

# Optimisation Algorithms - Week 8 Assignment

Ernests Kuznecovs - 17332791 - kuznecoe@tcd.ie

21st March

## Contents

<b>1 (a) Global Random Search</b>	<b>1</b>
1.1 (i) Global Random Search . . . . .	1
1.2 (ii) Global Random Search on $f_1$ and $f_2$ . . . . .	2
1.2.1 $f_1$ . . . . .	3
1.2.2 $f_2$ . . . . .	3
<b>2 (b) Population Based Sampling</b>	<b>3</b>
2.1 (i) Population Based Sampling . . . . .	3
2.1.1 Population Based Sampling . . . . .	3
2.2 (ii) Population Based Sampling Search on $f_1$ and $f_2$ . . . . .	4
2.2.1 $f_1$ . . . . .	4
2.2.2 $f_2$ . . . . .	5
<b>3 (c) Global Random Search to Choose Hyperparameters on Conv Net</b>	<b>5</b>
<b>4 Appendix</b>	<b>6</b>
4.1 Code Listing . . . . .	6

## 1 (a) Global Random Search

### 1.1 (i) Global Random Search

Global random search is defined with input arguments:

- 'intervals' has the type [(float, float)],
  - ith element of the list corresponds to the ith parameter of the function we are optimising.
  - First value of tuple is the minimum value the parameter can take.
  - Second value of tuple is the maximum value the parameter can take.
- 'N' has the type int
  - It's number of times to sample the parameters and run the function with those parameters.
- 'f' has the type function of arity len(intervals)
  - The function that takes in len(intervals) parameters and returns a scalar value, this is the function we are trying to find the minimum value for.

Inside our function, we keep a variable 'lowest' that keep track of what the lowest function value was and the corresponding parameters that achieved the lowest value. Each iteration (N max iterations) we randomly sample parameters for our function within the intervals we specified, then apply those parameters to the function and see if we get a new lowest value.

```
def global_random_search(intervals, N, f):

    # lowest :: (val, [float])
    # fst is the lowest function value achieved
    # snd is the list of parameter values
    lowest = None

    # unzip list of tuples
    l = [l for l, u in intervals]
    u = [u for l, u in intervals]

    # sample and run N times
    for s in range(N):
        r = np.random.uniform(l, u)
        v = f(r)
        if (not lowest) or lowest[0] > v:
            lowest = (v.copy(), r.copy())
    return lowest
```

## 1.2 (ii) Global Random Search on $f_1$ and $f_2$

- $f_1(x_1, x_2) = 3(x_1 - 9)^4 + 5(x_2 - 9)^2$
- $f_2(x, y) = 5|y - 9| + \max(0, x - 9)$

For evaluating function value vs execution time it will be difficult to measure as GRS has quite a lot of randomness. The result changes from run to run on GRS, while for GD it doesn't. It's hard to measure them together because performance is not even comparable for different values of  $x_0$ ,  $\alpha$ , intervals, the hyperparameters are completely different. The nature of the algorithm is completely different.

$\alpha$  and  $x_0$  will be kept the same for both function on GD because the intervals will also be kept the same for GRS.

The number of evaluations on GD is  $j * i$ , where  $j$  is the number of parameters the function to optimise takes, and  $i$  is the number of iterations. The number of evaluations for GRS is  $N$ , the number of times to sample and evaluate the function.

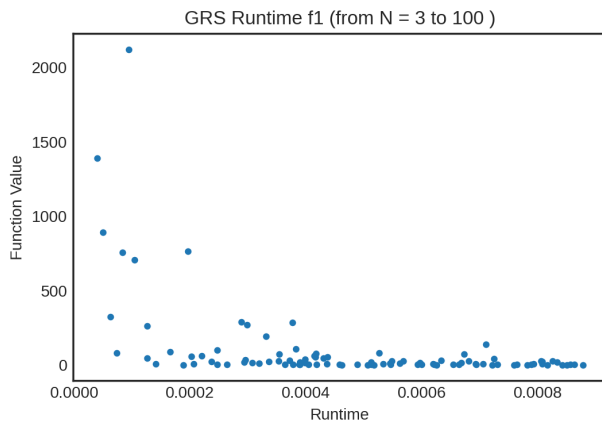


Figure 1

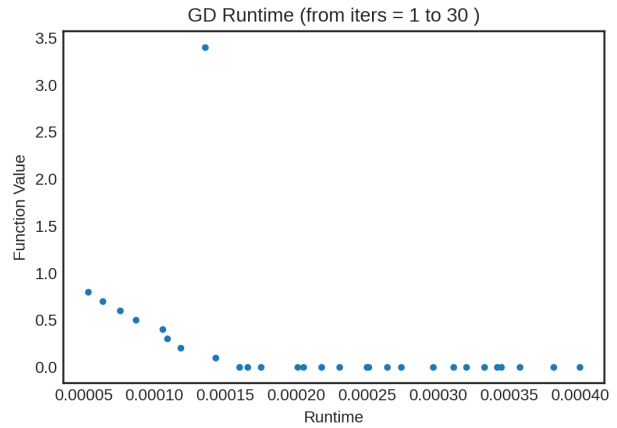


Figure 2

### 1.2.1 $f_1$

On  $f_1$  global random search can handle the steep nature of it, as the slope has no effect algorithm. Whereas on GD, it is very sensitive to initial  $x$ , and cause numerical errors. Though the chance of GRS landing on the minimum can be slim if intervals are too large,  $f_1$  has a relatively small area where the minimum lies at a scale of -10 to 10.

We can see that GD, fig 2 follows a curve, while GRS 1 can get a low number with lower runtime, but there is more variance the less iterations.

### 1.2.2 $f_2$

On  $f_2$ , because there the minimum is not so narrow on a -10,10 scale, GRS can land at function value of 0.5 with low amount of evaluations. GD is very well behaved on this function since it is concave-like, and reaches the minimum a lot in a lot faster than 100 iterations.

## 2 (b) Population Based Sampling

### 2.1 (i) Population Based Sampling

Two version are implemented, the first one with that doesn't grow exponentially in runtime, and one that does. The non exponential one will be presented. Exponential one can be found in the appendix of code under the function name grs2.

#### 2.1.1 Population Based Sampling

Population based sampling chooses  $N$  random points, takes the top  $M$  of them, and then for each of those  $M$  points,  $N$  points are sampled within the neighbourhood, and the top points are taken out of those, and the process is repeated.  $N$  random points are chosen in a region of  $(\frac{1}{M*\epsilon})^c$  of the original interval, where  $c$  is the depth of iteration,  $\epsilon$  is a hyperparameter to scale size of the neighbourhood, and  $M$  is the number of top points selected.

- If we assume, that the best  $M$  points are evenly placed across the interval, then having  $1/M$  of the size of the interval will mean that the sum of the sub intervals will span the range of the whole interval.

In the code, the algorithm first samples  $N$  points, top  $M$  are taken and this initiates the loop for a depth of  $c$ . From the parameter values calculated for an  $M$ , new intervals are centered around the parameter value, decreased and scaled with original range the parameter was initially, and each iteration reduces the neighbourhood. This algorithm does not throw away the  $M$  points when the new  $N$  are computed, the  $M$  points are included in picking the next top  $M$ .

```
def take_top(M, Nresults):
    # Nresults :: [(float, [float])]
    Nresults.sort(key=(lambda x : x[0]))
    return Nresults[0:M].copy()

# each param has a new interval centered around param value
# interval are centered around params
def get_new_intervals_2(params, intervals, M, c):
    # new_intervals :: [(float, float)]
    new_intervals = []
    for i, param_val in enumerate(params):
        l, u = intervals[i]
        interval_range = (u - l)
        offset = ((1/(M*c)) * interval_range) / 2
        new_l = np.clip(param_val-offset, l, u)
        new_h = np.clip(param_val+offset, l, u)
        new_intervals += [(new_l, new_h)]
    return new_intervals

def unzip_intervals(intervals):
    l = [l for l, u in intervals]
    u = [u for l, u in intervals]
```

```

    return l,u

def grs3(intervals, N, M, f, c, eps=1):
    # intervals :: [(l, u)]

    # Nresults :: [(float, [float])]
    # fst is the lowest function value achieved
    # snd is the list of parameter values
    Nresults = []
    l,u = unzip_intervals(intervals)
    for s in range(N):
        r = np.random.uniform(l, u) ; v = f(r)
        Nresults += [(v.copy(), r.copy())]
    # topM :: [(float, [float])]
    topM = take_top(M, Nresults)

    for i in range(c):
        Nresults = []
        for _, param_values in topM:
            l,u = unzip_intervals(get_new_intervals_2(param_values, intervals, M*eps,
            i+1))
            for _ in range(N):
                r = np.random.uniform(l, u) ; v = f(r)
                Nresults += [(v.copy(), r.copy())]
        Nresults += topM
        topM = take_top(M, Nresults)
    return take_top(1, topM)[0]

```

## 2.2 (ii) Population Based Sampling Search on $f_1$ and $f_2$

The number of evaluations of PBS is  $N + (N * M * c)$ .  $N$  samples on before the loop to get initial  $M$ , then inner loop does  $c$  iterations where  $N$  number of points for each  $M$  is taken and evaluated.

GRS and Population Based Sampling (referred in code as GRS3) are tested against each other, parameters are picked such that their evaluations are the same.

### 2.2.1 $f_1$

With unsepcialised parameters on PBS, it does not perform any better than GRS.

```

intervals = [(-10, 10), (-10, 10)]
testGRS3(intervals, N=25, M=2, f=f1, c=2, eps=1, runs=1000)

```

1000 runs of GRS3

Number of f evals: 125

Standard deviation on final function values: 37.52454415835108

Mean on final function values: 5.636444145103109

```

intervals = [(-10, 10), (-10, 10)]
testGRS(intervals, N=125, f=f1, runs=1000)

```

Number of f evals: 125

1000 runs of GRS

Standard deviation on final function values: 15.119353680012136

Mean on final function values: 7.6971497673503855

PBS can be made to pull ahead of GRS significantly at 100 evaluations by choosing  $M$  low and  $N=1$  with  $c=3$  and by bumping up the rate at which region narrows ( $\text{eps}=1.5$ ). These parameters give the PBS behaviour of rapidly narrowing into a single minimum.

```

intervals = [(-10, 10), (-10, 10)]
testGRS3(intervals, N=25, M=1, f=f1, c=3, eps=1.5, runs=1000)

```

1000 runs of GRS3  
 Number of f evals: 100  
 Standard deviation on final function values: 2.949378279695238  
 Mean on final function values: 0.545379791915719

```
intervals = [(-10, 10), (-10, 10)]
testGRS(intervals, N=100, f=f1, runs=1000)
```

Number of f evals: 100  
 1000 runs of GRS  
 Standard deviation on final function values: 24.52660098010104  
 Mean on final function values: 11.890304478212249

### 2.2.2 $f_2$

PBS can be made to pull ahead of GRS at 40 evaluations quite significantly, using a similar rapid narrowing configuration.

```
intervals = [(-10, 10), (-10, 10)]
testGRS3(intervals, N=10, M=1, f=f2, c=3, eps=2, runs=1000)
```

1000 runs of GRS3  
 Number of f evals: 40  
 Standard deviation on final function values: 0.5220113548425356  
 Mean on final function values: 0.30859770356078464

```
intervals = [(-10, 10), (-10, 10)]
testGRS(intervals, N=40, f=f2, runs=1000)
```

Number of f evals: 40  
 1000 runs of GRS  
 Standard deviation on final function values: 1.1866724491447593  
 Mean on final function values: 1.2623655712980348

## 3 (c) Global Random Search to Choose Hyperparameters on Conv Net

Applying random search to choose hyperparameters for conv net. Hyperparams are:

- Mini-batch size:  $b$
- Adam parameters:  $\alpha, \beta_1, \beta_2$
- Number of epochs: epochs

Would be good to discretise the ranges so that there is a smaller space to search.

```
def testParams(alpha, beta1, beta2, batch_size, epochs):
    model = keras.Sequential()
    model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train.shape[1:],
    activation='relu'))
    model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'))
    model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
    model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
    model.add(Dropout(0.5)) ; model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers
    .l1(0.0001)))

    adam = tf.keras.optimizers.Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2,
    name='Adam')
```

```

model.compile(loss="categorical_crossentropy", optimizer=adam, metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
validation_split=0.1)
preds = model.predict(x_test)
cce = tf.keras.losses.CategoricalCrossentropy()
val = cce(y_test, preds).numpy()
return val
t = lambda x: testParams(alpha=x[0], beta1=x[1], beta2=x[2], batch_size=x[3], epochs=
x[4])

```

```

intervals = [
    (0.001, 0.01),          # alpha
    (0.5, 0.999),          # beta1
    (0.5, 0.999),          # beta2
    (4, 256),              # batch_size
    (5, 30),               # epochs
]

```

```
v = grs3(intervals, 20, 1, t, c=3, eps=1.5)
```

```
print(v)
```

```
(1.348901, array([6.09204202e-03, 7.55942386e-01, 7.73003179e-01, 2.49500024e+02,
2.07511524e+01]))
```

With 80 evaluations (just under 1 hour of training on CPU), PBS picked: alpha=0.006, beta1=0.75, beta2=0.77, batchsize=250, epochs=21

```
v = global_random_search(intervals, 80, t)
```

```
print(v)
```

```
(1.3314114, array([3.76845908e-03, 8.97153808e-01, 7.73088738e-01, 1.25796853e+02,
2.81799326e+01]))
```

With 80 evaluations GRS picked: alpha=0.003, beta1=0.89, beta2=0.77, batchsize=125, epochs=28.

GRS achieved slightly lower cross entropy loss. The random nature and temporally expensive procedures are an unfortunate combination for picking hyperparameters.

## 4 Appendix

### 4.1 Code Listing

```

1 import matplotlib as mpl
2 mpl.rcParams['figure.dpi'] = 200
3 mpl.rcParams['figure.facecolor'] = '1'
4 import matplotlib.pyplot as plt
5 plt.style.use('seaborn-white')
6 import copy
7 import numpy as np
8 from sklearn import metrics
9
10 from OptimisationAlgorithmToolkit import Algorithms
11 from OptimisationAlgorithmToolkit import DataType
12 from OptimisationAlgorithmToolkit import Plotting
13 from OptimisationAlgorithmToolkit import Function
14 import importlib
15 importlib.reload(Function)
16 importlib.reload(Algorithms)
17 importlib.reload(DataType)
18 importlib.reload(Plotting)
19 from OptimisationAlgorithmToolkit.Function import BatchedFunction, SymbolicFunction

```

```

20 from OptimisationAlgorithmToolkit.Algorithms import ConstantStep, Polyak, RMSProp,
    HeavyBall, Adam
21 from OptimisationAlgorithmToolkit.DataType import create_labels, get_titles
22 from OptimisationAlgorithmToolkit.Plotting import ploty, plot_contour, plot_path,
    plot_step_size
23
24 from time import perf_counter
25
26 import numpy as np
27 import tensorflow as tf
28 from tensorflow import keras
29 from tensorflow.keras import layers, regularizers
30 from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
31 from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
32 from sklearn.metrics import confusion_matrix, classification_report
33 from sklearn.utils import shuffle
34 import matplotlib.pyplot as plt
35 plt.rc('font', size=18)
36 plt.rcParams['figure.constrained_layout.use'] = True
37 import sys
38
39 # Model / data parameters
40 num_classes = 10
41 input_shape = (32, 32, 3)
42
43 # the data, split between train and test sets
44 (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
45 n=5000
46 x_train = x_train[1:n]; y_train=y_train[1:n]
47 #x_test=x_test[1:500]; y_test=y_test[1:500]
48
49 # Scale images to the [0, 1] range
50 x_train = x_train.astype("float32") / 255
51 x_test = x_test.astype("float32") / 255
52 print("orig x_train shape:", x_train.shape)
53
54 # convert class vectors to binary class matrices
55 y_train = keras.utils.to_categorical(y_train, num_classes)
56 y_test = keras.utils.to_categorical(y_test, num_classes)
57
58 use_saved_model = False
59 if use_saved_model:
60     model = keras.models.load_model("cifar.model")
61 else:
62     model = keras.Sequential()
63     model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train.shape[1:],
64         activation='relu'))
65     model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'))
66     model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
67     model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
68     model.add(Dropout(0.5))
69     model.add(Flatten())
70     model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.
71         l1(0.0001)))
72     model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy
73         "])
74     model.summary()
75
76 batch_size = 128
77 epochs = 20
78 history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
79     validation_split=0.1)
80 model.save("cifar.model")
81 plt.subplot(211)
82 plt.plot(history.history['accuracy'])
83 plt.plot(history.history['val_accuracy'])
84 plt.title('model accuracy')
85 plt.ylabel('accuracy')

```

```

82 plt.xlabel('epoch')
83 plt.legend(['train', 'val'], loc='upper left')
84 plt.subplot(212)
85 plt.plot(history.history['loss'])
86 plt.plot(history.history['val_loss'])
87 plt.title('model loss')
88 plt.ylabel('loss'); plt.xlabel('epoch')
89 plt.legend(['train', 'val'], loc='upper left')
90 plt.show()
91
92 preds = model.predict(x_train)
93 y_pred = np.argmax(preds, axis=1)
94 y_train1 = np.argmax(y_train, axis=1)
95 print(classification_report(y_train1, y_pred))
96 print(confusion_matrix(y_train1, y_pred))
97
98 preds = model.predict(x_test)
99 y_pred = np.argmax(preds, axis=1)
100 y_test1 = np.argmax(y_test, axis=1)
101 print(classification_report(y_test1, y_pred))
102 print(confusion_matrix(y_test1, y_pred))
103
104 from sympy import symbols, Max, Abs
105 x1, x2 = symbols('x1 x2', real=True)
106
107 sym_f1 = 3 * (x1-9)**4 + 5 * (x2-9)**2
108 f1 = SymbolicFunction(sym_f1, [x1, x2], "f_1").function_list_arg
109 f1o = SymbolicFunction(sym_f1, [x1, x2], "f_1")
110
111 sym_f2 = Max(x1-9, 0) + 5 * Abs(x2-9)
112 f2 = SymbolicFunction(sym_f2, [x1, x2], "f_2").function_list_arg
113 f2o = SymbolicFunction(sym_f2, [x1, x2], "f_2")
114
115 def munzip(ll):
116     l = [l for l, u in ll]
117     u = [u for l, u in ll]
118     return l, u
119
120 def myt(lam):
121     ts = []
122     r1 = lam()
123     for i in range(50):
124         t1 = perf_counter(); lam(); t2 = perf_counter()
125         ts += [t2-t1]
126     return (sum(ts)/len(ts), r1)
127
128 x0 = np.array([10, 10])
129 alpha = 0.1
130 f = f1o
131 iters=100
132
133 # print("Final f:", o[0]['Y'][-1])
134 # print("Final xs:", o[0]['X'][-1])
135
136 gdif = lambda i, f: ConstantStep.set_parameters(x0=x0, alpha=alpha, f=f, iters=i).run
137     () [0] ['Y'] [-1]
138
138 i = list(range(1,30))
139 p1 = []
140 for ii in i:
141     p1 += [myt(lambda: gdif(ii, f2o))]
142
143 x,y = munzip(p1)
144 plt.scatter(x,y, s=10)
145
146 plt.xlabel("Runtime")
147 plt.ylabel("Function Value")
148 plt.title("GD Runtime (from iters = 1 to 30)")

```



```

149
150 intervals = [(-10, 10), (-10, 10)]
151
152 grs = lambda N, f: global_random_search(intervals, N, f)[0]
153
154 n = list(range(3,100))
155 p2 = []
156 for N in n:
157     p2 += [myt(lambda: grs(N, f2))]
158
159 # x,y = munzip(p1)
160 # plt.scatter(x,y, s=10, marker='^')
161 x,y = munzip(p2)
162 plt.scatter(x,y, s=10, marker='o')
163
164 plt.xlabel("Runtime")
165 plt.ylabel("Function Value")
166 plt.title("GRS Runtime f1 (from N = 3 to 100 )")
167
168 # x,y = munzip(p1)
169 # plt.scatter(x,y, s=10, marker='^')
170 x,y = munzip(p2)
171 plt.scatter(x,y, s=10, marker='o')
172
173 plt.xlabel("Runtime")
174 plt.ylabel("Function Value")
175 plt.title("GRS Runtime f2 (from N = 3 to 100 )")
176
177 gdif = lambda i, f: ConstantStep.set_parameters(x0=x0, alpha=alpha, f=f, iters=i).run
178     () [0] ['Y'] [-1]
179
180 def global_random_search(intervals, N, f):
181
182     # lowest :: (val, [float])
183     # fst is the lowest function value achieved
184     # snd is the list of parameter values
185     lowest = None
186
187     # unzip list of tuples
188     l = [l for l, u in intervals]
189     u = [u for l, u in intervals]
190
191     # sample and run N times
192     for s in range(N):
193         r = np.random.uniform(l, u)
194         v = f(r)
195         if (not lowest) or lowest[0] > v:
196             lowest = (v.copy(), r.copy())
197     return lowest
198
199 a = [1, 2, 3]
200 b = [4, 5, 6]
201 c = np.random.uniform(a, b)
202 print(c)
203
204 def testGRS(intervals, N, f, runs):
205     r = []
206     for i in range(runs):
207         r += [global_random_search(intervals, N, f)[0]]
208
209     print("Number of f evals:", N)
210     print(runs, "runs of GRS")
211     print("Standard deviation on final function values: ", np.std(r))
212     print("Mean on final function values: ", np.mean(r))
213
214 def take_top(M, Nresults):
215     # Nresults :: [(float, [float])]
216     Nresults.sort(key=(lambda x : x[0]))

```

```

216     return Nresults[0:M].copy()
217
218 # each param has a new interval centered around param value
219 # interval are centered around params
220 def get_new_intervals_2(params, intervals, M, c):
221     # new_intervals :: [(float, float)]
222     new_intervals = []
223     for i, param_val in enumerate(params):
224         l, u = intervals[i]
225         interval_range = (u - l)
226         offset = ((1/(M*c)) * interval_range) / 2
227         new_l = np.clip(param_val-offset, l, u)
228         new_h = np.clip(param_val+offset, l, u)
229         new_intervals += [(new_l, new_h)]
230     return new_intervals
231
232 def unzip_intervals(intervals):
233     l = [l for l, u in intervals]
234     u = [u for l, u in intervals]
235     return l,u
236
237 def grs3(intervals, N, M, f, c, eps=1):
238     # intervals :: [(l, u)]
239
240     # Nresults :: [(float, [float])]
241     # fst is the lowest function value achieved
242     # snd is the list of parameter values
243     Nresults = []
244     l,u = unzip_intervals(intervals)
245     for s in range(N):
246         r = np.random.uniform(l, u) ; v = f(r)
247         Nresults += [(v.copy(), r.copy())]
248     # topM :: [(float, [float])]
249     topM = take_top(M, Nresults)
250
251     for i in range(c):
252         Nresults = []
253         for _, param_values in topM:
254             l,u = unzip_intervals(get_new_intervals_2(param_values, intervals, M*eps,
255             i+1))
256             for _ in range(N):
257                 r = np.random.uniform(l, u) ; v = f(r)
258                 Nresults += [(v.copy(), r.copy())]
259             Nresults += topM
260             topM = take_top(M, Nresults)
261         return take_top(1, topM)[0]
262
263 # each param has a new interval centered around param value
264 # interval will be at a range of 1/M
265 def get_new_intervals(params, intervals, M):
266     # new_intervals :: [(float, float)]
267     new_intervals = []
268     for i, param_val in enumerate(params):
269         l, u = intervals[i]
270         interval_range = (u - l)
271         offset = ((1/M) * interval_range) / 2
272         new_l = np.clip(param_val-offset, l, u)
273         new_h = np.clip(param_val+offset, l, u)
274         new_intervals += [(new_l, new_h)]
275     return new_intervals
276
277 # global_random_search_2 returns (float, [float])
278 # fst is the lowest function value achieved
279 # snd is the list of parameter values
280 def grs2(intervals, N, M, f, c, eps):
281     # intervals :: [(l, u)]
282
283     # Nresults :: [(float, [float])]

```

```

283     # fst is the lowest function value achieved
284     # snd is the list of parameter values
285     Nresults = []
286
287     # unzip list of tuples
288     l = [l for l, u in intervals]
289     u = [u for l, u in intervals]
290
291     # sample and run N times
292     for s in range(N):
293         r = np.random.uniform(l, u)
294         v = f(r)
295         Nresults += [(v.copy(), r.copy())]
296
297     # topM :: [(float, [float])]
298     topM = take_top(M, Nresults)
299     if (c-1 == 0):
300         return topM[0]
301
302     # when c = 0 do the pulse
303
304     # collect the top results from applying grs to topM
305     # top_params :: [(float, [float])]
306     top_params = []
307     for (_, params) in topM:
308         new_intervals = get_new_intervals(params, intervals, M/eps)
309         top_params += [global_random_search_2(new_intervals, N, M, f, c-1, eps)]
310
311     return take_top(1, top_params)[0]
312
313 intervals = [(-10, 10), (-10, 10)]
314 testGRS3(intervals, N=25, M=2, f=f1, c=2, eps=1, runs=1000)
315
316 intervals = [(-10, 10), (-10, 10)]
317 testGRS(intervals, N=125, f=f1, runs=1000)
318
319 intervals = [(-10, 10), (-10, 10)]
320 testGRS3(intervals, N=25, M=1, f=f1, c=3, eps=1.5, runs=1000)
321
322 intervals = [(-10, 10), (-10, 10)]
323 testGRS(intervals, N=100, f=f1, runs=1000)
324
325 intervals = [(-10, 10), (-10, 10)]
326 testGRS3(intervals, N=10, M=1, f=f2, c=3, eps=2, runs=1000)
327
328 intervals = [(-10, 10), (-10, 10)]
329 testGRS(intervals, N=40, f=f2, runs=1000)
330
331 def testGRS3(intervals, N, M, f, c, eps, runs):
332     r = []
333     for i in range(runs):
334         r += [grs3(intervals, N, M, f, c, eps)[0]]
335
336     print(runs, "runs of GRS3")
337     print("Number of f evals:", (N + (N*M*c)))
338     print("Standard deviation on final function values: ", np.std(r))
339     print("Mean on final function values: ", np.mean(r))
340
341 intervals = [(-100, 100), (-100, 100)]
342 f = global_random_search_2(intervals, 100, 3, f1, 4, 1)
343 print(f)
344
345 intervals = [(-100, 100), (-100, 100)]
346 %timeit f = global_random_search_2(intervals, 100, 3, f2, 4, 1)
347 print(f)
348
349 intervals = [(-100, 100), (-100, 100)]
350 f = grs3(intervals, 100, 3, f2, 4, 1)

```

```

351 print(f)
352
353 intervals = [(-100, 100), (-100, 100)]
354 %timeit f = grs3(intervals, 100, 3, f1, 4, 1)
355 print(f)
356
357 def testParams(alpha, beta1, beta2, batch_size, epochs):
358     model = keras.Sequential()
359     model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train.shape[1:],
360         activation='relu'))
361     model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'))
362     model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
363     model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
364     model.add(Dropout(0.5)) ; model.add(Flatten())
365     model.add(Dense(num_classes, activation='softmax', kernel_regularizer=regularizers
366         .l1(0.0001)))
367
368     adam = tf.keras.optimizers.Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2,
369         name='Adam')
370     model.compile(loss="categorical_crossentropy", optimizer=adam, metrics=["accuracy
371         "])
372
373     model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
374         validation_split=0.1)
375     preds = model.predict(x_test)
376     cce = tf.keras.losses.CategoricalCrossentropy()
377     val = cce(y_test, preds).numpy()
378     return val
379
380 t = lambda x: testParams(alpha=x[0], beta1=x[1], beta2=x[2], batch_size=x[3], epochs=
381     x[4])
382
383 intervals = [
384     (0.001, 0.01),          # alpha
385     (0.5, 0.999),          # beta1
386     (0.5, 0.999),          # beta2
387     (4, 256),              # batch_size
388     (5, 30),               # epochs
389 ]
390
391 v = grs3(intervals, 20, 1, t, c=3, eps=1.5)
392
393 print(v)
394
395 v = global_random_search(intervals, 80, t)
396
397 print(v)
398
399 import numpy as np
400 import tensorflow as tf
401 from tensorflow import keras
402 from tensorflow.keras import layers, regularizers
403 from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
404 from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
405 from sklearn.metrics import confusion_matrix, classification_report
406 from sklearn.utils import shuffle
407 import matplotlib.pyplot as plt
408 plt.rc('font', size=18)
409 plt.rcParams['figure.constrained_layout.use'] = True
410 import sys
411
412 # Model / data parameters
413 num_classes = 10
414 input_shape = (32, 32, 3)
415
416 # the data, split between train and test sets
417 (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
418 n=5000
419 x_train = x_train[1:n]; y_train=y_train[1:n]

```

```

413 #x_test=x_test[1:500]; y_test=y_test[1:500]
414
415 # Scale images to the [0, 1] range
416 x_train = x_train.astype("float32") / 255
417 x_test = x_test.astype("float32") / 255
418 print("orig x_train shape:", x_train.shape)
419
420 # convert class vectors to binary class matrices
421 y_train = keras.utils.to_categorical(y_train, num_classes)
422 y_test = keras.utils.to_categorical(y_test, num_classes)

1 # Algorithms.py
2
3 # Algorithms implement a similar interface:
4 # - specific names on input arguments
5 # - accesses function related things through the OptimisableFunction class
6 # - needs to return X, Y
7
8 import numpy as np
9
10
11 from OptimisationAlgorithmToolkit.Function import FunctionIterator
12
13 class OptimisationAlgorithm:
14     def __init__(self, algorithm, algorithm_name):
15         self.algorithm = algorithm
16         self.algorithm_name = algorithm_name
17
18         arguments = algorithm.__code__.co_varnames[:algorithm.__code__.co_argcount]
19         self.mini_batch_parameters = ('b')
20         self.all_parameters = arguments
21         self.standard_parameters = ("x0", "f", "iters")
22         self.hyperparameters = list(filter(lambda arg: arg not in self.
standard_parameters, arguments))
23
24     def __type_check_parameters(self, input_record):
25         for key in input_record.keys():
26             if key not in self.all_parameters:
27                 raise NameError(key + " is not one of: " + str(self.all_parameters))
28         for key in self.all_parameters:
29             if key not in input_record:
30                 if key is not "b":
31                     raise NameError(key + " is missing from input: " + str(list(
input_record.keys()))))
32
33     def set_parameters(self, **input_record):
34         self.__type_check_parameters(input_record)
35         self.parameter_values = input_record
36         return self
37
38     def run(self):
39         inputs = self.__make_input()
40         for input in inputs:
41             input["X"], input["Y"] = self.algorithm(**input)
42             input["X"] = np.array(input["X"])
43             input["Y"] = np.array(input["Y"])
44             input["algorithm"] = self
45         return inputs
46
47     def __make_input(self):
48         kwargs = self.parameter_values.copy()
49         expected_vector = { "x0" }
50         for key, value in kwargs.items():
51             if key in expected_vector:
52                 value = np.array(value)
53                 if value.ndim == 1:
54                     kwargs[key] = [value]
55             else:
56                 if type(value) is not list:

```

```

57         kwargs[key] = [value]
58
59     keys = kwargs.keys()
60     partial_dicts = [{}]
61     for key in keys:
62         partial_dicts_new = []
63         for partial_dict in partial_dicts:
64             for value in kwargs[key]: # making a new partial dict for each value
65                 partial_dict_new = partial_dict.copy()
66                 partial_dict_new[key] = value
67                 partial_dicts_new += [partial_dict_new]
68         partial_dicts = partial_dicts_new
69     return partial_dicts
70
71
72
73 def polyak(x0, f, f_star, eps, iters, b=None):
74     fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*
x)]
75
76     for fN, dfs in fi:
77         fdif = f(*x) - f_star
78         df_squared_sum = np.sum(np.array([df(*x)**2 for df in dfs]))
79         alpha = fdif / (df_squared_sum + eps)
80         x = x - alpha * np.array([df(*x) for df in dfs])
81
82         X += [x] ; Y += [f(*x)]
83     return X, Y
84
85 Polyak = OptimisationAlgorithm(algorithm=polyak,
86                                algorithm_name="Polyak")
87
88 def constant_step(x0, alpha, f, iters, b=None):
89     fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*
x)]
90
91     for fN, dfs in fi:
92         step = alpha * np.array([df(*x) for df in dfs])
93         x = x - step
94
95         X += [x] ; Y += [f(*x)]
96     return X, Y
97
98 ConstantStep = OptimisationAlgorithm(algorithm=constant_step,
99                                      algorithm_name="Constant")
100
101 def adagrad(x0, f, alpha0, eps, iters, b=None):
102     fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*
x)]
103
104     df_vector_sum = np.zeros(len(dfs))
105     for fN, dfs in fi:
106         df_vec = np.array([df(*x) for df in dfs])
107         df_vector_sum += df_vec**2
108         alphas = alpha0 / (np.sqrt(df_vector_sum) + eps)
109         x = x - (alphas * df_vec)
110
111         X += [x] ; Y += [f(*x)]
112     return X, Y
113
114 Adagrad = OptimisationAlgorithm(algorithm=adagrad,
115                                algorithm_name="Adagrad")
116
117 def rmsprop(x0, f, alpha0, beta, eps, iters, b=None):
118     fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*
x)]
119
120     sum = np.zeros(len(x0)) ; alpha = alpha0

```

```

121     for fN, dfs in fi:
122         x = x - (alpha * np.array([df(*x) for df in dfs]))
123         sum = beta * sum + (1 - beta) * np.array([df(*x)**2 for df in dfs])
124         alpha = alpha0 / (np.sqrt(sum) + eps)
125
126     X += [x] ; Y += [f(*x)]
127     return X, Y
128
129 RMSProp = OptimisationAlgorithm(algorithm=rmsprop,
130                                 algorithm_name="RMSProp")
131
132
133 def heavy_ball(x0, f, alpha, beta, iters, b=None):
134     fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*
135     x)]
136
137     z = np.zeros(len(x0))
138     for fN, dfs in fi:
139         z = beta * z + alpha * np.array([df(*x) for df in dfs])
140         x = x - z
141
142     X += [x] ; Y += [f(*x)]
143     return X, Y
144
145 HeavyBall = OptimisationAlgorithm(algorithm=heavy_ball,
146                                   algorithm_name="Heavy Ball")
147
148 def adam(x0, f, eps, beta1, beta2, alpha, iters, b=None):
149     fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*
150     x)]
151
152     m = np.zeros(len(x0)) ; v = np.zeros(len(x0)) ; k = 1
153     for fN, dfs in fi:
154         m = beta1 * m + (1 - beta1) * np.array([df(*x) for df in dfs])
155         v = beta2 * v + (1 - beta2) * np.array([(df(*x)**2) for df in dfs])
156         mhat = (m / (1 - beta1**k))
157         vhat = (v / (1 - beta2**k))
158         x = x - alpha * (mhat / (np.sqrt(vhat) + eps))
159         k = k + 1
160
161     X += [x] ; Y += [f(*x)]
162     return X, Y
163
164 Adam = OptimisationAlgorithm(algorithm=adam,
165                              algorithm_name="Adam")

```

```

1  # Each record should contain its label depending on what are the other records in the
2  list.
3
4  # The user semi-manually inputs what the title should be.
5
6  # - Have utility functions to extract pieces of the title from the list of records.
7
8  # Function that takes in a list of records.
9
10 # - For each record determines the label based on what is in the list of records.
11
12
13 # Perhaps there should be a function that calculatesthe meta information that is used
14 by both
15
16 # - utility functions that extract peieces of title
17
18 # - function that assigns the labels to each individual record
19
20
21
22 # MetaInfo: extracts:
23
24 # - Which optimisaiton functions there area
25
26 # - For each optimisation function
27
28 # - What are the parameters that are not varying and what values do they have
29
30 # - What are the parameters that are varying and what values do they have

```

```

22
23 # {
24 #     ...
25 #     ...
26 #     label:
27 # }
28 # label made up from what uniquely identifies it
29 # - first is optimisation algorithm itself
30 # - second are the hyperparameters that uniquely identifies the cluster of algorithms
31 #     - RMSProp alpha0=0.4
32 #     - RMSProp alpha0=0.5
33 #     - Adam     beta1=0.2  beta2=0.4
34 #     - Adam     beta1=0.3  beta2=0.5
35
36 # - Then would like to extract the common descriptive pieces
37 #     - Different common pieces per algorithm used
38 #         - Records -> AlgorithmName -> CommonThingsString
39 #         - Adam: beta1=0.1 eps=0.0001 iters=50 x0=[1, 1]
40 #         - RMSProp: eps=0.0001 iters=50 x0=[1, 1]
41
42
43 # MetaRecord extracts
44 # - Algorithms and their corresponding Varying fields
45 # {
46 #     "Adam"      : ["eps", "beta1"]
47 #     "RMSProp"   : ["eps", "alpha0"]
48 # }
49
50
51 # meta_record = meta(inputs)
52 # inputs = create_labels(meta_record, inputs)
53 # inputs = get_title(meta_record, inputs)
54
55 # get_titles returns
56 # {
57 #     "Adam" : "Adam: beta1=0.1 eps=0.0001 iters=50 x0=[1, 1]",
58 #     "RMSProp" : "RMSProp: eps=0.0001 iters=50 x0=[1, 1]"
59 # }
60
61 import numpy as np
62
63 def get_titles(records):
64     m = meta(records)
65     t = {}
66     for alg_name in m.keys():
67         t[alg_name] = get_title(alg_name, records, m)
68     return t
69
70 def get_title(alg_name, records, meta):
71     title = f'{alg_name}:'
72     algs = alg(records, alg_name)
73
74     r = algs[0]
75     params = set(r["algorithm"].all_parameters)
76     varied = meta[alg_name]
77     params.remove('f')
78     params = params - varied
79
80     for p in params:
81         if p in r:
82             title += f' {p}={r[p]}'
83     return title
84
85 def create_labels(records):
86     m = meta(records)
87     for r in records:
88         r['label'] = create_label(r, m)
89

```



```

90 # e.g: Adam      beta1=0.2  beta2=0.4
91 def create_label(record, meta):
92     alg_name = record['algorithm'].algorithm_name
93     differing_fields = meta[alg_name]
94     label = f'{alg_name}'
95     for f in differing_fields:
96         label += f' {f}={record[f]}'
97     return label
98
99 # {
100 #   "Adam"      : ["eps", "beta1"]
101 #   "RMSProp"   : ["eps", "alpha0"]
102 # }
103 def meta(records):
104     mr = {}
105     algs = get_algs(records)
106     for a in algs:
107         a_records = alg(records, a)
108         mr[a] = differing_fields(a_records)
109     return mr
110
111 def differing_fields(records):
112     diff_fields = set({})
113     t = records[0]
114     for r in records:
115         for key, value in r.items():
116             # print("a")
117             # print(t[key])
118             # print(type(value))
119             # print(isinstance(value, list))
120
121             if isinstance(value, list):
122                 value = np.array(value)
123             if isinstance(t[key], list):
124                 t[key] = np.array(t[key])
125
126             b = t[key] == value
127             # print(b)
128             # print(type(b))
129             if type(b) == np.ndarray:
130                 b = b.all()
131             if not (b):
132                 diff_fields.add(key)
133
134     diff_fields.discard('X')
135     diff_fields.discard('Y')
136     return diff_fields
137
138 # extract one algorithm type, filter out the rest
139 def alg(records, algorithm_name):
140     return list(filter(lambda r: r['algorithm'].algorithm_name == algorithm_name,
141                       records))
142
143 # gets algorithms names in the records
144 def get_algs(records):
145     algs = set({})
146     for r in records:
147         algs.add(r['algorithm'].algorithm_name)
148     return algs
149
150 # wonder how this would look in haskell
151 # functional operators and stuff, would it make it easier.

```

```

1 # Functions that will be optimised:
2 # - Allows access to
3 #   - Partial Derivatives
4 #   - String representation of the function (latex)

```

```

5 # - Constructor uses sympy to obtain the above
6
7 from sympy import simplify, latex, lambdify
8 import numpy as np
9
10 class BatchedFunction:
11     def __init__(self, f, M, name="f"):
12         self.f = f
13         self.function = lambda x1, x2 : f(np.array([x1,x2]), minibatch=M)
14         self.M = M
15         self.function_name = name
16
17 class FunctionIterator:
18     # b = len(M) will behave like normal gradient descent
19     def __init__(self, f, b, i):
20         self.i = i
21         self.f = f
22         self.function = f.function
23         if type(f) is SymbolicFunction:
24             self.batch = False
25         else:
26             self.batch = True
27             self.M = f.M
28             self.m = len(self.M)
29             if b is None:
30                 self.b = len(self.M) # act as non stochastic
31             else:
32                 self.b = b
33             if self.b == len(self.M):
34                 self.shuffle = True
35             else:
36                 self.shuffle = True
37
38     def __iter__(self):
39         self.epoch = -1
40         self.batch_start_indices = iter(())
41         return self
42
43     def __next__(self):
44         if (self.i <= 0):
45             raise StopIteration
46         self.i -= 1
47         if not self.batch:
48             return self.function, self.f.partial_derivatives
49
50         self.batch_index = next(self.batch_start_indices, None)
51         if self.batch_index == None:
52             self.epoch += 1
53             if self.shuffle:
54                 np.random.shuffle(self.M)
55             self.batch_start_indices = iter(np.arange(0, (self.m-self.b)+1, self.b))
56             self.batch_index = next(self.batch_start_indices, None)
57
58         N = np.arange(self.batch_index, self.batch_index + self.b)
59         fN = lambda x: self.f.f(x, minibatch=self.M[N])
60         dfs = [(lambda x1, x2, xi=i : finite_diff(fN, np.array([x1, x2]), xi)) for i
61 in range(2)]
62         return fN, dfs
63
64 class SymbolicFunction:
65     def __init__(self, sympy_function, sympy_symbols, function_name):
66         self.sympy_symbols = sympy_symbols
67         self.function_name = function_name
68
69         self.sympy_function = sympy_function
70         self.function = lambdify(sympy_symbols, sympy_function, modules="numpy")
71         self.function_list_arg = lambda x: self.function(x[0], x[1])

```

```

72     self.sympy_partial_derivatives = [sympy_function.diff(symbol) for symbol in
sympy_symbols]
73     self.partial_derivatives = [lambdify(sympy_symbols, p, modules="numpy") for p
in self.sympy_partial_derivatives]
74
75     def __iter__(self):
76         return self
77
78     def __next__(self):
79         return self.function, self.partial_derivatives
80
81     def __parameters_string(self):
82         s = map(latex, self.sympy_symbols)
83         return ",".join(s)
84
85     def latex(self):
86         return self.function_name + "(" + self.__parameters_string() + ") = " + latex
(simplify(self.sympy_function))
87
88     def partials_latex(self):
89         s = map(latex, self.sympy_symbols)
90         z = zip(self.sympy_partial_derivatives, s)
91         return [ "\\frac{\\partial " + self.function_name + "{\\partial " +
partial_wrt_name + "}" "=" + latex(simplify(partial))
92                 for (partial, partial_wrt_name) in z]
93
94     def print_partials_latex(self):
95         for p in self.partials_latex():
96             print(p)
97
98
99 def finite_diff(f, x, i, delta=0.0001):
100     d = np.zeros(len(x)) ; d[i] = delta
101     return (f(x) - f(x - d)) / delta

```

```

1 import matplotlib as mpl
2 mpl.rcParams['figure.dpi'] = 200
3 mpl.rcParams['figure.facecolor'] = '1'
4 import matplotlib.pyplot as plt
5 plt.style.use('seaborn-white')
6
7 from OptimisationAlgorithmToolkit.DataType import create_labels, get_titles
8
9 from matplotlib.ticker import LogLocator
10
11 import numpy as np
12
13
14 def plot_contour(records, x1r, x2r, log=False, sym=False):
15     create_labels(records)
16     t = get_titles(records)
17
18     f = records[0]['f']
19
20     X1, X2 = np.meshgrid(x1r, x2r)
21     Z = np.vectorize(f.function)(X1, X2)
22     if log:
23         plt.contourf(X1, X2, Z, locator=LogLocator(), cmap=plt.get_cmap('gist_earth'))
24     else:
25         plt.contourf(X1, X2, Z, cmap=plt.get_cmap('gist_earth'))
26     xlim = plt.xlim()
27     ylim = plt.ylim()
28     for (X, label) in dicts_collect(("X", "label"), records):
29         plt.plot(X.T[0], X.T[1], linewidth=2.0, label=label)
30
31     f = records[0]['f']
32     function_name = f.function_name
33     if sym:

```

```

34         f_latex = f.latex()
35         title = rf'${f_latex}$' + " \n " + title_string(records)
36     else:
37         title = title_string(records)
38     plt.xlabel(r'$x_1$')
39     plt.ylabel(r'$x_2$')
40     plt.title(title)
41
42
43     plt.xlim(xlim)
44     plt.ylim(ylim)
45     plt.legend()
46     plt.colorbar()
47
48 def plot_path(records, xr):
49     create_labels(records)
50     f = records[0]['f'].function;
51     function_name = records[0]['f'].function_name
52     f_latex = records[0]['f'].latex()
53
54     yr = [f(x) for x in xr]
55     plt.plot(xr, yr)
56     xlim = plt.xlim()
57     ylim = plt.ylim()
58
59     for (X, label) in dicts_collect(("X", "label"), records):
60         xs = X.flatten()
61         ys = [f(x) for x in xs]
62         plt.plot(xs, ys, linewidth=2.0, label=label)
63
64     plt.xlim(xlim)
65     plt.ylim(ylim)
66     plt.legend()
67     title = rf'${f_latex}$' + "\n" + title_string(records)
68     plt.title(title)
69     plt.ylabel(f'${function_name}$')
70     plt.xlabel(r'$x$')
71
72 def plot_step_size(records, mean=True):
73     create_labels(records)
74     fig, ax = plt.subplots()
75     f_latex = records[0]['f'].latex()
76     for (X, label) in dicts_collect(("X", "label"), records):
77         if mean:
78             s = np.array([np.mean(x) for x in step_sizes(X).T])
79             ax.plot(np.arange(1, len(s)+1), s, linewidth=2.0, label=label)
80         else:
81             sX = step_sizes(X)
82             for i in range(len(sX)):
83                 x = i + 1
84                 s = sX[i]
85             ax.plot(np.arange(1, len(s)+1), s, linewidth=2.0, label=label + f'
86 $x_{x}$ step$')
87         ax.legend()
88
89     title = rf'${f_latex}$' + " \n " + title_string(records)
90     if mean:
91         ax.set_title("Mean Step Across x's \n" + title)
92     else:
93         ax.set_title("Mean Step Across x's \n" + title)
94     ax.set_ylabel(f'Step Size')
95     ax.set_xlabel(r'$i$')
96
97 def title_string(records):
98     title = ""
99     t = get_titles(records)
100     for _, v in t.items():

```

```

101         title += v + '\n'
102     return title
103
104 # [[x11 x21 x31 ...] [x12 x22 x32 ...] ...] -> [[x12-x11 x13-x12 ...] [x22-x21 x23-
105 x22 ...] ...]
106 def step_sizes(X):
107     return np.array([(x[1:] - x[:-1]) for x in X.T])
108
109
110 def ploty(records, sym=False):
111     create_labels(records)
112     t = get_titles(records)
113
114     fig, ax = plt.subplots()
115     for (X, Y, label) in dicts_collect(("X", "Y", "label"), records):
116         ax.plot(range(len(Y)), Y, linewidth=2.0, label=label)
117
118     f = records[0]['f']
119     function_name = f.function_name
120
121     if sym:
122         f_latex = f.latex()
123         title = rf'${f_latex}$' + " \n " + title_string(records)
124     else:
125         title = title_string(records)
126
127     ax.set_title(title)
128     ax.set_ylabel(f'${function_name}$')
129     ax.set_xlabel(r'$i$')
130
131     ax.legend()
132     return ax
133
134 def dicts_collect(keys, dicts):
135     values = []
136     for dict in dicts:
137         values += [[dict[key] for key in keys]]
138     return values
139

```