Optimisation Algorithms - Week 6 Assignment

Ernests Kuznecovs - 17332791 - kuznecoe@tcd.ie

2nd March

Contents

1 (a) Stochastic Gradient Descent

1.1 (i) Implementation of SGD

- Use approximate derivatives $Df_{\theta_1}(\theta)$ instead of exact derivatives $\frac{\partial f}{\partial \theta_1}(\theta)$
- For ML we are trying to optimise the function:
  - $J(\theta) = \frac{1}{m}\sum\limits_{i=1}^{m} loss(\theta, x^{(i)}, y^{(i)})$
  - $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$ is our training data.
  - Real derivatives are: $\frac{\partial J}{\partial \theta_1}(\theta) = \frac{1}{m}\sum\limits_{i=1}^{m} \frac{\partial loss}{\partial \theta_1}(\theta, x^{(i)}, y^{(i)})$, $\frac{\partial J}{\partial \theta_2}(\theta) = \frac{1}{m}\sum\limits_{i=1}^{m} \frac{\partial loss}{\partial \theta_2}(\theta, x^{(i)}, y^{(i)})$
- Pick random sample of b points from training data.
- Let N be the set of b indices.
- Then use approx derivatives:
  - $DJ_{\theta_1}(\theta) = \frac{1}{b}\sum\limits_{i \in N} \frac{\partial loss}{\partial \theta_1}(\theta, x^{(i)}, y^{(i)}), DJ_{\theta_2}(\theta) = \frac{1}{b}\sum\limits_{i \in N} \frac{\partial loss}{\partial \theta_2}(\theta, x^{(i)}, y^{(i)})$
- Below is an implementation using constant step size that uses all the data in the epoch rather than sampling randomly from the data each iteration.
  - The data is shuffled at the start of the "epoch" to have the effect of random sampling.

- During the iterations the same data cant be picked twice and data won't be wasted. Batches of size b are used to form the approximate derivative.
- finite difference method is used to get the derivatives for each parital i.

```
x = np.array([1, 2]); b = 5; m = len(M); alpha=0.4; iters=50
for _ in range(iters):
    np.random.shuffle(D)
    for i in np.arange(0, m, b): # 0 upto m-1, in steps of b. i.e index of each batch
     start
        N = np.arange(i, i + b)
        fN = lambda x: f(x, minibatch=M[N])
        DJ = np.array([finite_diff(fN, x, i) for i in range(len(x))])
        x = x - alpha * DJ
```

- Generalised version used provided in the appendix in which the function to be optimised is implemented as a Python iterator which returns a new set of approximate derivatives upon each iteration based on the batch. Each step size algorithm from previous assignment is adjusted to use this function iterator to retrieve the approximate gradients each iteration, thus giving us stochastic gradients, e.g for polyak:

```
def polyak(x0, f, f_star, eps, iters, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*
    x)]

    for fN, dfs in fi:
        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
```

1.2 (ii) Plotting Loss Function to Optimise

Fig 1 is the contour plot. Fig 2 is the wireframe plot.

The range of values chosen are -20 and 5. This is because for larger range it is strongly convex, but when zoomed in, there is an interesting dip to the side of the global minimum, which can be considered as a local minimum to test how the algorithm may behave coming across it. Also the function increses rapidly beyond -20 and 5, it's already at 10^3.

1.3 (iii) Calculating Derivative of f

- finite diff is defined such that can specify which input parameter of the function we add the perturbation.

```
def finite_diff(f, x, i, delta=0.0001):
    d = np.zeros(len(x)) ; d[i] = delta
    return (f(x) - f(x - d)) / delta
```

- We index our dataset M with N and create a closure in the lambda capturing the batch.
- Then we can pass the resulting function to the finite difference function.

```
fN = lambda x: f(x, minibatch=M[N])
x = np.array([10, 10])
Dfx1 = finite_diff(fN, x, 0) # w.r.t x1
Dfx2 = finite_diff(fN, x, 1) # w.r.t x2
```

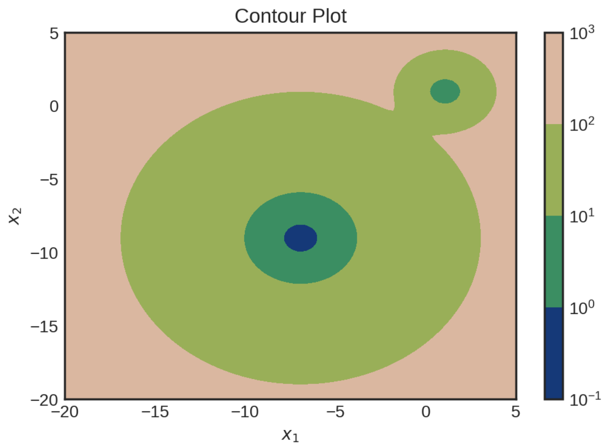

Figure 1

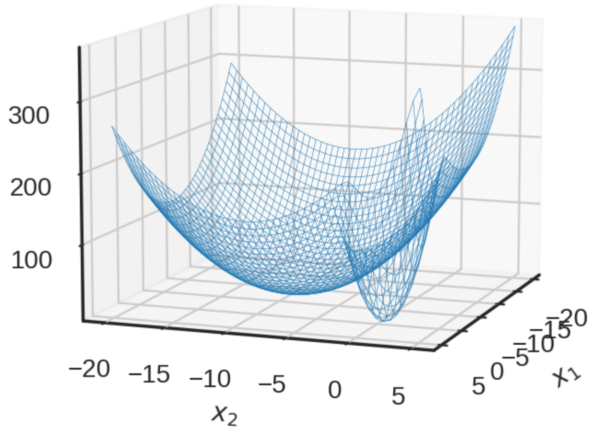

Figure 2

2 (b) Optimising f

- 25 datapoints are used for function f

2.1 (i) Gradient Descent with Constant Step-Size

A value of alpha of 0.085 is picked such that the gradient descent gets stuck in the local minimum, but an alpha a bit higher will cause it to escape. Perhaps the SGD will demonstrate that it will be able to escape it.

Fig 3 is gradient decent with constant step plotting y value acorss iteration. Fig 4 is gradient decent with constant step on countour plot.

The GD is seen to do a form of chattering around the local minimum in Fig 3

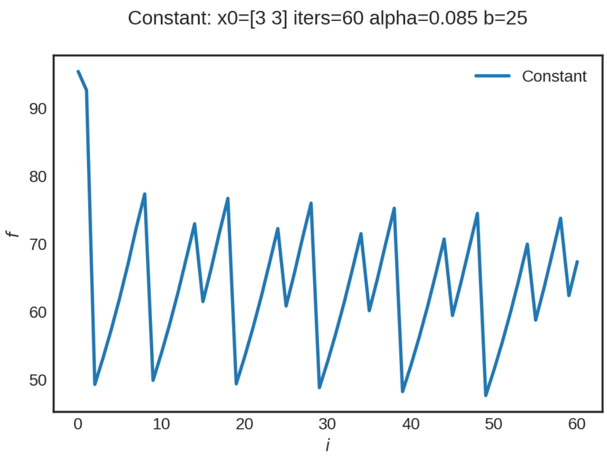

Figure 3

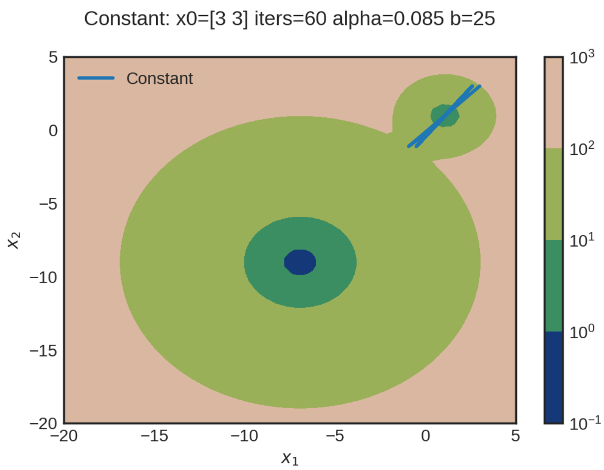

Figure 4

2.2 (ii) Mini-Batch Stochastic Gradient Descent

Figs Run 1 = (5, 6), Run 2 = (7, 8), Run 3 = (9, 10) shows the variance between runs. It can sometimes escape the local minimum, depending on if it gets the luckly batch of data that forms the function/gradient at critical times. We can see that the algorithm can walk around at the local mimimum, and then escape. And we also see that it can get luckly and it gets the lucky batch in a timely manner to avoid the dance at the local minimum and directly step over it. Perhaps a batch causes the slope to increase and allows for the step to hop over.

In Fig 6 we can see that the y value gets quite close to the minimum value, but perhaps the function is quite volatile at the small bowl due its size and then it gets a batch/gradient that allows it to jump out.

In gradient descent, it is stuck chattering at a predictable, preiodic fashion, this is because the gradients stay the same for the x1 and x2's the algorithm finds itself at. Whereas the chattering seen in 6 is not periodic at all due to the varying approximate derivatives due to the random sampling of the data that constructs the function.

2.3 (iii) Varying Mini-Batch Size on SGD

Figs Run 1 = (11, 12), Run 2 = (13, 14), Run 3 = (15, 16) shows various runs with various batch sizes. Batch size of 1 almost always escapse the local minimum, batch size 25 (out of 25 data points) never escapes. While batch sizes 5 and 10 sometimes escape.

The point at which x converges varies with batch size as the approximate gradient gets more and more noisy the smaller the batch size. In this case, all that is needed for it to converge to the globabl minimum is to get outside the local minimum. There is a higher chance that the algorithm will escape the local minimum when there is a lot of noise. Smaller batch sizes will encourage escape from narrow optimum points, which is a good thing as once the model is used on unseen data, that narrow point may be subject to change. In other words, it doesn't take much variance in the data to change a narrow parts of a function. Where as a large areas of a minimum might be a place where noise won't affect it as much.

2.4 (iv) Varying Step Size on SGD

Figs Run 1 = (17, 18), Run 2 = (19, 20), Run 3 = (21, 22) Alpha=0.06 doesnt seems like it has a miniscule chance to walk out of the local minimum. Whereas alpha=0.085 sometimes does, and alpha=0.1 almost always does.

The smaller alphas move too slowly around the local minimum and therfore the many iterations that happen under it average out too fast into the local minimum bowl. Whereas the higher the alpha the less affected they the averaging affect, and need less successive lucky gradients to get out.

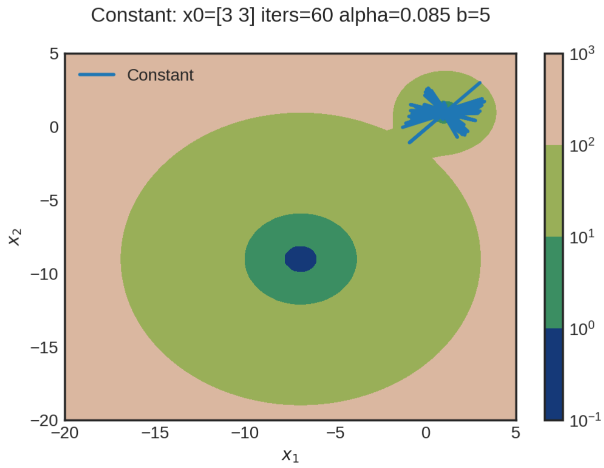

Figure 5

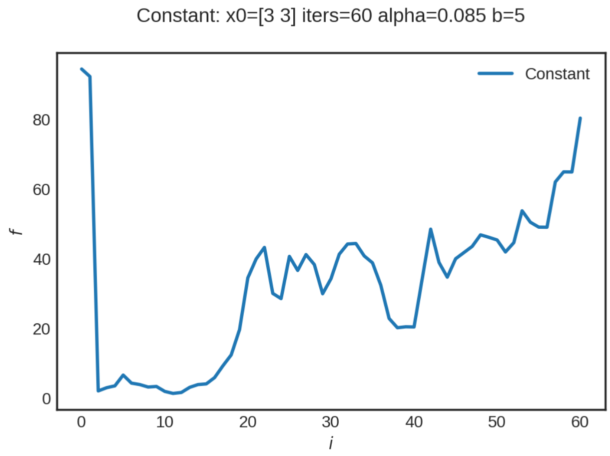

Figure 6

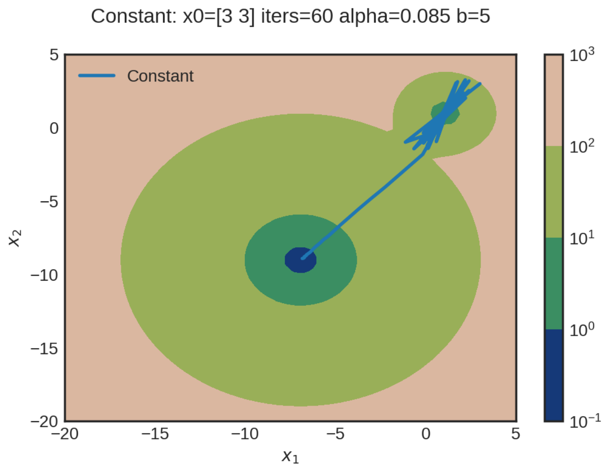

Figure 7

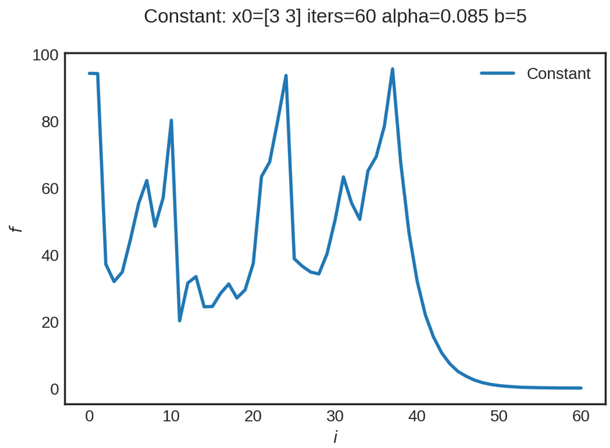

Figure 8

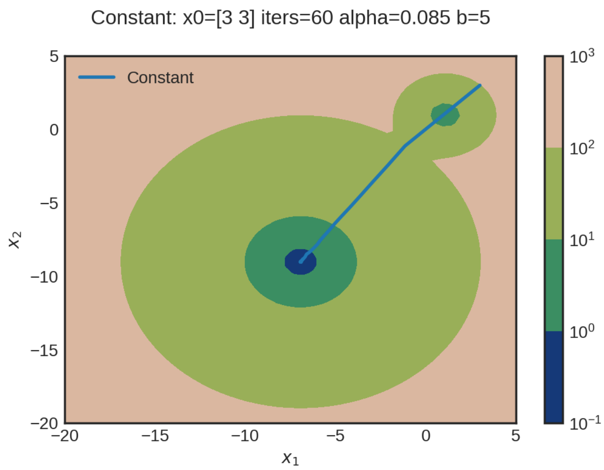

Figure 9

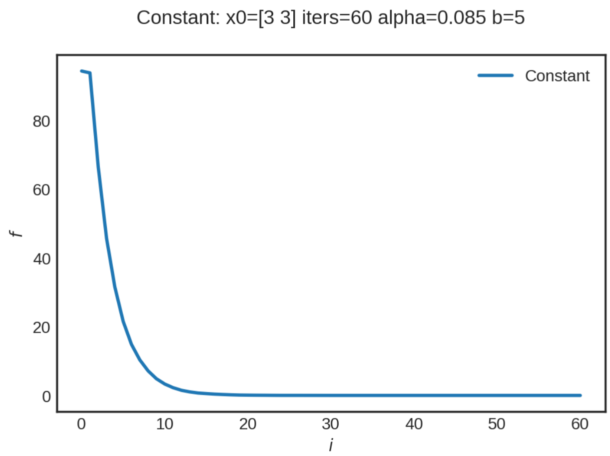

Figure 10

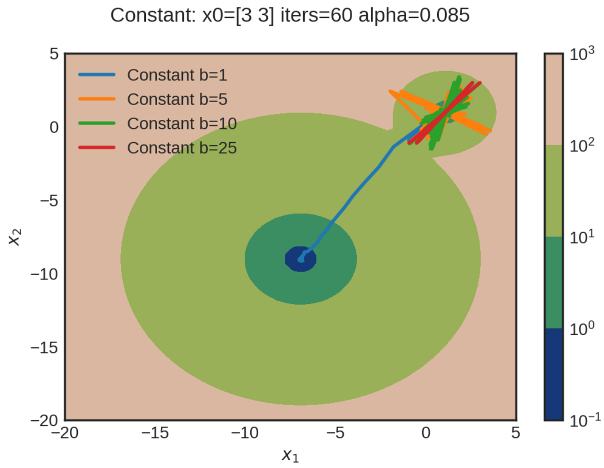

Figure 11

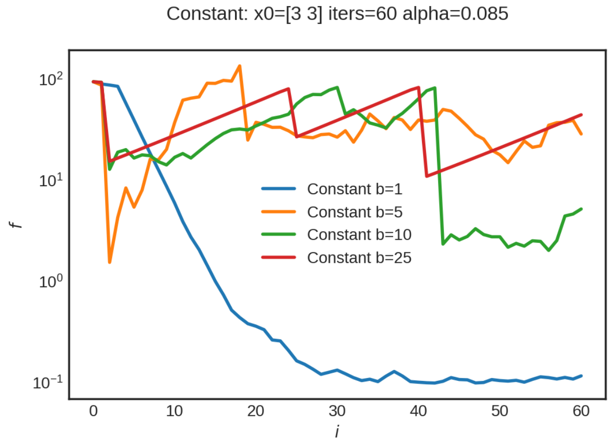

Figure 12

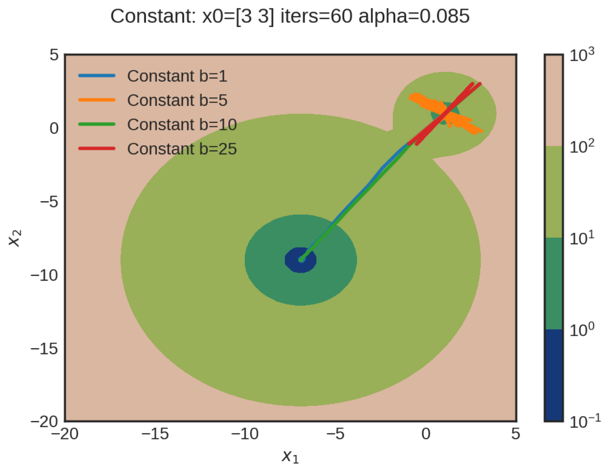

Figure 13

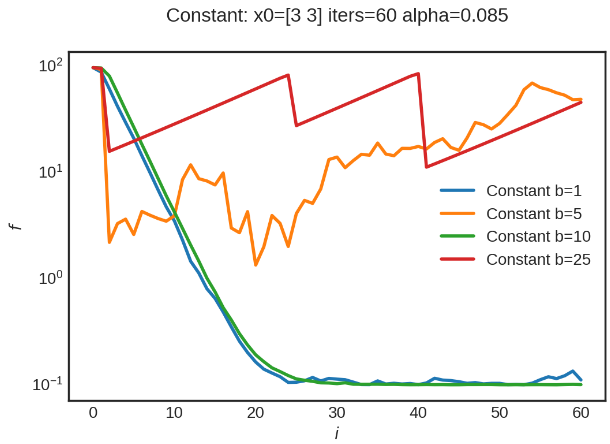

Figure 14

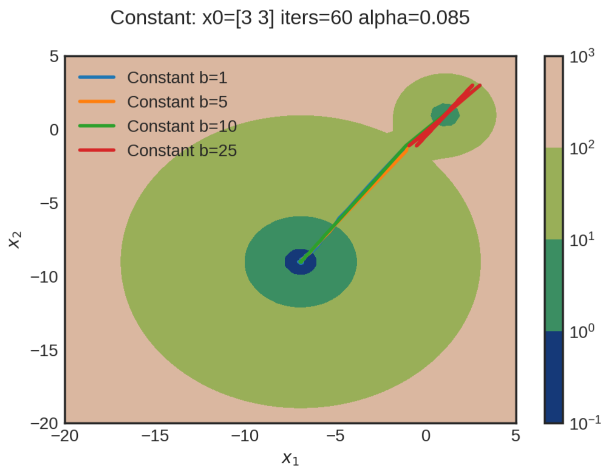

Figure 15

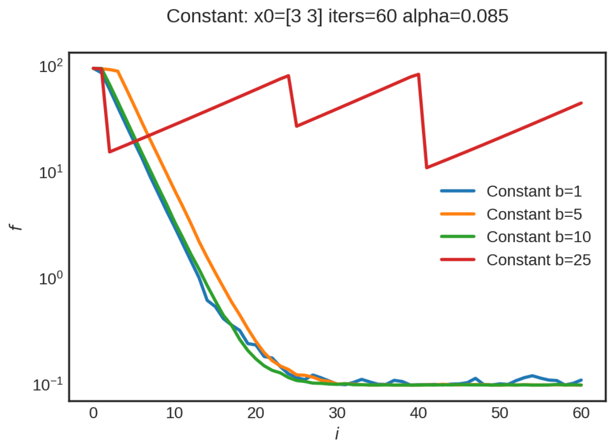

Figure 16

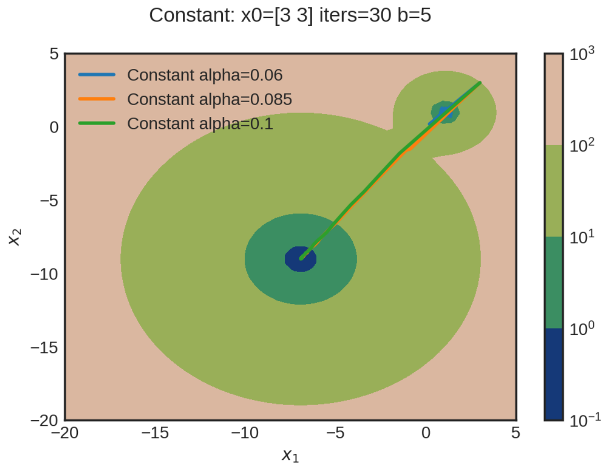

Figure 17

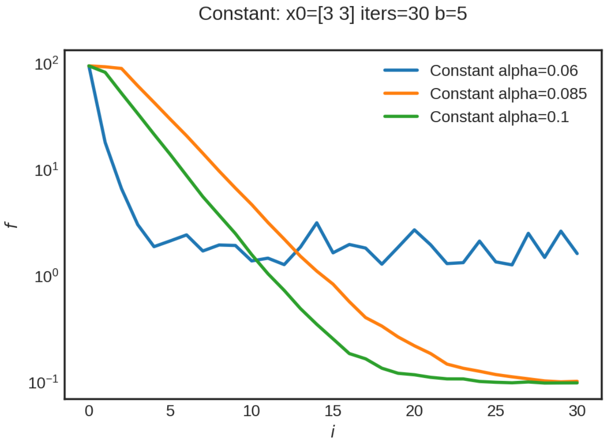

Figure 18

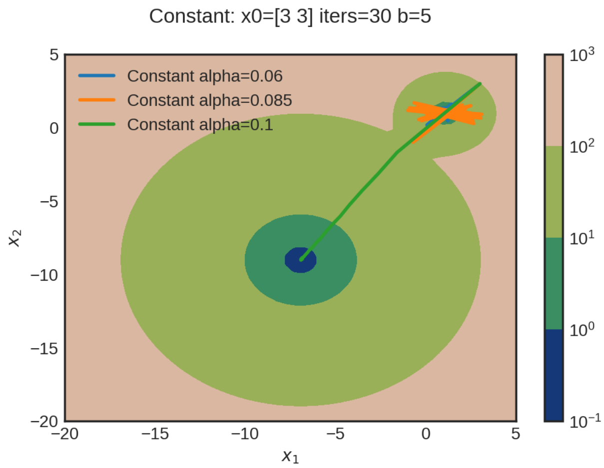

Figure 19

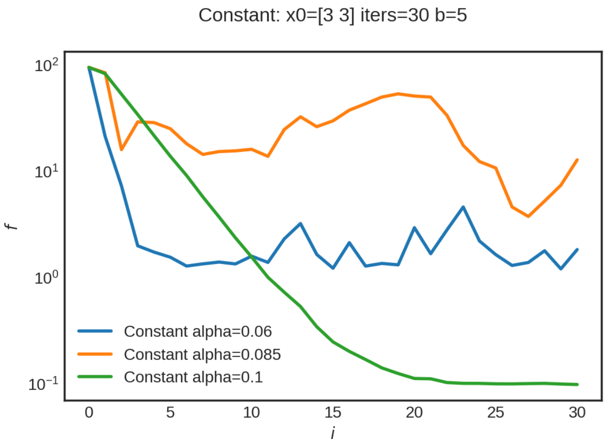

Figure 20

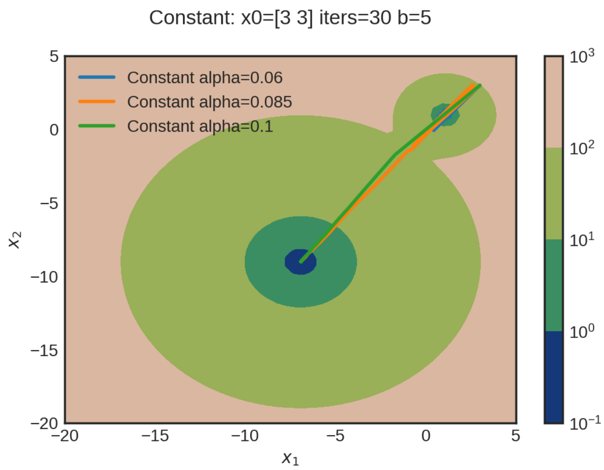

Figure 21

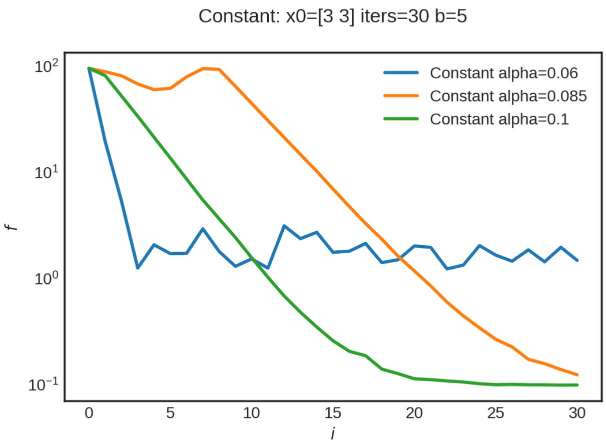

Figure 22

3 (c) Mini-Batch SGD with Different Step Calculations

- Select apropriate step size and explain choice.
- How do these different algorithms affect how f and x change over time.
- How is behaviour affected by choice of mini-batch size.
- Can use constant step size results from (b) as baseline comparison.

3.1 (i) Polyak Step Size

No matter the batch size, the amount of variance between runs on the output is very high on polyak is very high. A lot of times it fails to escape local minimum. Opposite to constant step, polyak runs seemed to have a high variance with higher b. Figs 23 24

3.2 (ii) RMPSProp

Beta and alpha were picked such that b = len(M) would get stuck in the local minimum. Having b 1 and 5 allowed b=1 to escape the local minimum, but RMSProp would run out of steam when it got to going down the big bowl. Figs 25 26

3.3 (iii) Heavy Ball

A highish beta was picked to see the momentum in action and an alpha that caused b = len(M) to just escape the local minimum. Lowering b e.g 1 and 5, in a lot of runs messes up the heavy balls ability to escape the local minimum, i.e it interrupted the momentum of the heavy ball. Heavy ball can be seen jerked around the local minimum due to the noise and sharp changes, in effect, the noise negating the momentum. Figs 27 28

3.4 (iv) Adam

beta1=0.99, and beta2=0.98 is set to accentuate the components of Adam. b can be seen to have quite a negligable effect. Even though for alpha=2, the full data batch size would be almost escaping, adding noise still doesn't allow it to escape, seems like the 2 averaging components of Adam are negating the effects of the noise of the function. We can see that the b=5 and b=25 are going side by side even on greatly different alphas. Figs 29 30

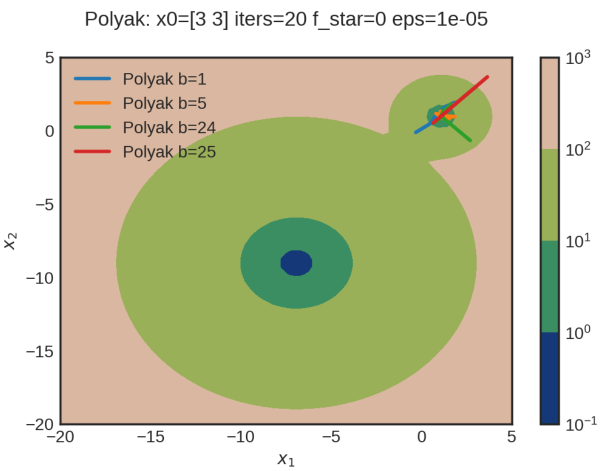

Figure 23

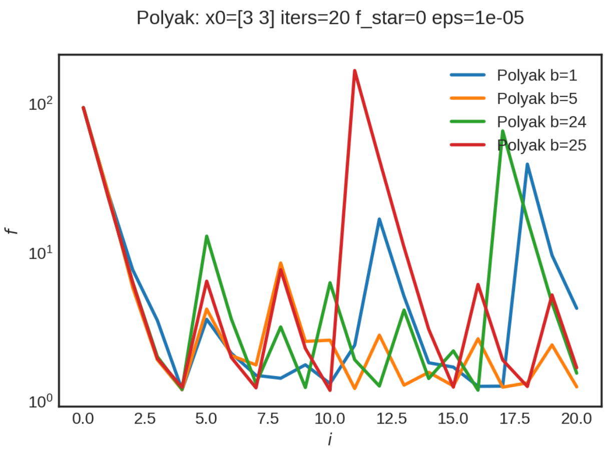

Figure 24

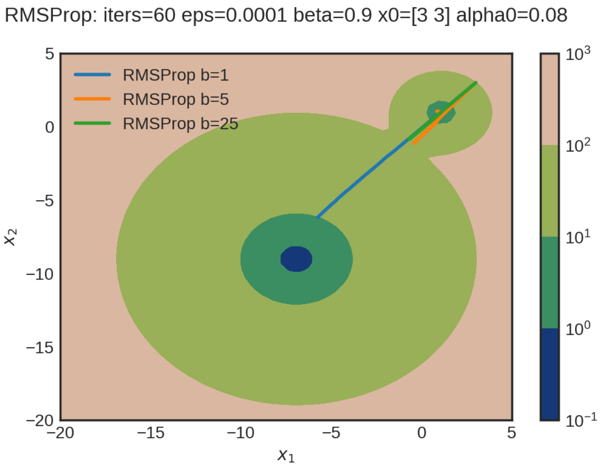

Figure 25

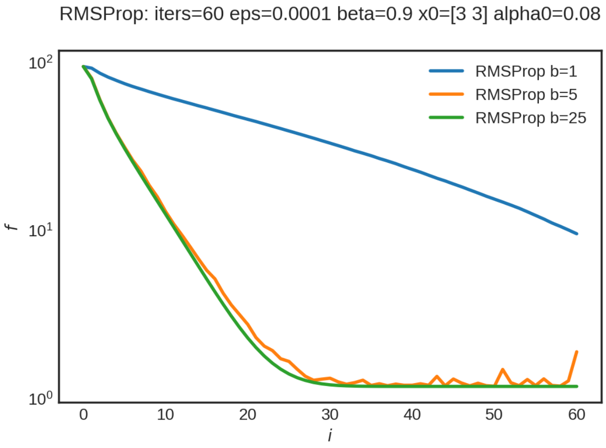

Figure 26

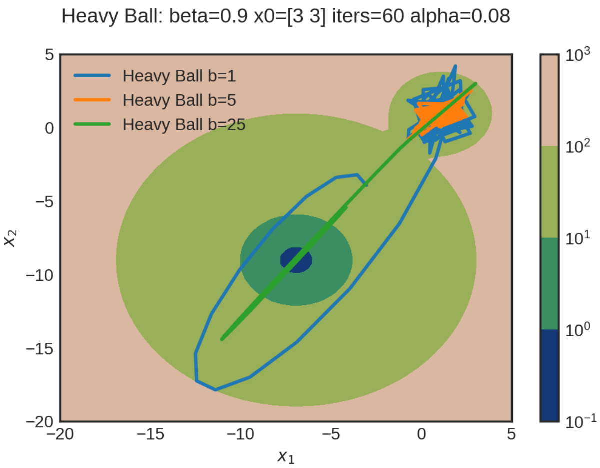

Figure 27

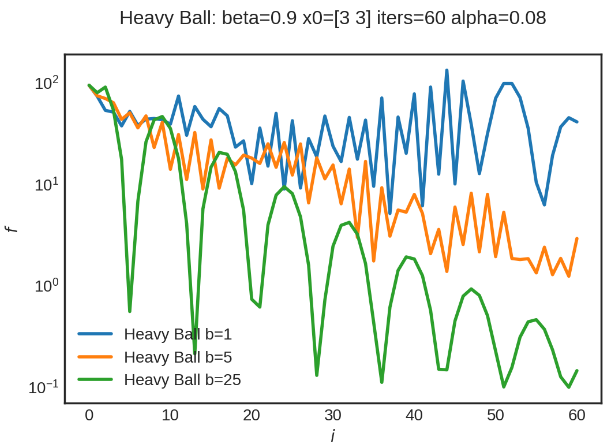

Figure 28

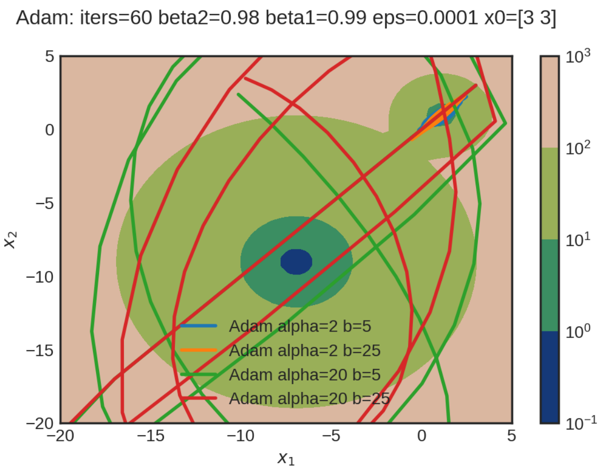

Figure 29

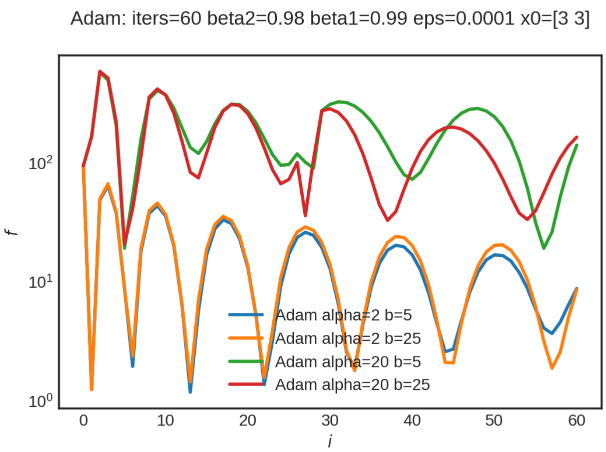

Figure 30

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

import numpy as np

def generate_trainingdata(m=25):
    return np.array([0,0])+0.25*np.random.randn(m,2)

def f(x, minibatch):
    # loss function sum_{w in training data} f(x,w)
    y=0; count=0
    for w in minibatch:
        z=x-w-1
        y=y+min(12*(z[0]**2+z[1]**2), (z[0]+8)**2+(z[1]+10)**2)
        count=count+1
    return y/count

M = generate_trainingdata()

x = np.array([1, 2]); b = 5; m = len(M); alpha=0.4; iters=50
for _ in range(iters):
    N = np.random.choice(np.arange(m), b)
    fN = lambda x: f(x, minibatch=M[N])
    DJ = np.array([finite_diff(fN, x, i) for i in range(len(x))])
    x = x - alpha * DJ

x = np.array([1, 2]); b = 5; m = len(M); alpha=0.4; iters=50
for _ in range(iters):
    np.random.shuffle(D)
    for i in np.arange(0, m, b): # 0 upto m-1, in steps of b. i.e index of each batch start
        N = np.arange(i, i + b)
        fN = lambda x: f(x, minibatch=M[N])
        DJ = np.array([finite_diff(fN, x, i) for i in range(len(x))])
        x = x - alpha * DJ

f = BatchedFunction(f, M, b=5, iters=50) ; fi = iter(f) ; f = f.function;
for fN, dfs in fi:
    step = alpha * np.array([df(*x) for df in dfs])
    x = x - step
```

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

np.arange(0, 10, 10)
np.arange(0, 0+10)

m = len(M) ; b = m ; N = np.arange(b)
fN = lambda x1, x2: f(np.array([x1, x2]), minibatch=M[N])
x1s = np.linspace(-40, 25, 200)
x2s = np.linspace(-40, 20, 200)
X1, X2 = np.meshgrid(x1s, x2s)
Z = np.vectorize(fN)(X1, X2)

from matplotlib.ticker import LogLocator
from matplotlib import cm

plt.contourf(X1, X2, Z,
             locator=LogLocator(),
             cmap= plt.get_cmap('gist_earth'))
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.title(r'Contour Plot')
plt.colorbar();

fig = plt.figure()
ax = plt.axes(projection='3d')
# ax.contour3D(X1, X2, Z, 50, cmap='autumn')
ax.plot_wireframe(X1, X2, Z, cmap=cm.coolwarm, linewidth=0.2)
ax.view_init(10, 80)
ax.set_title('Wireframe')
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')

def finite_diff(f, x, i, delta=0.0001):
    d = np.zeros(len(x)) ; d[i] = delta
    return (f(x) - f(x - d)) / delta

fN = lambda x: f(x, minibatch=M[N])
x = np.array([10, 10])
Dfx1 = finite_diff(fN, x, 0) # w.r.t x1
Dfx2 = finite_diff(fN, x, 1) # w.r.t x2
print(Dfx1)
print(Dfx2)

bf = BatchedFunction(f, M)
o = ConstantStep.set_parameters(x0 = np.array([3,3]),
                                alpha = 0.085,
                                f = bf,
                                iters=60,
                                b = len(M)).run()

ploty(copy.deepcopy(o))

x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

bf = BatchedFunction(f, M)
o = ConstantStep.set_parameters(x0 = np.array([3, 3]),
                                    alpha = 0.085,
                                    f = bf,
                                    iters=60,
                                    b=[5]).run()

o = ConstantStep.run()
ploty(copy.deepcopy(o))

o = ConstantStep.run()
```

```
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

o = ConstantStep.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

o = ConstantStep.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

bf = BatchedFunction(f, M)
o = ConstantStep.set_parameters(x0 = np.array([3, 3]),
                                alpha = 0.085,
                                f = bf,
                                iters=60,
                                b=[1, 5, 10, len(M)]).run()

o = ConstantStep.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

o = ConstantStep.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

o = ConstantStep.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

bf = BatchedFunction(f, M)
o = ConstantStep.set_parameters(x0 = np.array([3, 3]),
                                alpha =[0.05, 0.085, 0.1, 0.5],
                                f = bf,
                                iters=30,
                                b=5).run()

o = ConstantStep.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

bf = BatchedFunction(f, M)
o = Polyak.set_parameters(x0 = np.array([3, 3]),
                          f = bf,
                          iters=60,
                          f_star=0,
                          eps=0.0001,
                          b=5).run()

o = Polyak.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

bf = BatchedFunction(f, M)
o = RMSProp.set_parameters(x0 = np.array([3, 3]),
                           f = bf,
                           iters=60,
                           alpha0=0.085,
                           beta=0.8,
                           eps=0.0001,
```

```
                           b=5).run()

o = RMSProp.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

bf = BatchedFunction(f, M)
o = HeavyBall.set_parameters(x0 = np.array([3, 3]),
                           f = bf,
                           iters=60,
                           alpha=0.085,
                           beta=0.8,
                           b=5).run()

o = HeavyBall.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)

bf = BatchedFunction(f, M)
o = Adam.set_parameters(x0 = np.array([3, 3]),
                        f = bf,
                        iters=60,
                        alpha=10,
                        beta1=0.94,
                        beta2=0.97,
                        eps=0.0001,
                        b=5).run()

o = Adam.run()
x1s = np.linspace(-20, 5, 50)
x2s = np.linspace(-20, 5, 50)
plot_contour(copy.deepcopy(o), x1s, x2s, log=True)
```

```
# 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

from OptimisationAlgorithmToolkit.Function import FunctionIterator

def polyak(x0, f, f_star, eps, iters, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*
    x)]

    for fN, dfs in fi:
        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, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*
    x)]

    for fN, dfs in fi:
        step = alpha * np.array([df(*x) for df in dfs])
        x = x - step
```

```python

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

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

def adagrad(x0, f, alpha0, eps, iters, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    df_vector_sum = np.zeros(len(dfs))
    for fN, dfs in fi:
        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, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    sum = np.zeros(len(x0)) ; alpha = alpha0
    for fN, dfs in fi:
      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, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    z = np.zeros(len(x0))
    for fN, dfs in fi:
        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, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    m = np.zeros(len(x0)) ; v = np.zeros(len(x0)) ; k = 1
    for fN, dfs in fi:
        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**k))
        vhat = (v / (1 - beta2**k))
        x = x - alpha * (mhat / (np.sqrt(vhat) + eps))
        k = k + 1
```

```python

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

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

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.mini_batch_parameters = ('b')
        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:
                if key is not "b":
                    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
```

```python
# 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
# {
#   "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
```

```python

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:
        if p in r:
            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 BatchedFunction:
    def __init__(self, f, M, name="f"):
        self.f = f
        self.function = lambda  x1, x2 : f(np.array([x1,x2]), minibatch=M)
        self.M = M
        self.function_name = name

class FunctionIterator:
    # b = len(M) will behave like normal gradient descent
    def __init__(self, f, b, i):
        self.i = i
        self.f = f
        self.function = f.function
        if type(f) is SymbolicFunction:
            self.batch = False
        else:
            self.batch = True
            self.M = f.M
            self.m = len(self.M)
            if b is None:
                self.b = len(self.M) # act as non stochastic
            else:
                self.b = b
            if self.b == len(self.M):
                self.shuffle = True
            else:
                self.shuffle = True

    def __iter__(self):
        self.epoch = -1
        self.batch_start_indices = iter(())
        return self

    def __next__(self):
        if (self.i <= 0):
            raise StopIteration
        self.i -= 1
        if not self.batch:
            return self.function, f.partial_derivatives

        self.batch_index = next(self.batch_start_indices, None)
        if self.batch_index == None:
```

```
            self.epoch += 1
            if self.shuffle:
                np.random.shuffle(self.M)
            self.batch_start_indices = iter(np.arange(0, (self.m-self.b)+1, self.b))
            self.batch_index = next(self.batch_start_indices, None)

        N = np.arange(self.batch_index, self.batch_index + self.b)
        fN = lambda x: self.f.f(x, minibatch=self.M[N])
        dfs = [(lambda x1, x2, xi=i : finite_diff(fN, np.array([x1, x2]), xi)) for i in range(2)]
        return fN, dfs

class SymbolicFunction:
    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 __iter__(self):
        return self

    def __next__(self):
        return self.function, self.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)


def finite_diff(f, x, i, delta=0.0001):
    d = np.zeros(len(x)) ; d[i] = delta
    return (f(x) - f(x - d)) / delta
```

```
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_box(records, field)

```

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

    f = records[0]['f']

    X1, X2 = np.meshgrid(x1r, x2r)
    Z = np.vectorize(f.function)(X1, X2)
    if log:
        plt.contourf(X1, X2, Z, locator=LogLocator(), cmap=plt.get_cmap('gist_earth')
    )
    else:
        plt.contourf(X1, X2, Z, cmap=plt.get_cmap('gist_earth'))
    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)

    f = records[0]['f']
    function_name = f.function_name
    if sym:
        f_latex = f.latex()
        title = rf'${f_latex}$' + " \n " + title_string(records)
    else:
        title = title_string(records)
    plt.xlabel(r'$x_1$')
    plt.ylabel(r'$x_2$')
    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'$x$')

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:
```

```python
            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, sym=False):
    create_labels(records)
    t = get_titles(records)

    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

    if sym:
        f_latex = f.latex()
        title = rf'${f_latex}$' + " \n " + title_string(records)
    else:
        title = 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
```