i)

I implemented the Polyak step size gradient descent update by firstly subtracting f^* from the value of the function for the parameter values, then dividing this by the the partial derivatives which are the sum of the squares of the gradients for each parameter. It is important to note that for this assignment, I set f^* as 0 so we do not subtract anything before we divide. This step size is computed at each iteration, where ϵ is added to the denominator to stop dividing by 0. Subsequently, we subtract the product of the computed step size and the parameter's gradient from each parameter as shown below:

```
# Calculate the partial derivatives (sum of the squares of the gradients)
sum_of_squares_of_gradients = 0
for iteration in range(number_of_parameters):
    sum_of_squares_of_gradients +=
    array_of_lambda_functions[iteration](x[iteration]) ** 2
# Calculate the step size
step_size = function(*x) / (sum_of_squares_of_gradients + epsilon)
# Recalculate values
```

```
for iteration in range(number_of_parameters):
    x[iteration] -= (step_size *
    array_of_lambda_functions[iteration](x[iteration]))
```

Part (ii)

I implemented the RMSProp gradient descent update by firstly declaring the sums of historical square gradients and step sizes for each parameter as empty arrays. Before we iterate, step sizes are set to α_0. When we iterate, we subtract the product of the initial step size and the parameter's gradient from the parameter values. With β weighting current and historical gradients, the sum of historical square gradients has the current square of the parameter's gradient added to it. Step sizes are subsequently calculated with α_0 divided by the current square gradient sum's square root, with ϵ added to the denominator to prevent division by 0 as shown below:

```
sums = number_of_parameters * [0]
alphas = number_of_parameters * [a_0]
for _ in range(iterations):
    # Recalculate values
    for iteration in range(number_of_parameters):
        # Compute steps
        x[iteration] -= alphas[iteration] *
        array_of_lambda_functions[iteration](x[iteration])
        sums[iteration] = (b * sums[iteration]) + ((1 - b) *
        (array_of_lambda_functions[iteration](x[iteration]) ** 2))
        alphas[iteration] = a_0 / ((math.sqrt(sums[iteration])) + epsilon)
```

Part (iii)

I implemented the Heavy Ball gradient descent update by firstly setting the historical sum of square gradients to zero. When iterating, this value is recalculated by adding the current square of the parameter's gradient with β weighting the current and historical gradients. This time, the sum of square gradients includes every parameter, and only one step size is used instead of a step size for each parameter. The value of the function is divided by this sum for the parameter values (we need to subtract $f*$ but we don't need to do it as we set it as 0 in this assignment). The product of the updated step size and its gradient is subtracted from every parameter as shown:

```
historical_sum_of_square_gradients = 0
for _ in range(iterations):
    # Calculate sum of the squares of the gradients (partial derivatives)
    sum_of_squares_of_gradients = 0
    for iteration in range(number_of_parameters):
        sum_of_squares_of_gradients +=
        array_of_lambda_functions[iteration](x[iteration]) ** 2
    # Calculate the step size
    historical_sum_of_square_gradients = (b *
    historical_sum_of_square_gradients) + (a * function(*x) /
    (sum_of_squares_of_gradients + epsilon))
    # Recalculate values
    for iteration in range(number_of_parameters):
        x[iteration] -= historical_sum_of_square_gradients *
        array_of_lambda_functions[iteration](x[iteration])
```

Part (iv)

I implemented the Adam gradient descent update by firstly declaring the weighted historical sums of gradients and square gradients for each parameter as empty arrays. Then, we keep track of the iteration count and set it to 0 initially. During each iteration, we increase this counter and the historical sums of gradients and square gradients are updated by adding the current square of the parameter's gradient to the sum of historical square gradients (similar to RMSProp), but using β_1 to control the weight of the regular gradients and β_2 to control the weight of the square gradients. We divide the new sums by (1 - $\beta_x^{iteration\_count}$) where $x = 1$ or $x = 2$ depending on the weight, which is to the power of the iteration counter. Next, we subtract the product of the learning rate α and the scaled regular gradient sum divided by the scaled square gradient sum's square root from each parameter, where ϵ prevents division by 0 through its addition as shown:

```
ms = [0] * number_of_parameters
vs = [0] * number_of_parameters
step = [0] * number_of_parameters
iteration_count = 0
for _ in range(iterations):
    # Compute steps
    iteration_count += 1
    # Recalculate values
    for iteration in range(number_of_parameters):
        ms[iteration] = (b1 * ms[iteration]) + ((1 - b1) *
        array_of_lambda_functions[iteration](x[iteration]))
        vs[iteration] = (b2 * vs[iteration]) + ((1 - b2) *
        (array_of_lambda_functions[iteration](x[iteration]) ** 2))
        m_hat = ms[iteration] / (1 - (b1 ** iteration_count))
        v_hat = vs[iteration] / (1 - (b2 ** iteration_count))
        step[iteration] = alpha * (m_hat / ((v_hat ** 0.5) + epsilon))
        x[iteration] -= step[iteration]
```

Question (b)

I applied RMSProps, Heavy Ball and Adam to both f_1 and f_2 to see the effect of changing the parameters for each algorithm. The analysis of each algorithm with their subsequent optimal parameter values are discussed below.

Part (i)

I applied the RMSProps gradient descent update with f_1 and f_2 to examine the effect of changing α and β. For 200 iterations, I used [0.001, 0.01, 0.1] for α_0, [0.25, 0.9] for β, 3 for x_0 and 0 for y_0. The resulting plot for f_1 shows that firstly β does not have much of an an impact at all for these values, but $\beta = 0.9$ converges slightly sooner. Secondly, $\alpha_0 = 0.01$ is the most optimal because it converges at 150 iterations. Values that are too big such as $\alpha_0 = 0.1$ diverged in a way that the results do not even show on the plot (meaning that this failed to perform gradient descent effectively). Values that are too small such as $\alpha_0 = 0.001$ converge too slowly, as it is not close to convergence at 200 iterations.

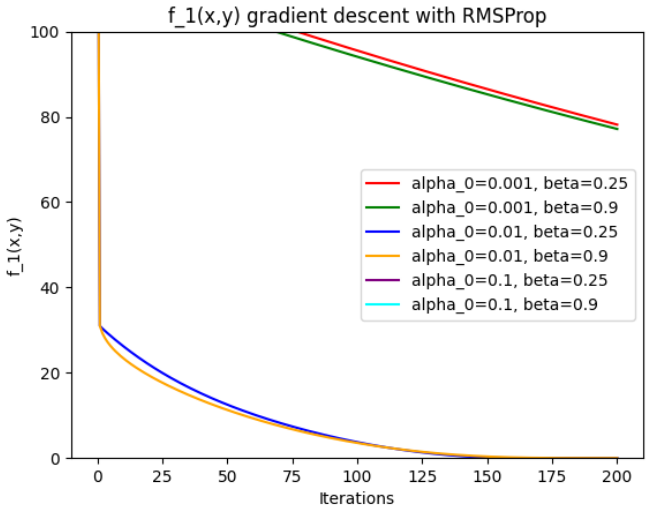

Figure 2: RMSProps with f_1

The resulting plot for f_2 shows that once again β has a limited effect on the algorithm, but $\beta = 0.9$ converges slightly sooner. Secondly, $\alpha_0 = 0.01$ is the most optimal because it converges at around 180 iterations with both values of $\beta = 0.25$ and $\beta = 0.9$ making the algorithm go to 0.01 and 0.08 respectively (near the minimum). Values that are too big such as $\alpha_0 = 0.1$ converge fast to a minimum but oscillate around a point above the minimum on repeat which is not ideal. Values that are too small such as $\alpha_0 = 0.001$ converges too slowly, as it is not close to convergence at 200 iterations. Both f_1 and f_2 converge the most efficiently with $\alpha_0 = 0.01$ and $\beta = 0.9$.

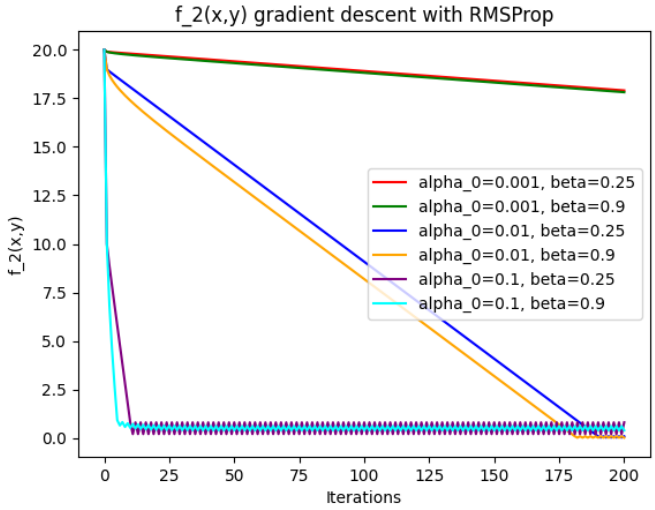

Figure 3: RMSProps with f_2

Part (ii)

I applied the Heavy Ball gradient descent update with f_1 and f_2 to examine the effect of changing α and β. For 200 iterations, I used [0.01, 0.1, 1] for α, [0.25, 0.9] for β, 3 for x_0 and 0 for y_0. The resulting plot for f_1 shows that firstly $\alpha = 1$ converges quickly to a minimum with $\beta = 0.25$ but diverges away from the minimum it reached for $\beta = 0.9$ and goes in the opposite direction. Furthermore, $\alpha = 0.1$ converges on a minimum quick with $\beta = 0.25$ but diverges wildly again thereafter for $\beta = 0.9$ a few times but returns to the

minimum. However, $\alpha = 0.01$ converges too slowly to reach a minimum with $\beta = 0.25$ but $\beta = 0.25$ does converge to a minimum after around 80 iterations.

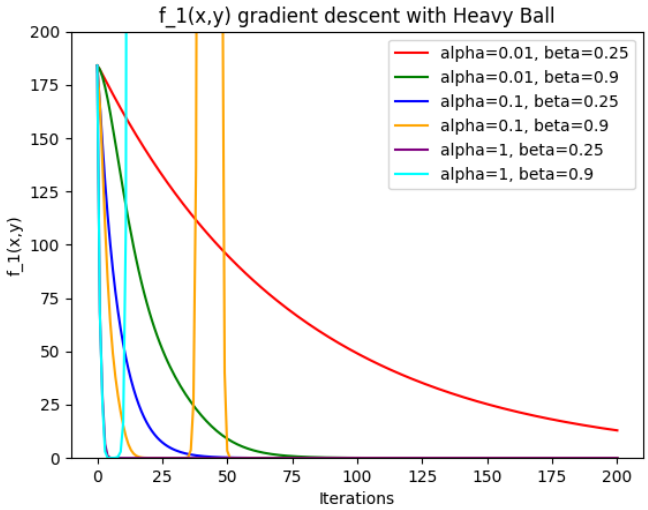

Figure 4: Heavy Ball with f_1

The resulting plot for f_2 shows that firstly $\alpha = 1$ converges quickly to a minimum with $\beta = 0.25$ but diverges away from the minimum it reached for $\beta = 0.9$ and goes the opposite direction. Furthermore, $\alpha = 0.1$ converges on a minimum quick with $\beta = 0.25$ after around 30 iterations but 100 iterations for $\beta = 0.9$ since it oscillates while trying to reach the minimum (longer convergence). However, $\alpha = 0.01$ converges too slowly to reach a minimum with $\beta = 0.25$ but $\beta = 0.25$ does converge on a minimum after around 30 iterations. Both f_1 and f_2 converge most effectively to the minimum with $\alpha = 1$ and $\beta = 0.25$, with a better gradient descent update compared to RMSProp.

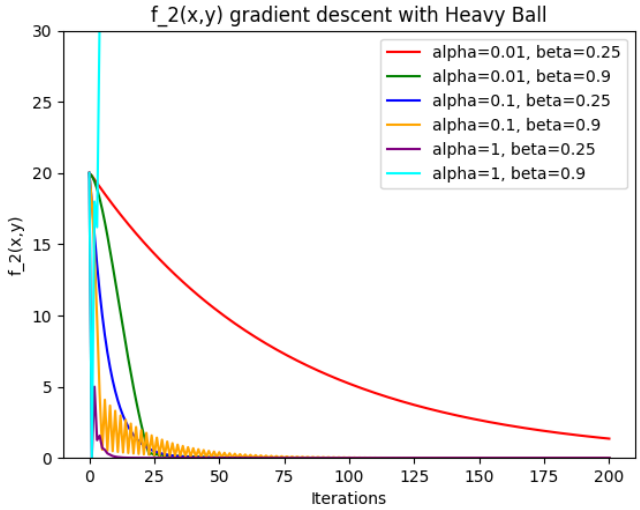

Figure 5: Heavy Ball with f_2

Part (iii)

I applied the Adam gradient descent update with f_1 and f_2 to examine the effect of changing α, β_1 and β_2. For 200 iterations, I used [0.01, 0.1, 1] for α, [0.25, 0.9] for β_1, [0.9, 0.999] for β_2, 3 for x_0 and 0 for y_0. The resulting plots for f_1 show that firstly

using $\alpha = 0.1$ is the most optimal as the functions converge on a minimum at around 60 iterations, with $\beta_1 = 0.9$ and $\beta_2 = 0.9$ being the best combination.

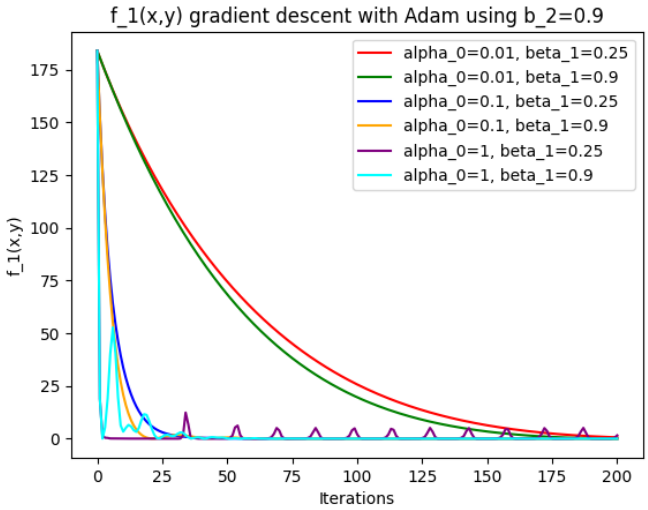

Figure 6: Adam with f_1 and $\beta_2 = 0.9$

Using an α value that is too big such as $\alpha = 1$ results in quick convergence with $\beta_1 = 0.25$. However, using $\beta_2 = 0.9$ causes a repeated divergence every 20 iterations or so which means its results are not stable. Furthermore, using $\beta_2 = 0.999$ causes it to diverge temporarily at around 115 iterations before it goes back to the minimum. Using an α value that is too small such as $\alpha = 0.01$ results in a slow convergence for every combination of β_1 and β_2, with $\beta_2 = 0.9$ converging at around 200 iterations and $\beta_2 = 0.999$ not converging at 200 iterations.

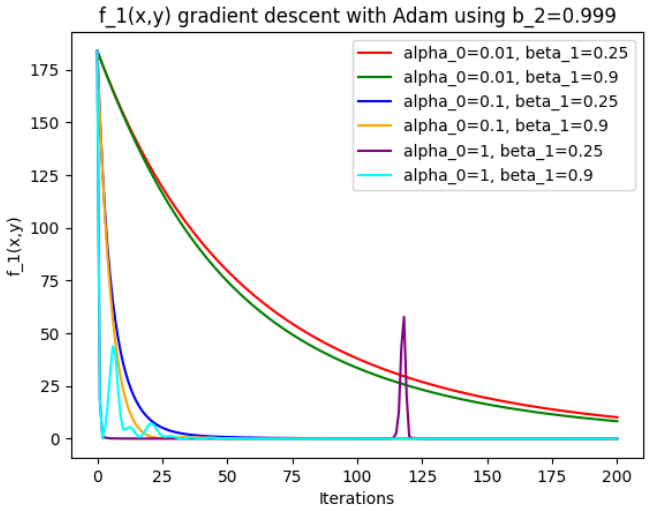

Figure 7: Adam with f_1 and $\beta_2 = 0.999$

These divergences are interestingly caused by the step size of y oscillating without stopping (unlike the step size of x) as shown:

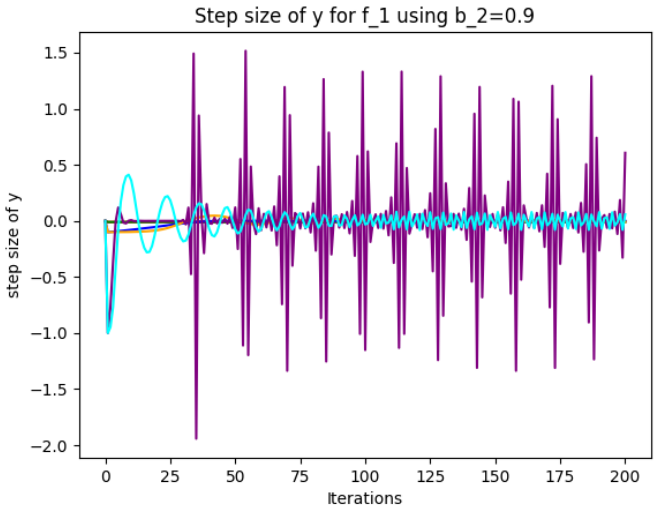

Figure 8: Step size of y for f_1 with $\beta_2 = 0.9$

The resulting plots for f_2 show that firstly using $\beta_2 = 0.9$ and $\beta_2 = 0.999$ has minimal impact on the algorithm. Using $\alpha = 0.1$ is the optimal value for α but using $\beta_1 = 0.25$ causes convergence close to the minimum (to 0.13) but oscillates before it reaches it. Using $\beta_1 = 0.9$ gets closer to the minimum (to 0.1) but gets stuck in a mixed state where it is close to convergence but also oscillates, hence there is no definite convergence.

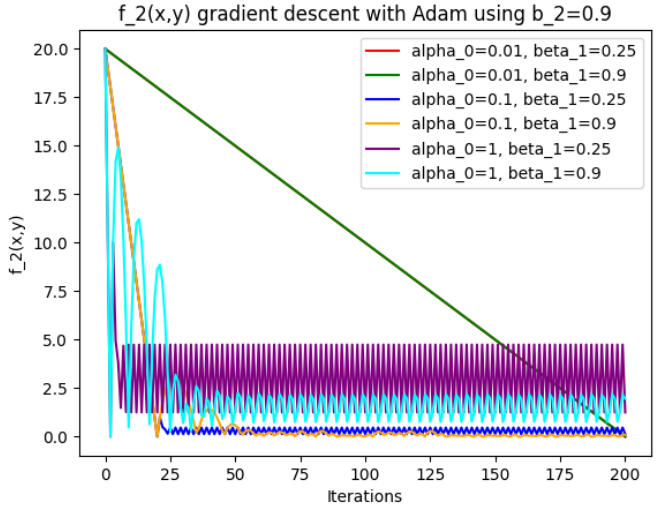

Figure 9: Adam with f_2 and $\beta_2 = 0.9$

Using a small value for α such as $\alpha = 0.01$ causes slow convergence on a minimum at around 200 iterations. Using a large value for α such as $\alpha = 1$ causes quick convergence initially but starts to oscillate around a point on the way to the minimum without stopping.

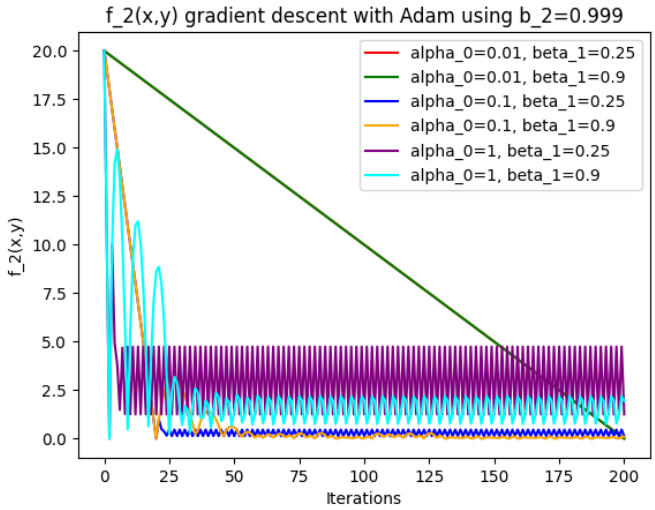

Figure 10: Adam with f_2 and $\beta_2 = 0.999$

These divergences are also causes by the step size of y oscillating as well (unlike the step sizes of x):

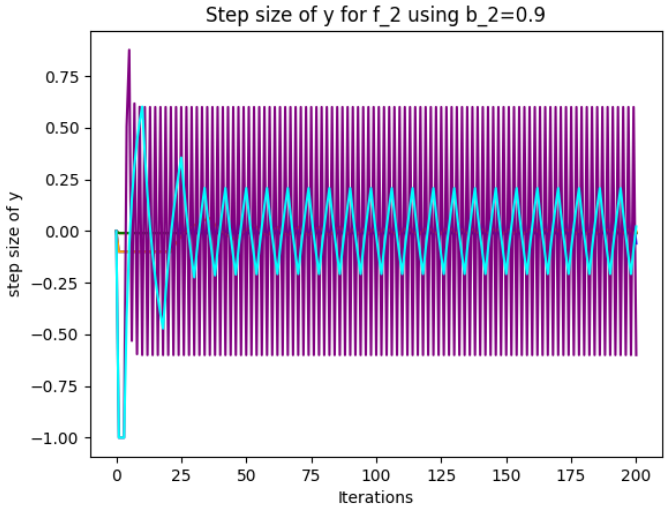

Figure 11: Step size of y for f_2 with $\beta_2 = 0.9$

Overall, f_1 has optimal parameters of $\alpha = 1$, $\beta_1 = 0.25$ and $\beta_2 = 0.999$, however the oscillations and divergences such as at around 115 iterations can cause unstable results. For $f2$, there are no optimal parameters since only $\alpha = 0.01$ converges but this is slow and takes around 200 iterations (not optimal). For f_2, the oscillations are because of the y's gradient having the *sign* function, which flips the sign across iterations repeatedly around the minimum when α (learning rate) is too big.

Question (c)

The most reasonable parameters for $f2$ using RMSProp was $\alpha_0 = 0.01$ and $\beta = 0.9$. Using Heavy Ball, the optimal parameters was $\alpha = 1$ and $\beta = 0.25$. For Adam, it was $\alpha = 0.01$ and $\beta_1 = 0.9$ and $\beta_2 = 0.999$. I used these parameters to run each algorithm for 105 iterations (to add a buffer to ensure they converged if they did).

Part (i)

Using $x_0 = -1$, the algorithms all converge at the minimum of 0 in 1 iteration. This happens because the value of max(-1, 0) is 0 which means it starts at the minimum. Furthermore, all of the algorithms multiply a calculated value using the value of the gradient $\theta(-1)$, but the value of the gradient $\theta(-1)$ is 0, thus the computed step size is 0. Hence, the function is alwavs 0.

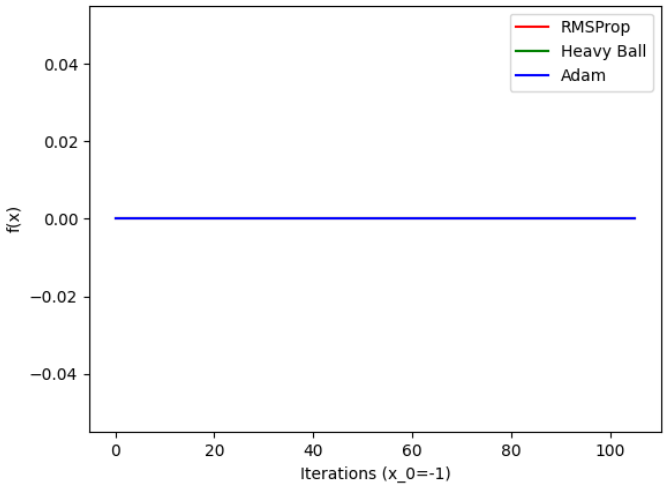

Figure 12: Using $x_0 = -1$ for RMSProp, Heavy Ball and Adam

Part (ii)

Using $x_0 = 1$, all algorithms converge at a minimum, where Heavy Ball is the quickest at 2 iterations, RMSProp is the 2nd quickest at 91 iterations and Adam is the slowest at 101 iterations. Heavy Ball is the quickest because the value of the historical sum of square gradients is $\frac{1}{1+\epsilon}$ at the first iteration which is multiplied by the gradient $\theta(1) = 1$ and subsequently subtracted from x_0. ϵ changes the updated value of x to be a little bit bigger than the minimum, so it requires another iteration to converge (i.e. 2 iterations in total).

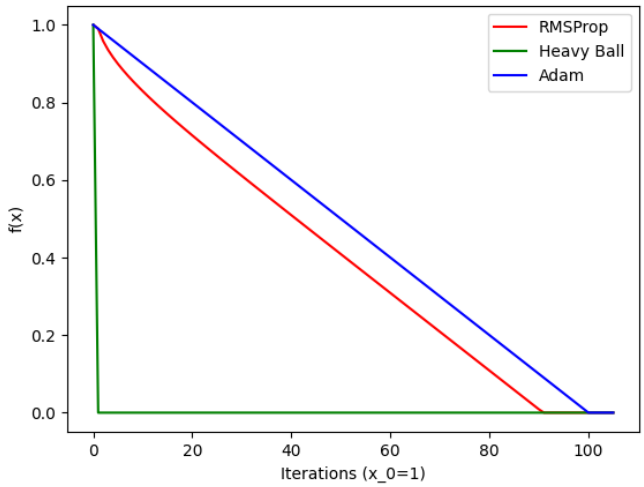

Figure 13: Using $x_0 = 1$ for RMSProp, Heavy Ball and Adam

On the other hand, RMSProp and Adam only decrease using a constant convergence rate since the gradient $\theta(x)$ is constant for $x > 0$, although RMSProp does have a bit faster

of a convergence initially because x is subtracted by a_0 (constant) before the computed a values are used.

Part (iii)

Using $x_0 = 100$, only Heavy Ball converges on a minimum in 2 iterations (like in part (ii)), but RMSProp and Adam do not converge. This is because Heavy Ball subtracts $\frac{100}{1+\epsilon}$ from x which is slightly above the minimum and requires 1 more iteration to converge.

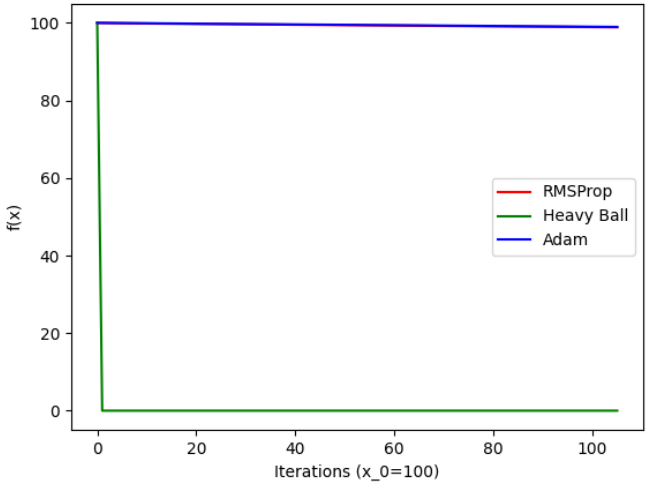

Figure 14: Using $x_0 = 100$ for RMSProp, Heavy Ball and Adam

On the other hand, the change in the initial value of x has no impact on the rate of the function decreasing for RMSProp and Adam since the value of $f(x)$ is never used when we compute the step size. The gradient has an impact on this rate but it is constant for $x > 0$, so they do not converge quickly like in part (ii) and it takes 10,000 iterations for them to converge, making them $\frac{10{,}000}{100} = 100$ times slower.

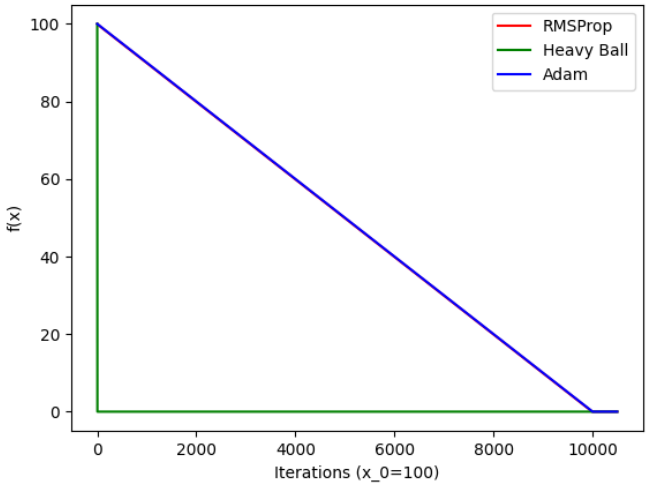

Figure 15: Using $x_0 = 100$ for RMSProp, Heavy Ball and Adam (10,000 iterations)

1 Appendix

1.1 Get function derivatives

```
import math
import sympy as sp
from copy import deepcopy

def get_derivative_of_functions():
    x, y = sp.symbols('x y', real=True)
    first_function = (9 * ((x - 5) ** 4)) + (10 * ((y - 2) ** 2))
    derivative_first_function_in_terms_of_x = sp.diff(first_function, x)
    derivative_first_function_in_terms_of_y = sp.diff(first_function, y)
    second_function = sp.Max(x - 5, 0) + (10 * sp.Abs(y - 2))
    derivative_second_function_in_terms_of_x = sp.diff(second_function, x)
    derivative_second_function_in_terms_of_y = sp.diff(second_function, y)
    return (derivative_first_function_in_terms_of_x,
    derivative_first_function_in_terms_of_y,
            derivative_second_function_in_terms_of_x,
            derivative_second_function_in_terms_of_y)
```

1.2 Question (a) (i)

```
from copy import deepcopy

def polyak(function, array_of_lambda_functions, x0, iterations):
    number_of_parameters = len(array_of_lambda_functions)
    x = deepcopy(x0)
    x_param_values_at_each_iteration = [deepcopy(x)]
    corresponding_values_of_function = [function(*x)]
    steps_at_each_iteration = []
    epsilon = pow(10, -8)
    for _ in range(iterations):
        # Calculate the partial derivatives (sum of the squares of the
        gradients)
        sum_of_squares_of_gradients = 0
        for iteration in range(number_of_parameters):
            sum_of_squares_of_gradients +=
            array_of_lambda_functions[iteration](x[iteration]) ** 2
        # Calculate the step size
        step_size = function(*x) / (sum_of_squares_of_gradients + epsilon)
        # Recalculate values
        for iteration in range(number_of_parameters):
            x[iteration] -= (step_size *
            array_of_lambda_functions[iteration](x[iteration]))
        x_param_values_at_each_iteration.append(deepcopy(x))
        corresponding_values_of_function.append(function(*x))
        steps_at_each_iteration.append(step_size)
    return x_param_values_at_each_iteration,
    corresponding_values_of_function, steps_at_each_iteration
```

1.3 Question (a) (ii)

```
import math
from copy import deepcopy

def rms_prop(function, array_of_lambda_functions, x0, params, iterations):
    number_of_parameters = len(array_of_lambda_functions)
    x = deepcopy(x0)
    x_param_values_at_each_iteration = [deepcopy(x)]
    corresponding_values_of_function = [function(*x)]
    a_0, b = params
    steps_at_each_iteration = [[a_0] * number_of_parameters]
    epsilon = pow(10, -8)
    sums = number_of_parameters * [0]
    alphas = number_of_parameters * [a_0]
    for _ in range(iterations):
        # Recalculate values
        for iteration in range(number_of_parameters):
            # Compute steps
            x[iteration] -= alphas[iteration] *
            array_of_lambda_functions[iteration](x[iteration])
            sums[iteration] = (b * sums[iteration]) + ((1 - b) *
            (array_of_lambda_functions[iteration](x[iteration]) ** 2))
            alphas[iteration] = a_0 / ((math.sqrt(sums[iteration])) +
            epsilon)
        x_param_values_at_each_iteration.append(deepcopy(x))
        corresponding_values_of_function.append(function(*x))
        steps_at_each_iteration.append(deepcopy(alphas))
    return x_param_values_at_each_iteration,
    corresponding_values_of_function, steps_at_each_iteration
```

1.4 Question (a) (iii)

```
from copy import deepcopy

def heavy_ball(function, array_of_lambda_functions, x0, params,
iterations):
    number_of_parameters = len(array_of_lambda_functions)
    x = deepcopy(x0)
    x_param_values_at_each_iteration = [deepcopy(x)]
    corresponding_values_of_function = [function(*x)]
    a, b = params
    steps_at_each_iteration = [0]
    epsilon = pow(10, -8)
    historical_sum_of_square_gradients = 0
    for _ in range(iterations):
        # Calculate sum of the squares of the gradients (partial
        derivatives)
        sum_of_squares_of_gradients = 0
        for iteration in range(number_of_parameters):
            sum_of_squares_of_gradients +=
            array_of_lambda_functions[iteration](x[iteration]) ** 2
        # Calculate the step size
```

```
        historical_sum_of_square_gradients = (b *
        historical_sum_of_square_gradients) + (a * function(*x) /
        (sum_of_squares_of_gradients + epsilon))
        # Recalculate values
        for iteration in range(number_of_parameters):
            x[iteration] -= historical_sum_of_square_gradients *
            array_of_lambda_functions[iteration](x[iteration])
        x_param_values_at_each_iteration.append(deepcopy(x))
        corresponding_values_of_function.append(function(*x))
        steps_at_each_iteration.append(historical_sum_of_square_gradients)
    return x_param_values_at_each_iteration,
    corresponding_values_of_function, steps_at_each_iteration
```

1.5 Question (a) (iv)

```
from copy import deepcopy

def adam(function, array_of_lambda_functions, x0, params, iterations):
    number_of_parameters = len(array_of_lambda_functions)
    x = deepcopy(x0)
    x_param_values_at_each_iteration = [deepcopy(x)]
    corresponding_values_of_function = [function(*x)]
    steps_at_each_iteration = [[0] * number_of_parameters]
    alpha, b1, b2 = params
    epsilon = pow(10, -8)
    ms = [0] * number_of_parameters
    vs = [0] * number_of_parameters
    step = [0] * number_of_parameters
    iteration_count = 0
    for _ in range(iterations):
        # Compute steps
        iteration_count += 1
        # Recalculate values
        for iteration in range(number_of_parameters):
            ms[iteration] = (b1 * ms[iteration]) + ((1 - b1) *
            array_of_lambda_functions[iteration](x[iteration]))
            vs[iteration] = (b2 * vs[iteration]) + ((1 - b2) *
            (array_of_lambda_functions[iteration](x[iteration]) ** 2))
            m_hat = ms[iteration] / (1 - (b1 ** iteration_count))
            v_hat = vs[iteration] / (1 - (b2 ** iteration_count))
            step[iteration] = alpha * (m_hat / ((v_hat ** 0.5) + epsilon))
            x[iteration] -= step[iteration]
        x_param_values_at_each_iteration.append(deepcopy(x))
        corresponding_values_of_function.append(function(*x))
        steps_at_each_iteration.append(deepcopy(step))
    return x_param_values_at_each_iteration,
    corresponding_values_of_function, steps_at_each_iteration
```

1.6 Question (b) (i)

```
from matplotlib import pyplot as plt
```

```
def part_b_i(function, array_of_lambda_functions, x, number_of_function):
    iterations = 200
    list_of_a_0 = [0.001, 0.01, 0.1]
    list_of_b = [0.25, 0.9]
    list_of_iterations = list(range(iterations + 1))
    plot_legend = []
    colors = ['red', 'green', 'blue', 'orange', 'purple', 'cyan']
    index_of_color = 0
    for a_0 in list_of_a_0:
        for b in list_of_b:
            x_param_values_at_each_iteration,
            corresponding_values_of_function, steps_at_each_iteration =
            rms_prop(
                function, array_of_lambda_functions, x, [a_0, b],
                iterations)
            # function values vs iterations
            plt.figure(1)
            plt.plot(list_of_iterations, corresponding_values_of_function,
            color=colors[index_of_color])
            # x step size vs iterations
            step_size_x = [step[0] for step in steps_at_each_iteration]
            plt.figure(2)
            plt.plot(list_of_iterations, step_size_x,
            color=colors[index_of_color])
            # y step size vs iterations
            step_size_y = [step[1] for step in steps_at_each_iteration]
            plt.figure(3)
            plt.plot(list_of_iterations, step_size_y,
            color=colors[index_of_color])
            plot_legend.append(f'alpha_0={a_0}, beta={b}')
            print(f'final value = {corresponding_values_of_function[-1]}
            when alpha_0 = {a_0}, beta = {b}')
            index_of_color += 1
    plt.figure(1)
    # Set y-axis limits to fit as much as possible
    if number_of_function == 1:
        plt.ylim([0, 100])
    plt.ylabel(f"f_{number_of_function}(x,y)")
    plt.xlabel("Iterations")
    plt.title(f"f_{number_of_function}(x,y) gradient descent with RMSProp")
    plt.legend(plot_legend)
    plt.savefig(f"plots/part_b/part_i_f_{number_of_function}")
    plt.figure(2)
    plt.ylabel("step size of x")
    plt.xlabel("Iterations")
    plt.title(f"f_{number_of_function} with step size of x")
    plt.savefig(f"plots/part_b/part_i_f_{number_of_function}_step_x")
    plt.figure(3)
    plt.ylabel("step size of y")
    plt.xlabel("Iterations")
    plt.title(f"Step size of y for f_{number_of_function}")
    plt.savefig(f"plots/part_b/part_i_f_{number_of_function}_step_y")
    plt.show()
```

1.7 Question (b) (ii)

```
from matplotlib import pyplot as plt

def part_b_ii(function, array_of_lambda_functions, x, number_of_function):
    iterations = 200
    list_of_a = [0.01, 0.1, 1]
    list_of_b = [0.25, 0.9]
    list_of_iterations = list(range(iterations + 1))
    plot_legend = []
    colors = ['red', 'green', 'blue', 'orange', 'purple', 'cyan']
    index_of_color = 0
    for a in list_of_a:
        for b in list_of_b:
            x_param_values_at_each_iteration,
            corresponding_values_of_function, steps_at_each_iteration =
            heavy_ball(
                function, array_of_lambda_functions, x, [a, b], iterations)
            plt.plot(list_of_iterations, corresponding_values_of_function,
            color=colors[index_of_color])
            plot_legend.append(f"alpha={a}, beta={b}")
            print(f'final value = {corresponding_values_of_function[-1]}
            when alpha = {a}, beta = {b}')
            index_of_color += 1
    plt.ylim([0, 30])
    # Set y-axis limits to fit as much as possible
    if number_of_function == 1:
        plt.ylim([0, 200])
    plt.ylabel(f"f_{number_of_function}(x,y)")
    plt.xlabel("Iterations")
    plt.title(f"f_{number_of_function}(x,y) gradient descent with Heavy
    Ball")
    plt.legend(plot_legend)
    plt.savefig(f"plots/part_b/part_ii_f_{number_of_function}")
    plt.show()
```

1.8 Question (b) (iii)

```
from matplotlib import pyplot as plt

def part_b_iii(function, array_of_lambda_functions, x, number_of_function):
    iterations = 200
    list_of_a = [0.01, 0.1, 1]
    list_of_b1 = [0.25, 0.9]
    list_of_b2 = [0.9, 0.999]
    list_of_iterations = list(range(iterations + 1))
    plot_legend = []
    colors = ['red', 'green', 'blue', 'orange', 'purple', 'cyan']
    index_of_color = 0
    for b2 in list_of_b2:
        for a in list_of_a:
            for b1 in list_of_b1:
```

```
                x_param_values_at_each_iteration,
                corresponding_values_of_function, steps_at_each_iteration
                = adam(
                    function, array_of_lambda_functions, x, [a, b1, b2],
                    iterations)
                # function values vs iterations
                plt.figure(1)
                plt.plot(list_of_iterations,
                corresponding_values_of_function,
                color=colors[index_of_color])
                # x step size vs iterations
                step_size_x = [step[0] for step in steps_at_each_iteration]
                plt.figure(2)
                plt.plot(list_of_iterations, step_size_x,
                color=colors[index_of_color])
                # y step size vs iterations
                step_size_y = [step[1] for step in steps_at_each_iteration]
                plt.figure(3)
                plt.plot(list_of_iterations, step_size_y,
                color=colors[index_of_color])
                plot_legend.append(f'alpha_0={a}, beta_1={b1}')
                print(f'final value =
                {corresponding_values_of_function[-1]} when alpha = {a},
                beta 1 = {b1}, beta 2 = {b2}')
                index_of_color = (index_of_color + 1) % len(colors)
        plt.figure(1)
        plt.ylabel(f"f_{number_of_function}(x,y)")
        plt.xlabel("Iterations")
        plt.title(f"f_{number_of_function}(x,y) gradient descent with Adam
        using b_2={b2}")
        plt.legend(plot_legend)
        plt.savefig(f"plots/part_b/part_iii_f_{number_of_function}_b_2={b2}
        .png")
        plt.figure(2)
        plt.ylabel("step size of x")
        plt.xlabel("Iterations")
        plt.title(f"f_{number_of_function} with step size of x using
        b_2={b2}")
        plt.savefig(f"plots/part_b/part_iii_f_{number_of_function}_step_x_b
        _2={b2}.png")
        plt.figure(3)
        plt.ylabel("step size of y")
        plt.xlabel("Iterations")
        plt.title(f"Step size of y for f_{number_of_function} using
        b_2={b2}")
        plt.savefig(f"plots/part_b/part_iii_f_{number_of_function}_step_y_b
        _2={b2}.png")
        plt.show()
```

1.9 Question (b) main

```
import numpy as np
from matplotlib import pyplot as plt
```

```
if __name__ == "__main__":
    # Create function 1 and function 2 as lamda functions
    function_1 = lambda x, y: 9 * (x - 5) ** 4 + 10 * (y - 2) ** 2
    function_2 = lambda x, y: 10 * abs(y - 2) + max(0, x - 5)

    # Differentiate function 1 and function 2 in terms of x and y
    print(get_derivative_of_functions())

    # Creating lambda functions from output
    differentiate_function_1_in_terms_of_x = lambda x: 36 * (x - 5) ** 3
    differentiate_function_1_in_terms_of_y = lambda y: 20 * y - 40
    differentiate_function_2_in_terms_of_x = lambda x: np.heaviside(x - 5,
    0)
    differentiate_function_2_in_terms_of_y = lambda y: 10 * np.sign(y - 2)

    part_b_i(function_1, [differentiate_function_1_in_terms_of_x,
    differentiate_function_1_in_terms_of_y], [3, 0], 1)
    part_b_i(function_2, [differentiate_function_2_in_terms_of_x,
    differentiate_function_2_in_terms_of_y], [3, 0], 2)

    part_b_ii(function_1, [differentiate_function_1_in_terms_of_x,
    differentiate_function_1_in_terms_of_y], [3, 0], 1)
    part_b_ii(function_2, [differentiate_function_2_in_terms_of_x,
    differentiate_function_2_in_terms_of_y], [3, 0], 2)

    part_b_iii(function_1, [differentiate_function_1_in_terms_of_x,
    differentiate_function_1_in_terms_of_y], [3, 0], 1)
    part_b_iii(function_2, [differentiate_function_2_in_terms_of_x,
    differentiate_function_2_in_terms_of_y], [3, 0], 2)
```

1.10 Question (c) (i)

```
from matplotlib import pyplot as plt

def part_c(iterations, function, function_derivative):
    list_of_iterations = list(range(iterations + 1))
    colors = ['red', 'green', 'blue', 'orange', 'purple', 'cyan']
    index_of_color = 0
    for x0 in [-1, 1, 100]:
        _, values, _ = rms_prop(function, [function_derivative], [x0],
        [0.01, 0.9], iterations)
        print(f'RMSProp (x0={x0}): {values[-1]}')
        plt.plot(list_of_iterations, values, color=colors[0])
        _, values, _ = heavy_ball(function, [function_derivative], [x0],
        [1, 0.25], iterations)
        print(f'Heavy Ball (x0={x0}): {values[-1]}')
        plt.plot(list_of_iterations, values, color=colors[1])
        _, values, _ = adam(function, [function_derivative], [x0], [0.01,
        0.9, 0.999], iterations)
        print(f'Adam (x0={x0}): {values[-1]}')
        plt.plot(list_of_iterations, values, color=colors[2])
        index_of_color += 1
```

```
        plt.legend(['RMSProp', 'Heavy Ball', 'Adam'])
        plt.ylabel('f(x)')
        plt.xlabel(f'Iterations (x_0={x0})')
        plt.savefig(f"plots/part_c/iterations={iterations}_x_0={x0}.png")
        plt.show()
```

1.11 Question (c) main

```
import numpy as np
from matplotlib import pyplot as plt

if __name__ == "__main__":
    function = lambda x: max(x, 0)
    function_derivative = lambda x: np.heaviside(x, 0)
    # Run for 100 iterations (+5 as buffer)
    part_c(105, function, function_derivative)
    # Run for 10,000 iterations (+500 as buffer)
    part_c(10500, function, function_derivative)
```