Optimisation Algorithms - Week 4 Assignment

Ernests Kuznecovs - 17332791 - kuznecoe@tcd.ie

2nd March

Contents

1 (a) Implementing Optimisation Aglorithms

Numpy is used for the elegent vectorised multiplication, division, addition, substraction.

- e.g in numpys notation: [3 2 1] * [6 5 4] = [(3 * 6) (2 * 5) (1 * 6)]
  - And this sort of element wise operations works the same for
    - (-) (+) (/) (np.sqrt())

1.1 Polyak Step Size

Step size is calcluated with $\alpha = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x) + \epsilon}$

- x is a vector
- $[\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x)] = \nabla f(x)$
- $\nabla f(x)^T \nabla f(x) = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(x)^2$
- f^* is our prediction of the minimum value of f.
- ϵ is mainly to prevent division by zero, but also has the effect of making the algebra work out such that the expression doesn't reduce to a constant value for when $f^* = 0$ aswell.

In the code:

- Each partial derivative is calculated at x and squared, and then summed.
- ϵ is added to the sum and then used as the divisor for $f(x) - f^*$, the resulting number is the step size.
- Each parital at x is multiplied by the step size and the x is updated by taking away the resulting product.

```
for _ in range(iters):
    fdif = f(*x) - f_star
    df_squared_sum = np.sum(np.array([df(*x)**2 for df in dfs]))
    alpha = fdif / (df_squared_sum + eps)
    x = x - alpha * np.array([df(*x) for df in dfs])
```

1.2 RMSProp

For one $\frac{df(x)}{dx}$, $a_t = \frac{a_0}{\sqrt{(1-\beta)\beta^t \frac{df}{dx}(x_0)^2 + (1-\beta)\beta^{t-1}\frac{df}{dx}(x_1)^2 + ... + (1-\beta)\frac{df}{dx}(x_{t-1})^2 + \epsilon}}$, $0 < \beta \leq 1$

- The summing and multiplicaiton of past derivatives values can be implemented by keeping track of the derivatives sums and then simply multiplying the previous iterations sum by β.
  - Since only need to keep track of the sum, as the we dont keep track of previous x's.
- Each partial derivatives gets its own running average.
- We then calculate alpha for each by squaring each of the sums, adding epsilon, and then using it as a divisor for alpha0, which is a hyperparameter that we choose.
- The older derivatives become less and less impactful for the sum (smaller beta, faster forgetful), allowing the step size to increase if reach a region of small gradients for a while.
  - Whereas succesive large gradients will cause the step size to reduce.

```
sum = np.zeros(len(dfs)) ; alpha = alpha0
for _ in range(iters):
    x = x - (alpha * np.array([df(*x) for df in dfs]))
    sum = beta * sum + (1 - beta) * np.array([df(*x)**2 for df in dfs])
    alpha = alpha0 / (np.sqrt(sum) + eps)
```

1.3 Heavy Ball / Polyak Momentum

- Here each partial is affected by its own history of steps just like RMSProp.
- β is used to gradually forget the previous steps, by multiplying the previous step z_{t-1} by $0 < \beta \leq 1$ on each iteration.
- $z_{t-1} * \beta$ is added onto $\alpha * \nabla f(x)$ to construct z_t, where α is our hyperparameter we choose.
- The vector z_t is used as the step updates for our vector x.
- If z ocillates forwards and backwards (keeps taking steps forwards and backwards), the next steps, will be inclined to go towards the middle of the two, since we are summing the negative and positive steps together.
  - Wheras if going in one direction in successions theres many sums in that direction on the tail of z, so even when slope becomes small for current iteration, it still keeps the "momentum".

```
z = np.zeros(len(dfs))
for _ in range(iters):
    z = beta * z + alpha * np.array([df(*x) for df in dfs])
    x = x - z
```

1.4 Adam

Adam $\approx$ RMSprop + heavy ball

- $m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla f(x_t)$ heavy ball bit
  - Although instead of the α that was used, we have the proper weighted running average counterpart, $(1-\beta)$.
- $v_{t+1} = \beta_2 v_t + (1-\beta_2)[\frac{\partial f}{\partial x_1}(x_t)^2, \frac{\partial f}{\partial x_2}(x_t)^2, \ldots, \frac{\partial f}{\partial x_n}(x_t)^2]$ this is rms bit.
- $\hat{m} = \frac{m_{t+1}}{(1-\beta_1^t)}, \hat{v} = \frac{v_{t+1}}{(1-\beta_2^t)}$
- $x_{t+1} = x_t - \alpha[\frac{\hat{m_1}}{\sqrt{\hat{v_1}}+\epsilon}, \frac{\hat{m_2}}{\sqrt{\hat{v_2}}+\epsilon}, \ldots, \frac{\hat{m_n}}{\sqrt{\hat{v_n}}+\epsilon}]$
- m is running average of gradient $\nabla f(x_t)$, giving us information of the direction, for averaging/-momentum.
- v is running average of square gradients, giving us information of the magnitude, for varying size of step.

Thanks to numpy, the implementation looks quite identical to the formula.

- We keep track of iteration number since we need it for mhat and vhat.
- Same concept of keeping the sum part of the previous average, so that we can keep mulitply by β to reduce the weight of the previous steps.
  - Each weighted average has its won hyperparameters β_1, β_2
- The weighted sums are normalised by $\frac{1}{(1-\beta^i)}$
  - eps is used to prevent division by zero
  - alpha scales the resulting step.

```
m = np.zeros(len(dfs)) ; v = np.zeros(len(dfs))
for k in range(iters):
    i = k + 1
    m = beta1 * m + (1 - beta1) * np.array([df(*x) for df in dfs])
    v = beta2 * v + (1 - beta2) * np.array([(df(*x)**2) for df in dfs])
    mhat = (m / (1 - beta1**i))
    vhat = (v / (1 - beta2**i))
    x = x - alpha * (mhat / (np.sqrt(vhat) + eps))
```

2 (b) Inspecting Algorithm Behaviour

2.1 (i) α and β in RMSProp

2.1.1 Function 1

Figs 1 , 2, 5 show plots of function value vs iteration, contour plot and path of algorithm, and step size vs iteration, for function 1.

- An alpha higher than 2 would cause the optimisation algorithm to break on the first few iterations as it shoots off very far due to the very steep function.
- The larger alphas shoot off into the distanace and very slowly begin making their way back to the optimum. The "reasonable" alphas start heading towards the optimium, but at a very slow pace (due to alphas being low).
  - The ones that shoot off far make their way back slowly due to the step size being inverted to the magnitude of the past gradients. The huge initial jumps makes the step succeeding steps tiny.

- For the ones that shoot off, we see that the lower betas allow it to begin converging faster, this is because they forget the huge initial steps faster.
- Although we see beta=0.94 overtake the beta=0.6 as 0.6 gets stuck, for both large and small alphas.
- This is because the gradient becomes very flat towards the optimum, and hence the forgetful ones gain a larger step size more quickly, but this step size causes it to overstep to opposite sides causing chattering.
- The non-forgetful ones still are impacted by the large steps it had taken before, and therefore keeps the step size smaller avoid overstepping.

- This function required a very large number of iterations, due to the very steep nature of the function, which RMSProp cant perform well in, so needed large iterations to see behaviour.

2.1.2 Function 2

Figs 1 , 2, 5 shows similar plots for function 2.

- Among alpha=4, the beta=0.98 jumped further into the x1 dimension in the first iteration simply because the beta acts simply as a weight on the current gradient.
- Both alpha=100 shoot off, beta=0.98 has larger chattering, but it decreases faster due to beta being large and remembering previous magnitudes, and the fact that it started with larger steps.
  - The lower beta has trouble with the chattering, and the chattering doesnt reduce due to forgetting that it the large steps its taking, and therefore increasing step size.
- Worth noting had to drastically change alpha value between function 1 and 2.

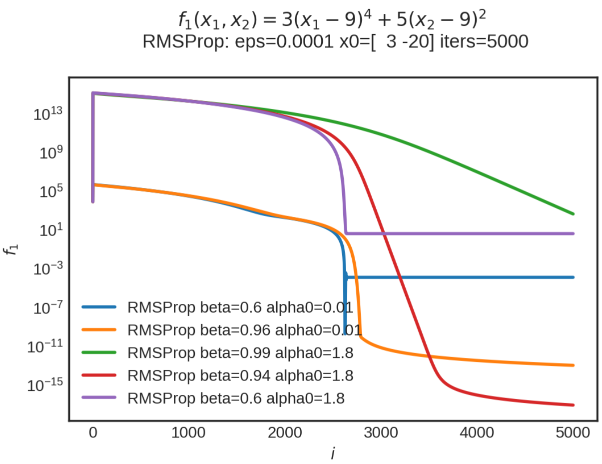

Figure 1

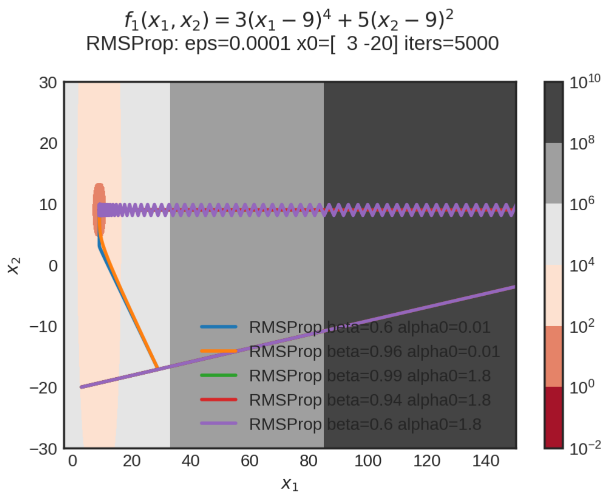

Figure 2

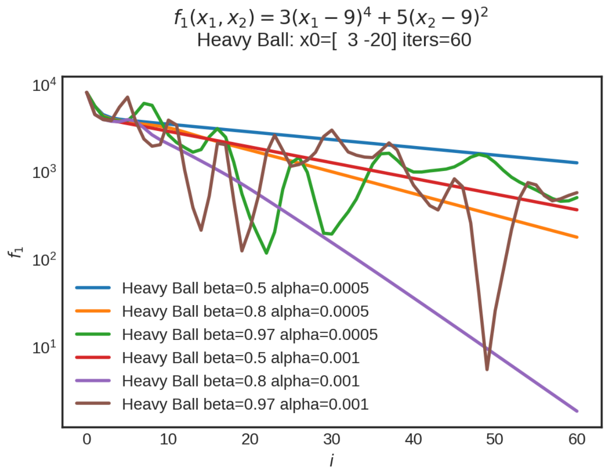

Figure 3

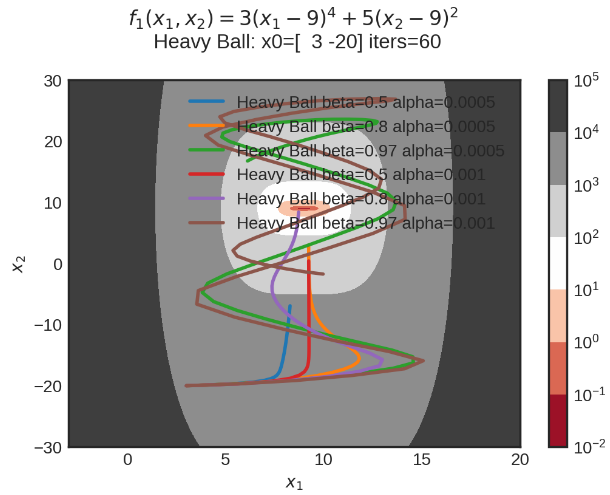

Figure 4

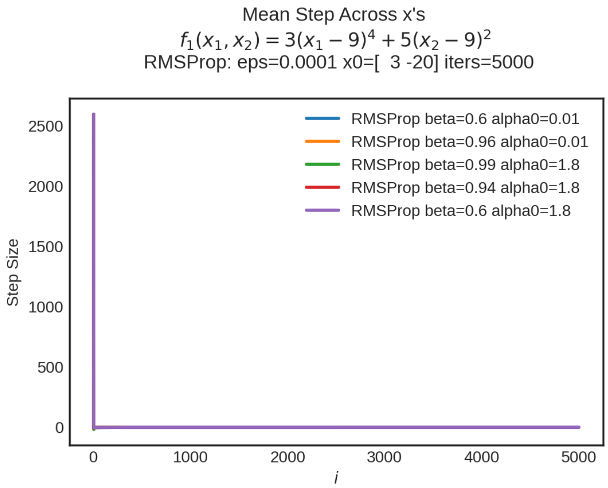

Figure 5

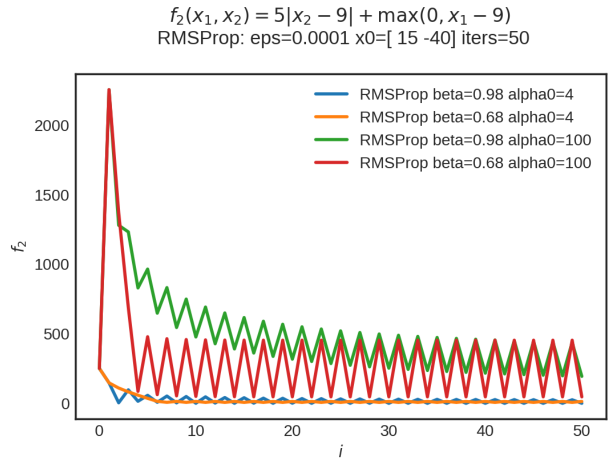

Figure 6

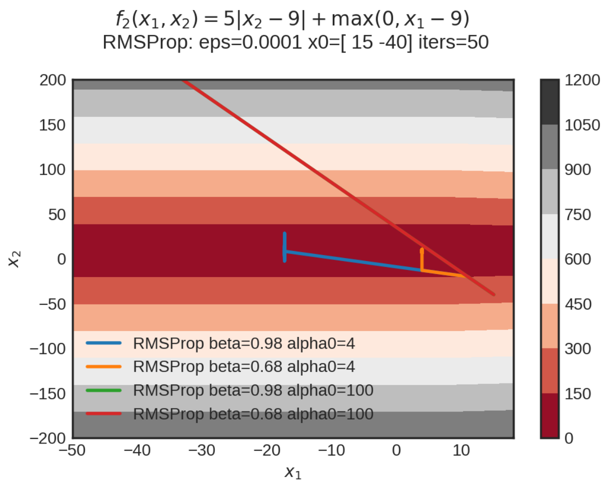

Figure 7

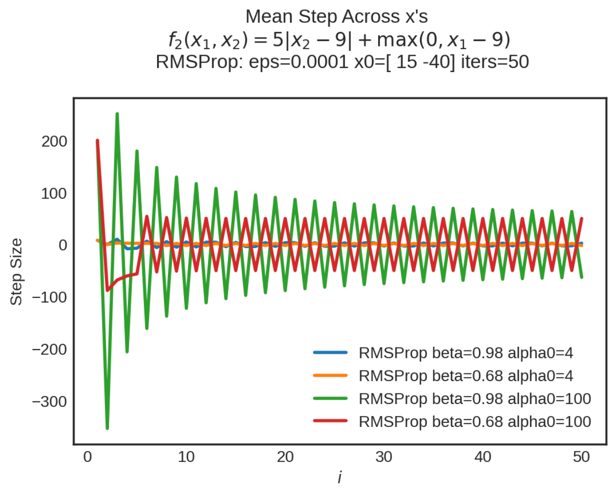

Figure 8

2.2 (ii) α and β in Heavy Ball

2.2.1 Function 1

Figs 3 , 4, 9 shows plots for Heavy Ball Function 1.

- Heavy ball extremely sensitive to alpha for this steep function, especially with high beta.
  - High beta causes it to maintain the momentum, and the initial steepness of the step will cause it to have a lot of momentum.
  - Even for smaller alphas, a high beta will still cause it to go back and forth a lot.
  - Smaller betas are better suited for the rapidly changing gradients where the optimum lies in this case.
    - Smaller beta will ditch the preceding momentums that the algorithm has gathered for more suitable step sizes closer to the optimum.
  - Alpha=0.001 beta=0.8 demonstartes the nice behavoiur.
  - Smaller betas, will cause constant step size behaviour.

2.2.2 Function 2

Figs 10 , 11, 12 shows same for function 1.

- Smaller betas tend to work better here, to discard momentum.
  - For this somewhat quadratic-like function, consant step size-like betas seems to work well.
  - Larger alphas cant really settle at the minimum, chattering happens even with alpha=0.5, due to the kink, it can never quite sit still in the kink to accumulate the low gradient momentum.
    - Same with small alpha and large beta, the momentum will cause it to jump out the the flat region a lot, and cause it to keep further accumulating momentum.

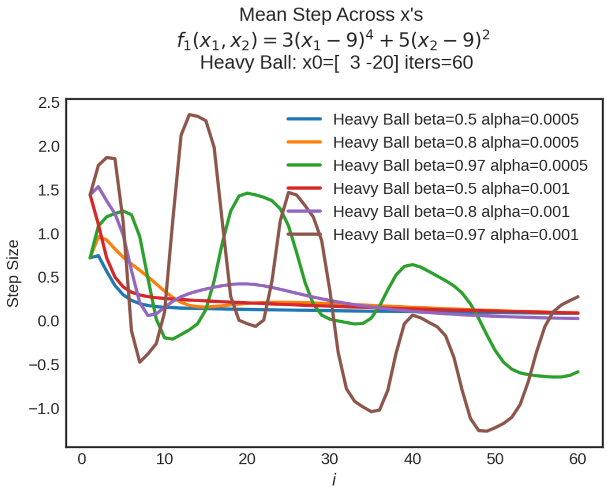

Figure 9

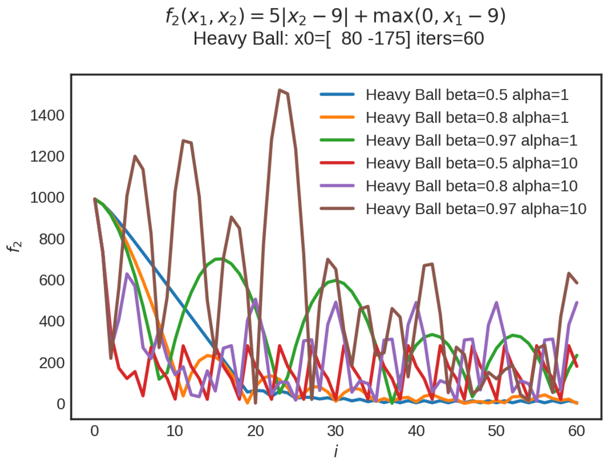

Figure 10

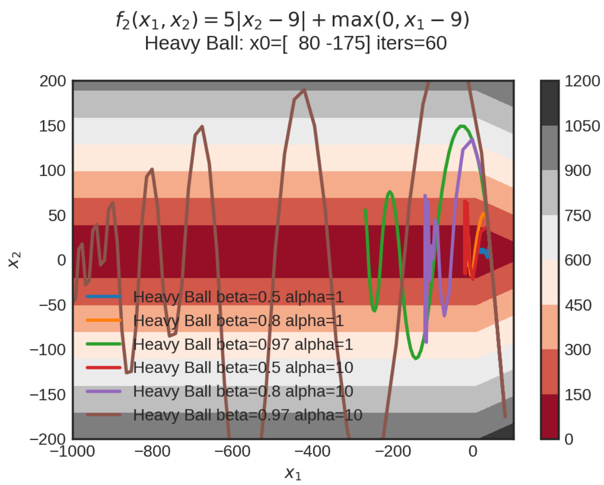

Figure 11

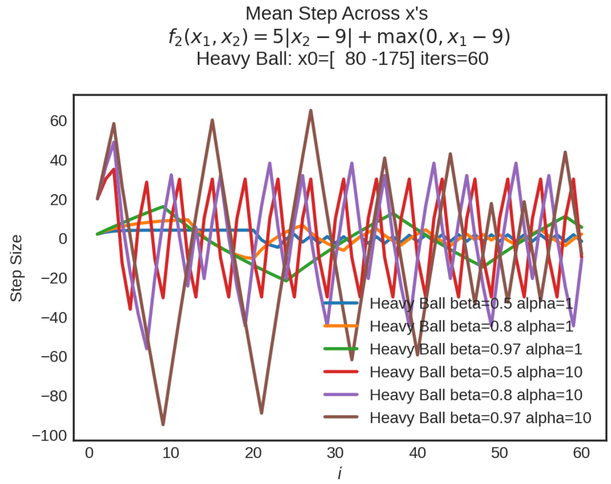

Figure 12

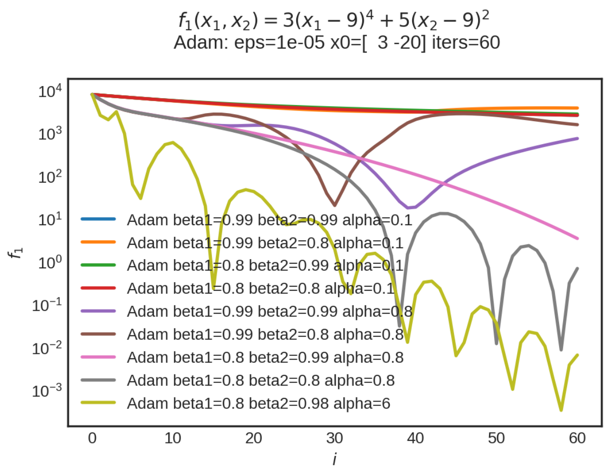

Figure 13

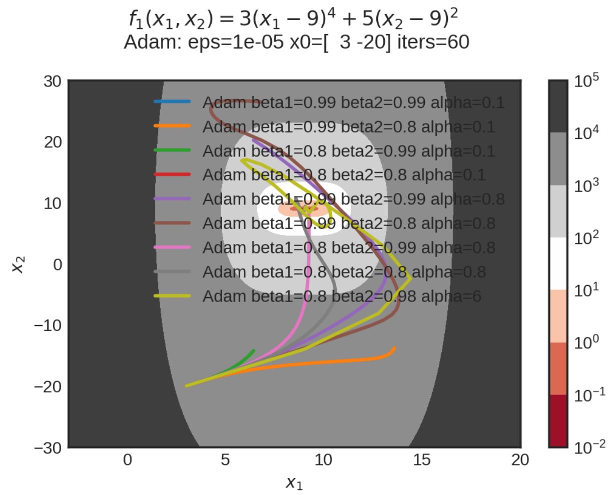

Figure 14

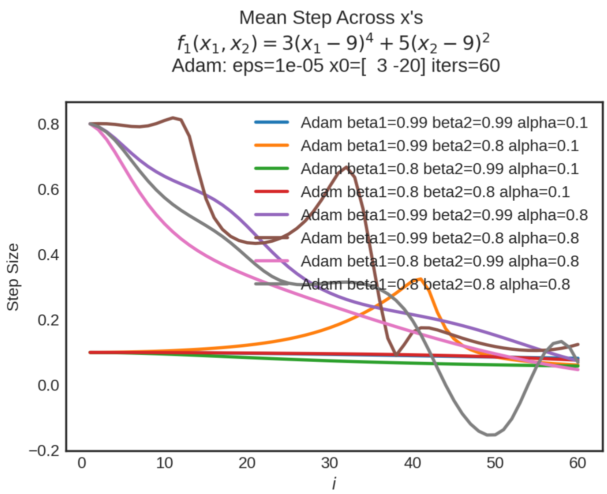

Figure 15

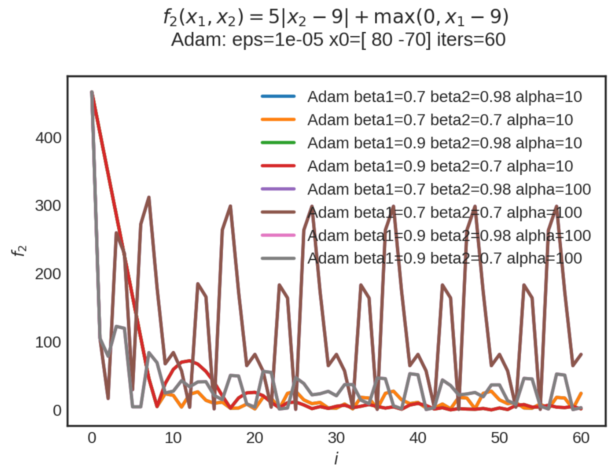

Figure 16

2.3 (iii) α, β_1 and β_2 in Adam

2.3.1 Function 1

Figs 13 , 14, 15 shows plots for Adam Function 1.

- Adam allows to crank up the alpha value but still cause it to converge nicely (beta1=0.8, beta2=0.98, alpha=6)
  - The RMS bit regulates the explosive steps.
- The momentum allows it to keep moving in the rapidly decreasing areas.
- beta1 is heavy ball bit, beta2 is rms bit.
- Low Heavy ball and High RMS with Low alpha doest let it move anywhere
  - Whereas the same onfig with but higher alpha steadily goes towards optimum
    * (beta1=.8 beta2=.99 alpha=.8)
  - Medium/Low Heavy Ball and High RMS could be a "steadiness".
    * High RMS meaning, the larger the gradients the slower it goes.
    * Medium Heavy Ball means its not going to overshoot the flat bits.
      · We can see same config with high Heavy ball (beta1=.99 beta2=.99 alpha=.8) it overshoots.
    * Low RMS is not bad too, but it still overshoots a bit due to not slowing down when it reaches low parts.
      · b1=0.8,b2=0.8,a=0.8
    * Low RMS and High momentum overshoots quite a lot
      · b1=0.99,b2=0.8,a=0.8
- "Steadiness" works well for rapidly changing slopes.

2.3.2 Function 2

Figs 16 , 17, 18 shows same for Function 2.

- Alpha can range a large amount and still give quite good performance depending on betas.
- Comparing
  - b1=0.7, b2=0.7, a=100
  - b1=0.9, b2=0.7, a=100
  - Increased heavy ball influences causes it average out the chattering caused by the massive step size.
  - Then looking at b1=0.9, b2=0.98, a=100, the rms bit causes it to stop the chattering quite quickly.
- The lower alphas are ideally behaved.
- Betas can caputre a behavoiur acoording to characteristics of the slopes.
  - Allowing heavy cranking of alpha.

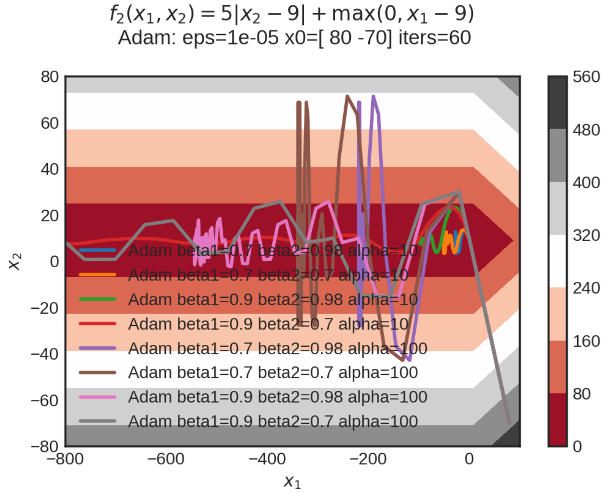

Figure 17

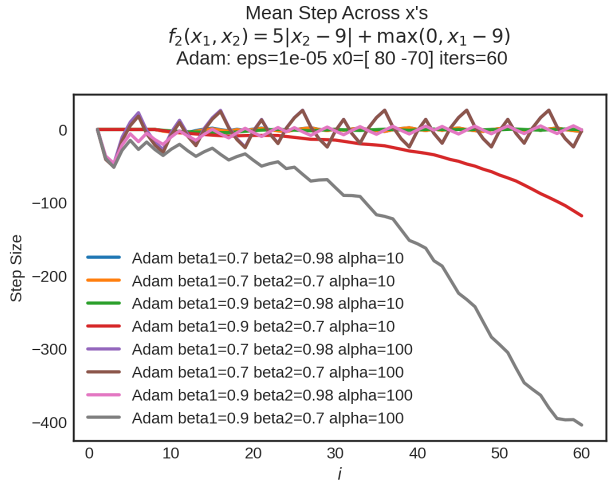

Figure 18

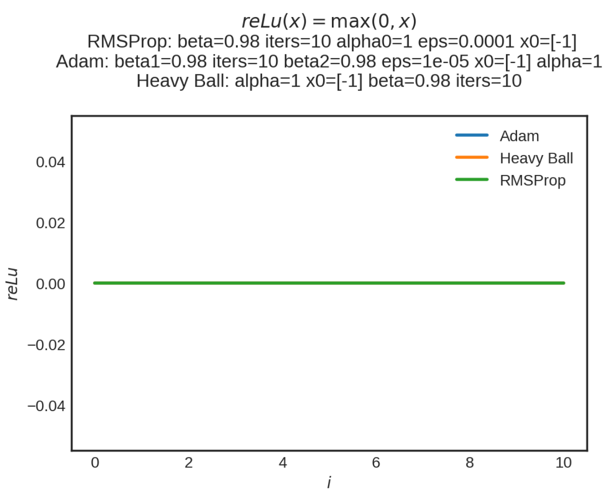

Figure 19

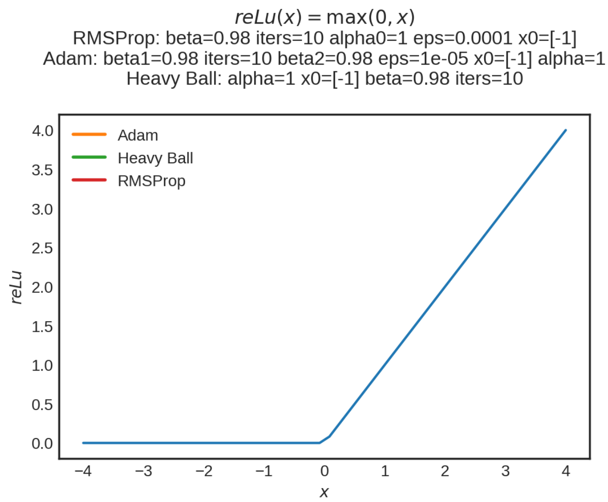

Figure 20

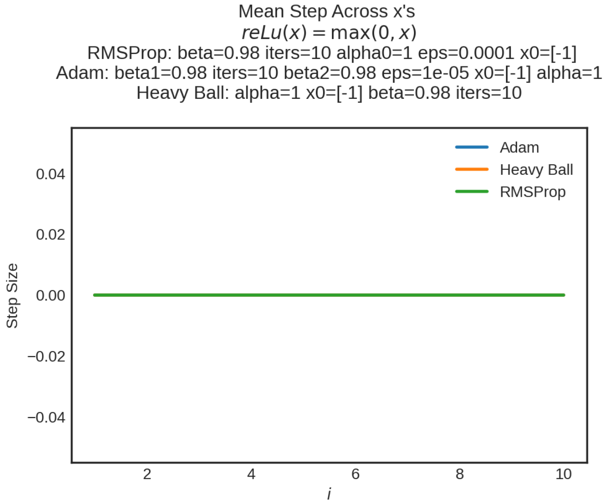

Figure 21

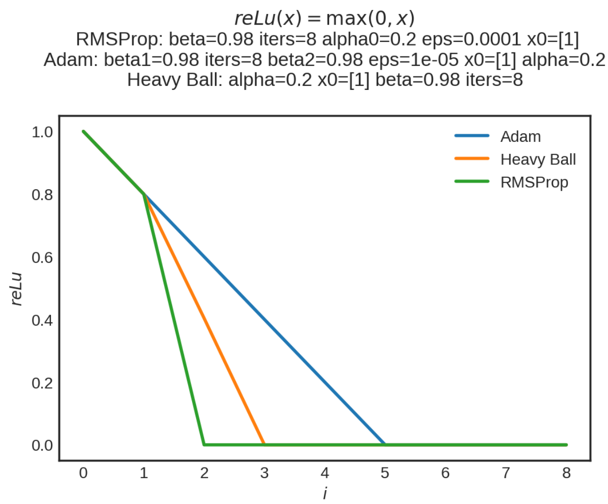

Figure 22

3 (c) Optimising ReLu - $Max(0, x)$

- (i) Initial Condition $x = -1$
  - Figs 19 , 20, 21
  - Start at no gradient, therefore doesnt move anywhere.
- (ii) Initial Condition $x = +1$
  - Figs 22 , 23, 24, 25, 26
  - All move towards 0, adam stick close to slope, rms and heavy ball jump over, heavy ball keeps going cause of momentum, rms just stays there because gradient is zero.
- (iii) Initial Condition $x = +100$
  - Figs 27 , 28, 29, 30, 31
  - Adam doesnt makes it least down the slope, heavy ball makes it down the most due to momentum, rms also does well although not as good as HB
  - RMS step size slows down over time.
  - Adam has constant step size behaviour

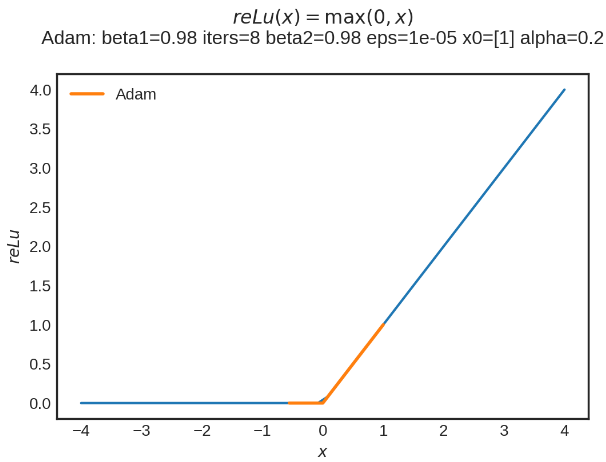

Figure 23

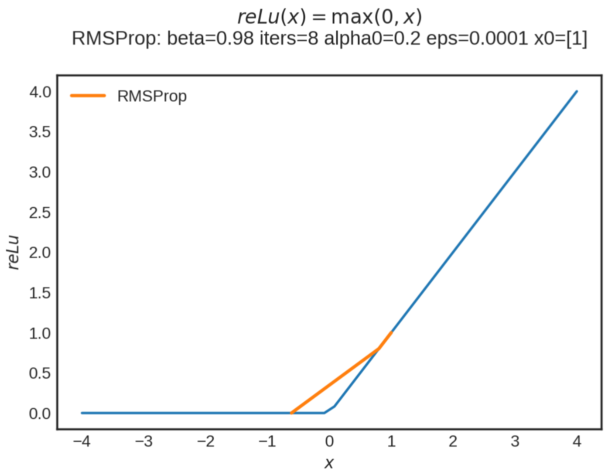

Figure 24

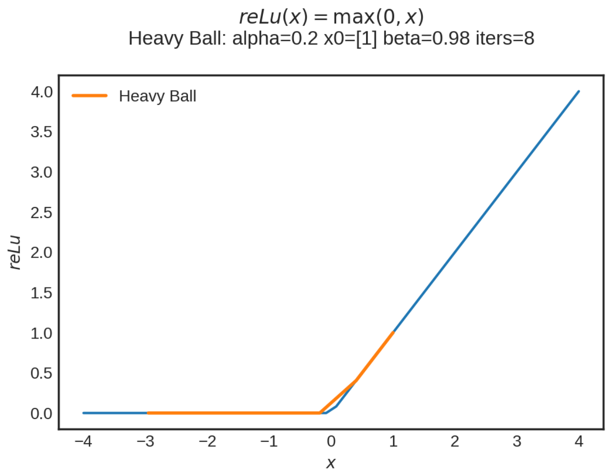

Figure 25

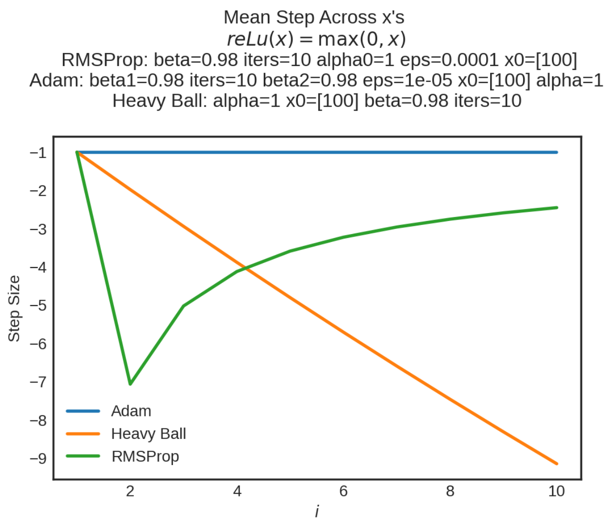

Figure 26

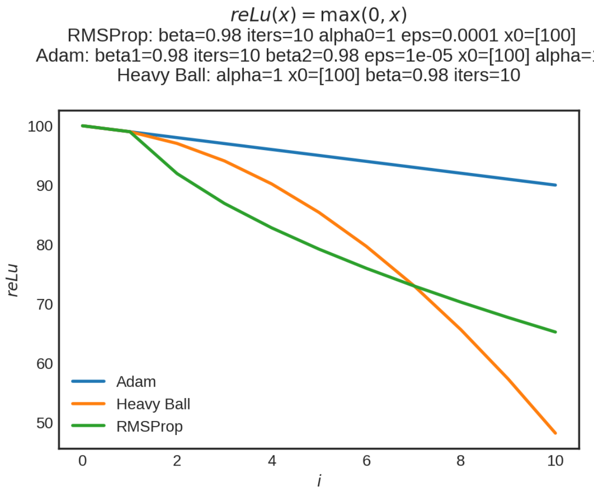

Figure 27

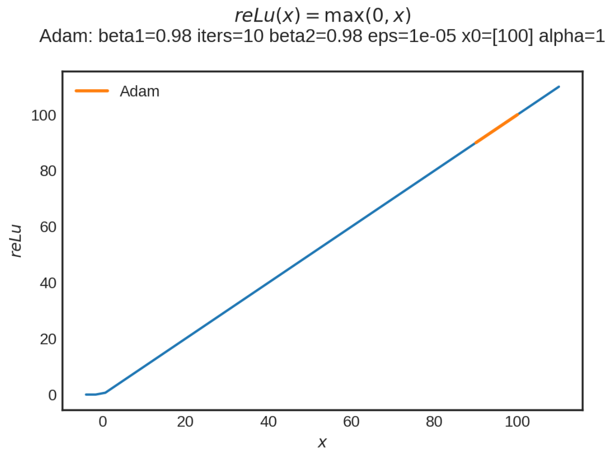

Figure 28

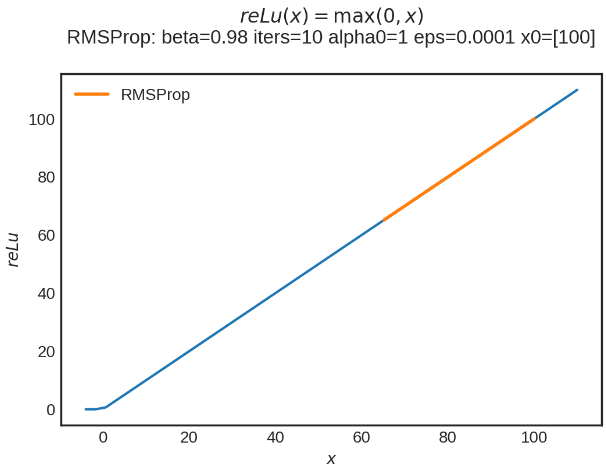

Figure 29

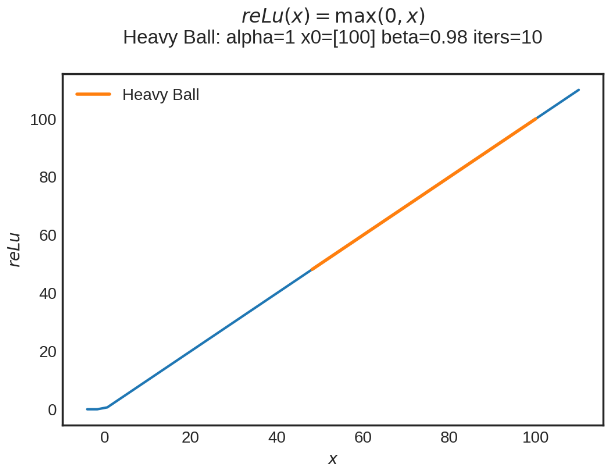

Figure 30

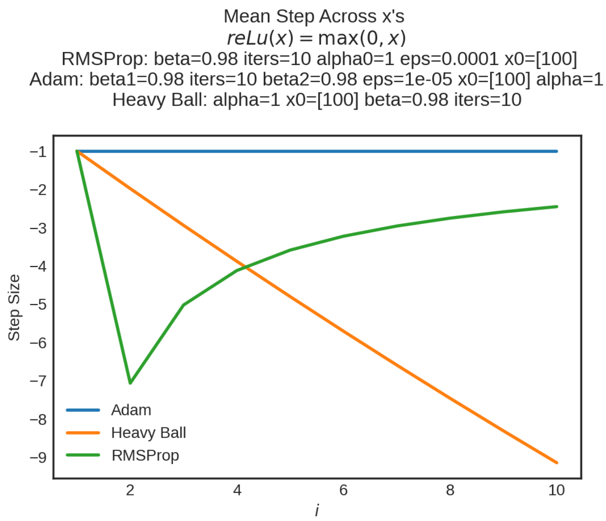

Figure 31

4 Appendix

4.1 Code Listing

```
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 200
mpl.rcParams['figure.facecolor'] = '1'
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
import copy
import numpy as np

# import OptimisationAlgorithmToolkit
from OptimisationAlgorithmToolkit.Function import OptimisableFunction
from OptimisationAlgorithmToolkit import Algorithms
from OptimisationAlgorithmToolkit import DataType
from OptimisationAlgorithmToolkit import Plotting
import importlib
importlib.reload(Algorithms)
importlib.reload(DataType)
importlib.reload(Plotting)
from OptimisationAlgorithmToolkit.Algorithms import Polyak, Adam, HeavyBall, RMSProp, Adagrad, ConstantStep
from OptimisationAlgorithmToolkit.DataType import create_labels, get_titles
from OptimisationAlgorithmToolkit.Plotting import ploty, plot_contour, plot_path, plot_step_size

from sympy import symbols, Max, Abs

x1, x2 = symbols('x1 x2', real=True)
sym_f1 = 3 * (x1-9)**4 + 5 * (x2-9)**2
f1 = OptimisableFunction(sym_f1, [x1, x2], "f_1")

sym_f2 = Max(x1-9 ,0) + 5 * Abs(x2-9)
f2 = OptimisableFunction(sym_f2, [x1, x2], "f_2")

x = symbols('x', real=True)
sym_f_quadratic = x**2
f_quadratic = OptimisableFunction(sym_f_quadratic, [x], "f_q")

from matplotlib.ticker import LogLocator

l = np.linspace(-3, 19, 40)
l2 = np.linspace(-100, 100, 40)
l2 = l
x1s = l
x2s = l2
X1, X2 = np.meshgrid(x1s, x2s)
Z = np.vectorize(f1.function)(X1, X2)
plt.contourf(X1, X2, Z, locator=LogLocator(), cmap='RdGy')
plt.colorbar();

l = np.linspace(-10, 40, 100)
x1s = l
x2s = l
X1, X2 = np.meshgrid(x1s, x2s)
Z = np.vectorize(f2.function)(X1, X2)
# plt.contour(X1, X2, Z, cmap='RdGy')
plt.contourf(X1, X2, Z, cmap='RdGy')
plt.colorbar();

for _ in range(iters):
    fdif = f(*x) - f_star
    df_squared_sum = np.sum(np.array([df(*x)**2 for df in dfs]))
    alpha = fdif / (df_squared_sum + eps)
    x = x - alpha * np.array([df(*x) for df in dfs])

sum = np.zeros(len(dfs)) ; alpha = alpha0
```

```
for _ in range(iters):
    x = x - (alpha * np.array([df(*x) for df in dfs]))
    sum = beta * sum + (1 - beta) * np.array([df(*x)**2 for df in dfs])
    alpha = alpha0 / (np.sqrt(sum) + eps)

z = np.zeros(len(dfs))
for _ in range(iters):
    z = beta * z + alpha * np.array([df(*x) for df in dfs])
    x = x - z

m = np.zeros(len(dfs)) ; v = np.zeros(len(dfs))
for k in range(iters):
    i = k + 1
    m = beta1 * m + (1 - beta1) * np.array([df(*x) for df in dfs])
    v = beta2 * v + (1 - beta2) * np.array([(df(*x)**2) for df in dfs])
    mhat = (m / (1 - beta1**i))
    vhat = (v / (1 - beta2**i))
    x = x - alpha * (mhat / (np.sqrt(vhat) + eps))

iters = 5000
o1 = RMSProp.set_parameters(
    x0=[3, -20],
    f=f1,
    iters=iters,
    alpha0=[0.01],
    beta=[0.6, 0.96],
    eps=0.0001).run()
o2 = RMSProp.set_parameters(
    x0=[3, -20],
    f=f1,
    iters=iters,
    alpha0=1.8,
    beta=[0.99, 0.94, 0.6],
    eps=0.0001).run()
o3 = o1 + o2
# o3 = o2

ploty(copy.deepcopy(o3)).semilogy()

x = np.linspace(-3, 150, 300)
y = np.linspace(-30, 30, 300)
plot_contour(copy.deepcopy(o3), x, y, log=True)

plot_step_size(copy.deepcopy(o3))

iters = 50
f = f2
x0 = [15, -40]
o1 = RMSProp.set_parameters(
    x0=x0,
    f=f,
    iters=iters,
    alpha0=[4, 100],
    beta=[0.98, 0.68],
    eps=0.0001).run()
o3 = o1

ploty(copy.deepcopy(o3))

x = np.linspace(-50, 18, 300)
y = np.linspace(-200, 200, 300)
plot_contour(copy.deepcopy(o3), x, y)

plot_step_size(copy.deepcopy(o3))

iters = 60
o1 = HeavyBall.set_parameters(
    x0=[3, -20],
```

```
    f=f1,
    iters=iters,
    alpha=[0.0005, 0.001],
    beta=[0.5, 0.8, 0.97]).run()
o3 = o1

ploty(copy.deepcopy(o3)).semilogy()

x = np.linspace(-3, 20, 300)
y = np.linspace(-30, 30, 300)
plot_contour(copy.deepcopy(o3), x, y, log=True)

plot_step_size(copy.deepcopy(o3))

iters = 60
o1 = HeavyBall.set_parameters(
    x0=[80, -175],
    f=f2,
    iters=iters,
    alpha=[1, 10],
    beta=[0.5, 0.8, 0.97]).run()
o3 = o1

ploty(copy.deepcopy(o3))

x = np.linspace(-1000, 100, 300)
y = np.linspace(-200, 200, 300)
plot_contour(copy.deepcopy(o3), x, y)

plot_step_size(copy.deepcopy(o3))

iters = 60
o1 = Adam.set_parameters(
    x0=[3, -20],
    f=f1,
    iters=iters,
    alpha=[0.1, 0.8],
    beta1=[0.99, 0.8],
    beta2=[0.99, 0.8],
    eps=1e-5).run()
o2 = Adam.set_parameters(
    x0=[3, -20],
    f=f1,
    iters=iters,
    alpha=[6],
    beta1=[0.8],
    beta2=[0.98],
    eps=1e-5).run()
o3 = o1 + o2

ploty(copy.deepcopy(o3)).semilogy()

x = np.linspace(-3, 20, 300)
y = np.linspace(-30, 30, 300)
plot_contour(copy.deepcopy(o3), x, y, log=True)

plot_step_size(copy.deepcopy(o3))

iters = 60
o1 = Adam.set_parameters(
    x0=[80, -70],
    f=f2,
    iters=iters,
    alpha=[10, 100],
    # beta1=[0.99, 0.8],
    # beta2=[0.99, 0.8],
    beta1=[0.7, 0.9],
    beta2=[0.98, 0.7],
```

```
    eps=1e-5).run()
o3 = o1

ploty(copy.deepcopy(o3))

x = np.linspace(-800, 100, 300)
y = np.linspace(-80, 80, 300)
plot_contour(copy.deepcopy(o3), x, y)

plot_step_size(copy.deepcopy(o3))

x = symbols('x', real=True)
sym_f_relu = Max(0, x)
f_relu = OptimisableFunction(sym_f_relu, [x], "reLu")

x_init = -1

adam_o = Adam.set_parameters(
    x0=[x_init],
    f=f_relu,
    iters=10,
    alpha=1,
    beta1=[0.98],
    beta2=[0.98],
    eps=1e-5).run()
heavyball_o = HeavyBall.set_parameters(
    x0=[x_init],
    f=f_relu,
    iters=10,
    alpha=[1],
    beta=[0.98]).run()
rmsprop_o = RMSProp.set_parameters(
    x0=[x_init],
    f=f_relu,
    iters=10,
    alpha0=[1],
    beta=[0.98],
    eps=0.0001).run()
o3 = adam_o + heavyball_o + rmsprop_o

ploty(copy.deepcopy(o3))

x = np.linspace(-4, 4, 50)
# plot_path(copy.deepcopy(o3), x)
plot_path(copy.deepcopy(o3), x)

plot_step_size(copy.deepcopy(o3))

x_init = +1

iters=8

adam_o = Adam.set_parameters(
    x0=[x_init],
    f=f_relu,
    iters=iters,
    alpha=0.2,
    beta1=[0.98],
    beta2=[0.98],
    eps=1e-5).run()
heavyball_o = HeavyBall.set_parameters(
    x0=[x_init],
    f=f_relu,
    iters=iters,
    alpha=[0.2],
    beta=[0.98]).run()
rmsprop_o = RMSProp.set_parameters(
    x0=[x_init],
```

```
    f=f_relu,
    iters=iters,
    alpha0=[0.2],
    beta=[0.98],
    eps=0.0001).run()
o3 = adam_o + heavyball_o + rmsprop_o

ploty(copy.deepcopy(o3))

x = np.linspace(-4, 4, 50)
plot_path(copy.deepcopy(adam_o), x)

x = np.linspace(-4, 4, 50)
plot_path(copy.deepcopy(rmsprop_o), x)

x = np.linspace(-4, 4, 50)
plot_path(copy.deepcopy(heavyball_o), x)

plot_step_size(copy.deepcopy(o3))

x_init = +100

adam_o = Adam.set_parameters(
    x0=[x_init],
    f=f_relu,
    iters=10,
    alpha=1,
    beta1=[0.98],
    beta2=[0.98],
    eps=1e-5).run()
heavyball_o = HeavyBall.set_parameters(
    x0=[x_init],
    f=f_relu,
    iters=10,
    alpha=[1],
    beta=[0.98]).run()
rmsprop_o = RMSProp.set_parameters(
    x0=[x_init],
    f=f_relu,
    iters=10,
    alpha0=[1],
    beta=[0.98],
    eps=0.0001).run()
o3 = adam_o + heavyball_o + rmsprop_o

ploty(copy.deepcopy(o3))

x = np.linspace(-4, 110, 50)
plot_path(copy.deepcopy(adam_o), x)

x = np.linspace(-4, 110, 50)
plot_path(copy.deepcopy(rmsprop_o), x)

x = np.linspace(-4, 110, 50)
plot_path(copy.deepcopy(heavyball_o), x)

plot_step_size(copy.deepcopy(o3))
```

```
# Algorithms.py

# Algorithms implement a similar inteface:
# - specific names on input arguments
# - accesses function related things through the OptimisableFunction class
# - needs to return X, Y

import numpy as np

class OptimisationAlgorithm:
    def __init__(self, algorithm, algorithm_name):
```

```
        self.algorithm = algorithm
        self.algorithm_name = algorithm_name

        arguments = algorithm.__code__.co_varnames[:algorithm.__code__.co_argcount]
        self.all_parameters = arguments
        self.standard_parameters = ("x0", "f", "iters")
        self.hyperparameters = list(filter(lambda arg: arg not in self.standard_parameters, arguments))

    def __type_check_parameters(self, input_record):
        for key in input_record.keys():
            if key not in self.all_parameters:
                raise NameError(key + " is not one of: " + str(self.all_parameters))
        for key in self.all_parameters:
            if key not in input_record:
                raise NameError(key + " is missing from input: " + str(list(input_record.keys())))

    def set_parameters(self, **input_record):
        self.__type_check_parameters(input_record)
        self.parameter_values = input_record
        return self

    def run(self):
        inputs = self.__make_input()
        for input in inputs:
            input["X"], input["Y"] = self.algorithm(**input)
            input["X"] = np.array(input["X"])
            input["Y"] = np.array(input["Y"])
            input["algorithm"] = self
        return inputs

    def __make_input(self):
        kwargs = self.parameter_values.copy()
        expected_vector = { "x0" }
        for key, value in kwargs.items():
            if key in expected_vector:
                value = np.array(value)
                if value.ndim == 1:
                    kwargs[key] = [value]
            else:
                if type(value) is not list:
                    kwargs[key] = [value]

        keys = kwargs.keys()
        partial_dicts = [{}]
        for key in keys:
            partial_dicts_new = []
            for partial_dict in partial_dicts:
                for value in kwargs[key]: # making a new partial dict for each value
                    partial_dict_new = partial_dict.copy()
                    partial_dict_new[key] = value
                    partial_dicts_new += [partial_dict_new]
                    partial_dicts = partial_dicts_new
        return partial_dicts

def polyak(x0, f, f_star, eps, iters):
    dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    for _ in range(iters):
        fdif = f(*x) - f_star
        df_squared_sum = np.sum(np.array([df(*x)**2 for df in dfs]))
        alpha = fdif / (df_squared_sum + eps)
        x = x - alpha * np.array([df(*x) for df in dfs])

        X += [x] ; Y += [f(*x)]
    return X, Y

```

```
Polyak = OptimisationAlgorithm(algorithm=polyak,
                               algorithm_name="Polyak")

def constant_step(x0, alpha, f, iters):
    dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    for _ in range(iters):
        step = alpha * np.array([df(*x) for df in dfs])
        x = x - step

        X += [x] ; Y += [f(*x)]
    return X, Y

ConstantStep = OptimisationAlgorithm(algorithm=constant_step,
                                     algorithm_name="Constant")

def adagrad(x0, f, alpha0, eps, iters):
    dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    df_vector_sum = np.zeros(len(dfs))
    for _ in range(iters):
        df_vec = np.array([df(*x) for df in dfs])
        df_vector_sum += df_vec**2
        alphas = alpha0 / (np.sqrt(df_vector_sum) + eps)
        x = x  - (alphas * df_vec)

        X += [x] ; Y += [f(*x)]
    return X, Y

Adagrad = OptimisationAlgorithm(algorithm=adagrad,
                                algorithm_name="Adagrad")

def rmsprop(x0, f, alpha0, beta, eps, iters):
    dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    sum = np.zeros(len(dfs)) ; alpha = alpha0
    for _ in range(iters):
      x = x - (alpha * np.array([df(*x) for df in dfs]))
      sum = beta * sum + (1 - beta) * np.array([df(*x)**2 for df in dfs])
      alpha = alpha0 / (np.sqrt(sum) + eps)

      X += [x] ; Y += [f(*x)]
    return X, Y

RMSProp = OptimisationAlgorithm(algorithm=rmsprop,
                                algorithm_name="RMSProp")


def heavy_ball(x0, f, alpha, beta, iters):
    dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    z = np.zeros(len(dfs))
    for _ in range(iters):
        z = beta * z + alpha * np.array([df(*x) for df in dfs])
        x = x - z

        X += [x] ; Y += [f(*x)]
    return X, Y

HeavyBall = OptimisationAlgorithm(algorithm=heavy_ball,
                                  algorithm_name="Heavy Ball")

def adam(x0, f, eps, beta1, beta2, alpha, iters):
    dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    m = np.zeros(len(dfs)) ; v = np.zeros(len(dfs))
    for k in range(iters):
        i = k + 1
```

```
        m = beta1 * m + (1 - beta1) * np.array([df(*x) for df in dfs])
        v = beta2 * v + (1 - beta2) * np.array([(df(*x)**2) for df in dfs])
        mhat = (m / (1 - beta1**i))
        vhat = (v / (1 - beta2**i))
        x = x - alpha * (mhat / (np.sqrt(vhat) + eps))

        X += [x] ; Y += [f(*x)]
    return X,Y

Adam = OptimisationAlgorithm(algorithm=adam,
                             algorithm_name="Adam")
```

```
# Each record should contain its label depending on what are the other records in the
    list.

# The user semi-mannually inputs what the title should be.
# - Have utility functions to extract pieces of the title from the list of records.

# Function that takes in a list of records.
#  - For each record determines the label based on what is in the list of records.

# Perhaps there should be a function that calculatesthe meta information that is used
    by both
# - utility functions that extract peieces of title
# - function that assigns the labels to each individual record


# MetaInfo: extracts:
# - Which optimisaiton functions there area
# - For each optimisation function
#   - What are the parameters that are not varying and what values do they have
#   - What are the parameters that are varying and what values do they have




# {
#   ...
#   ...
#   label:
# }
# label made up from what uniquely identifies it
# - first is optimisation algorithm itself
# - second are the hyperparmeters that uniqely identifies the cluster of algorithms
#   - RMSProp alpha0=0.4
#   - RMSProp alpha0=0.5
#   - Adam    beta1=0.2  beta2=0.4
#   - Adam    beta1=0.3  beta2=0.5

# - Then would like to extract the common descriptive pieces
#   - Different common pieces per algorithm used
#     - Records -> AlgorihtmName -> CommonThingsString
#       - Adam: beta1=0.1 eps=0.0001 iters=50 x0=[1, 1]
#       - RMSProp:  eps=0.0001 iters=50 x0=[1, 1]


# MetaRecord extracts
# - Algorithms and their corresponding Varying fields
# {
#   "Adam"    : ["eps", "beta1"]
#   "RMSProp" : ["eps", "alpha0"]
# }


# meta_record = meta(inputs)
# inputs = create_labels(meta_record, inputs)
# inputs = get_title(meta_record, inputs)

# get_titles returns
```

```python
# {
#   "Adam" : "Adam: beta1=0.1 eps=0.0001 iters=50 x0=[1, 1]",
#   "RMSProp" : "RMSProp:  eps=0.0001 iters=50 x0=[1, 1]"
# }

import numpy as np

def get_titles(records):
    m = meta(records)
    t = {}
    for alg_name in m.keys():
        t[alg_name] = get_title(alg_name, records, m)
    return t

def get_title(alg_name, records, meta):
    title = f'{alg_name}:'
    algs = alg(records, alg_name)

    r = algs[0]
    params = set(r["algorithm"].all_parameters)
    varied = meta[alg_name]
    params.remove('f')
    params = params - varied

    for p in params:
        title += f' {p}={r[p]}'
    return title

def create_labels(records):
    m = meta(records)
    for r in records:
        r['label'] = create_label(r, m)

# e.g: Adam    beta1=0.2  beta2=0.4
def create_label(record, meta):
    alg_name = record['algorithm'].algorithm_name
    differing_fields = meta[alg_name]
    label = f'{alg_name}'
    for f in differing_fields:
        label += f' {f}={record[f]}'
    return label

# {
#   "Adam"    : ["eps", "beta1"]
#   "RMSProp" : ["eps", "alpha0"]
# }
def meta(records):
    mr = {}
    algs = get_algs(records)
    for a in algs:
        a_records = alg(records, a)
        mr[a] = differing_fields(a_records)
    return mr

def differing_fields(records):
    diff_fields = set({})
    t = records[0]
    for r in records:
        for key, value in r.items():
            # print("a")
            # print(t[key])
            # print(type(value))
            # print(isinstance(value, list))

            if isinstance(value, list):
                value = np.array(value)
            if isinstance(t[key], list):
                t[key] = np.array(t[key])
```

```

            b = t[key] == value
            # print(b)
            # print(type(b))
            if type(b) == np.ndarray:
                b = b.all()
            if not (b):
                diff_fields.add(key)


    diff_fields.discard('X')
    diff_fields.discard('Y')
    return diff_fields

# extract one algorithm type, filter out the rest
def alg(records, algorithm_name):
    return list(filter(lambda r: r['algorithm'].algorithm_name == algorithm_name, records))

# gets algorithms names in the records
def get_algs(records):
    algs = set({})
    for r in records:
        algs.add(r['algorithm'].algorithm_name)
    return algs


# wonder how this would look in haskell
# funcitonal operators and stuff, would it make it easier.
```

```
# Functions that will be optimised:
# - Allows access to
#   - Parital Derivatives
#   - String representation of the function (latex)
# - Constructor uses sympy to obtain the above

from sympy import simplify, latex, lambdify
import numpy as np

class OptimisableFunction:
    def __init__(self, sympy_function, sympy_symbols, function_name):
        self.sympy_symbols = sympy_symbols
        self.function_name = function_name

        self.sympy_function = sympy_function
        self.function = lambdify(sympy_symbols, sympy_function, modules="numpy")

        self.sympy_partial_derivatives = [sympy_function.diff(symbol) for symbol in sympy_symbols]
        self.partial_derivatives = [lambdify(sympy_symbols, p, modules="numpy") for p in self.sympy_partial_derivatives]

    def __parameters_string(self):
        s = map(latex, self.sympy_symbols)
        return ",".join(s)

    def latex(self):
        return self.function_name + "(" + self.__parameters_string() + ") = " + latex(simplify(self.sympy_function))

    def partials_latex(self):
        s = map(latex, self.sympy_symbols)
        z = zip(self.sympy_partial_derivatives, s)
        return [ "\\frac{\\partial " +  self.function_name + "}{\\partial " + partial_wrt_name + "}" "=" + latex(simplify(partial))
                for (partial, partial_wrt_name) in z]

    def print_partials_latex(self):
        for p in self.partials_latex():
```

```
            print(p)
```

```
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 200
mpl.rcParams['figure.facecolor'] = '1'
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')

from OptimisationAlgorithmToolkit.DataType import create_labels, get_titles

from matplotlib.ticker import LogLocator

import numpy as np

def plot_contour(records, x1r, x2r, log=False):
    create_labels(records)
    t = get_titles(records)

    f = records[0]['f'].function;
    f_name = records[0]['f'].function_name;
    f_latex = records[0]['f'].latex()

    X1, X2 = np.meshgrid(x1r, x2r)
    Z = np.vectorize(f)(X1, X2)

    if log:
        plt.contourf(X1, X2, Z, locator=LogLocator(), cmap='RdGy')
    else:
        plt.contourf(X1, X2, Z, cmap='RdGy')
    xlim = plt.xlim()
    ylim = plt.ylim()

    for (X, label) in dicts_collect(("X", "label"), records):
        plt.plot(X.T[0], X.T[1], linewidth=2.0, label=label)

    plt.xlabel(r'$x_1$')
    plt.ylabel(r'$x_2$')

    title = rf'${f_latex}$' + " \n " + title_string(records)
    plt.title(title)

    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.legend()
    plt.colorbar()

def plot_path(records, xr):
    create_labels(records)
    f = records[0]['f'].function;
    function_name = records[0]['f'].function_name
    f_latex = records[0]['f'].latex()

    yr = [f(x) for x in xr]
    plt.plot(xr, yr)
    xlim = plt.xlim()
    ylim = plt.ylim()

    for (X, label) in dicts_collect(("X", "label"), records):
        xs = X.flatten()
        ys = [f(x) for x in xs]
        plt.plot(xs, ys, linewidth=2.0, label=label)

    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.legend()
    title = rf'${f_latex}$' + "\n" + title_string(records)
    plt.title(title)
    plt.ylabel(f'${function_name}$')
    plt.xlabel(r'$x$')
```

```python

def plot_step_size(records, mean=True):
    create_labels(records)
    fig, ax = plt.subplots()
    f_latex = records[0]['f'].latex()
    for (X, label) in dicts_collect(("X", "label"), records):
        if mean:
            s = np.array([np.mean(x) for x in  step_sizes(X).T])
            ax.plot(np.arange(1, len(s)+1), s, linewidth=2.0, label=label)
        else:
            sX = step_sizes(X)
            for i in range(len(sX)):
                x = i + 1
                s = sX[i]
                ax.plot(np.arange(1, len(s)+1), s, linewidth=2.0, label=label + f'
    $x_{x} step$')
    ax.legend()

    title = rf'${f_latex}$' + " \n " + title_string(records)
    if mean:
        ax.set_title("Mean Step Across x's \n" + title)
    else:
        ax.set_title("Mean Step Across x's \n" + title)
    ax.set_ylabel(f'Step Size')
    ax.set_xlabel(r'$i$')


def title_string(records):
    title = ""
    t = get_titles(records)
    for _, v in t.items():
        title += v + '\n'
    return title

# [[x11 x21 x31 ...] [x12 x22 x32 ...] ...]  -> [[x12-x11 x13-x12 ...] [x22-x21 x23-
    x22 ...] ...]
def step_sizes(X):
    return np.array([(x[1:] - x[:-1]) for x in X.T])



def ploty(records):
    create_labels(records)
    t = get_titles(records)
    f_latex = records[0]['f'].latex()

    fig, ax = plt.subplots()
    for (X, Y, label) in dicts_collect(("X", "Y", "label"), records):
        ax.plot(range(len(Y)), Y, linewidth=2.0, label=label)

    f = records[0]['f']
    function_name = f.function_name

    title = rf'${f_latex}$' + " \n " + title_string(records)


    ax.set_title(title)

    ax.set_ylabel(f'${function_name}$')
    ax.set_xlabel(r'$i$')
    ax.legend()
    return ax

def dicts_collect(keys, dicts):
    values = []
    for dict in dicts:
        values += [[dict[key] for key in keys]]
    return values
```