

# Optimisation Algorithms - Week 2 Assignment

Ernests Kuznecovs - 17332791 - kuznecoe@tcd.ie

16th February 2022

## Contents

<b>1</b>	<b>(a) Derivatives and Finite Difference for <math>y(x) = x^4</math></b>	<b>1</b>
1.1	(i) Symbolic Derivative . . . . .	1
1.2	(ii) Finite Difference Implementation . . . . .	1
1.3	(iii) Varying $\delta$ on Finite Difference . . . . .	2
<b>2</b>	<b>(b) Gradient Descent Optimisation Algorithm</b>	<b>2</b>
2.1	(i) Gradient Descent Implementation . . . . .	2
2.2	(ii) Visualising Gradient Descent . . . . .	3
2.3	(iii) Varying Step Size $\alpha$ and $x_0$ . . . . .	5
<b>3</b>	<b>(c) Optimising <math>y(x) = \gamma x^2</math> and <math>y(x) = \gamma x </math></b>	<b>5</b>
3.1	(i) Optimising $y(x) = \gamma x^2$ . . . . .	5
3.2	(ii) Optimising $y(x) = \gamma x $ . . . . .	7
<b>4</b>	<b>Appendix</b>	<b>7</b>
4.1	Code Listing . . . . .	7

## 1 (a) Derivatives and Finite Difference for $y(x) = x^4$

### 1.1 (i) Symbolic Derivative

Using the symbolic maths library sympy, a symbol object  $x$  is created,  $x \in \mathbb{R}$ . Then the `**4` operator is applied to the object, now it becomes the expression  $x^4$ . The resulting expression can be passed to the `sympy.diff` function to differentiate it with respect to  $x$ . Differentiating it will now give a sympy object representing  $4x^3$ .

```
x = sympy.symbols('x', real=True)
y = x**4
dydx = sympy.diff(y,x)
print(dydx)
```

Using these expressions, sympy can turn them into functions that takes an argument with the `sympy.lambdify` function. Effectively giving us the expressions  $y(x) = x^4$  and  $\frac{dy}{dx}(x) = 4x^3$ .

```
y = sympy.lambdify(x, y)
dydx = sympy.lambdify(x, dydx)
```

### 1.2 (ii) Finite Difference Implementation

The python function that computes the finite difference of a function:

- Inputs are:
  - $f$ : the function
  - $x$ : input value for the function
  - $\delta$ : the perturbation

- The finite difference can be implemented as  $\frac{f(x) - f(x - \delta)}{\delta}$
- $\frac{f(x + \delta) - f(x - \delta)}{2 * \delta}$  could be used to negate the offset (perturbation is  $2 * \delta$  in this case).
- Finite difference nudges a function a tiny bit in a direction and then divides by that difference to find by how much the function value changed relative to that nudge, giving the slope.

```
def finiteDiff(f, x, delta):
    return (f(x) - f(x - delta)) / delta
```

Fig 1 shows finite difference method with  $\delta = 0.01$  generates a curve almost identical to the symbolic one, although a slight fringe of blue is seen at  $x > 1$  and  $x < -1$ , indicating the tiny offset caused by the nudge in one direction.

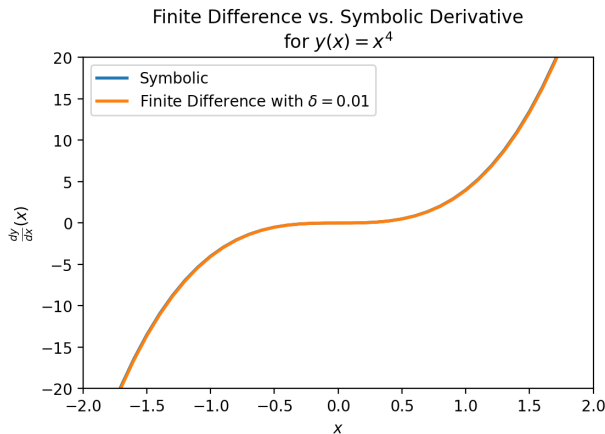


Figure 1

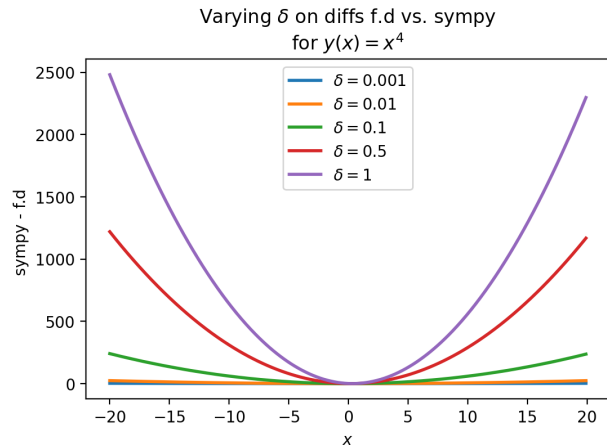


Figure 2

### 1.3 (iii) Varying $\delta$ on Finite Difference

The difference between symbolic derivative and finite difference is plotted. In Fig 2, as  $\delta$  increases, we can see the error getting bigger and bigger for values that are away from  $x = 0$ . Error is bigger further away as a nudge in  $x$  causes a larger change. A  $\delta < 0.01$ , seems to produce little error even at large  $x$ .

## 2 (b) Gradient Descent Optimisation Algorithm

### 2.1 (i) Gradient Descent Implementation

Gradient descent (g.d) finds the  $x$  that minimises some function  $f(x)$  i.e g.d finds  $\operatorname{argmin}_x f(x)$ .

- The implementation uses the derivative of  $f(x)$  i.e  $\frac{df}{dx}(x)$ .
- g.d requires a starting  $x$  value i.e  $x_0$ .
- For some defined number of iterations  $i_{max}$ , g.d iteratively adjusts  $x_i$ .
- One iteration approximates how to modify  $x_i$  in order to move towards the minimum of  $f(x)$ .
- Approximating is accomplished by using  $\frac{df}{dx}(x)$  to find the slope of the curve at point  $x_i$ , and using the slope as the local approximation for which direction relative to the point  $f(x_i)$ , the minimum of  $f(x)$  lies.
- A step size for  $x_i$  is calculated by multiplying  $\frac{df}{dx}(x)$  by some scalar  $\alpha$ , in this case  $\alpha$  is manually picked and stays constant throughout all the iterations, although the magnitude of  $\frac{df}{dx}(x)$  itself may change and alter the step magnitude.

- The negative of  $\frac{df}{dx}(x_i)$  guarantees an instantaneous step for  $x_i$  in the downwards direction for  $f(x_i)$ .  $x_{i+1} = x_i + \text{step}$ , and the process is repeated.

```
def gradient_descent(df, x0, alpha=0.15, i_max=50):
    x = x0
    for k in range(i_max):
        step = alpha * -df(x)
        x = x + step
    return x
```

## 2.2 (ii) Visualising Gradient Descent

Gradient Descent is run with,  $x_0 = 1$ ,  $\alpha = 0.1$ ,  $y(x) = x^4$ .  $x$  and  $y(x)$  vary with each gradient descent iteration.

- Figure 6 plots the function to be optimised; it is convex, but there is a very flat portion at  $-0.5 < x < 0.5$ . We know that the  $\text{argmin}_x f(x) = 0$  for this function.
- Figure 4 plots  $x_i$  against  $i$ ;  $x_i$  decreases rapidly on the very first iteration, reaching 40% of the way to 0, but then begins to slow down rapidly, this is because the slope of the function is significantly smaller at  $x < 0.6$  compared to  $x = 1$ , and the slope keeps on decreasing at a rate of  $4x^3$ , which is quite rapid for a constant  $\alpha$ , and the slope is important in the step as  $\text{step} = \alpha * \text{slope}_{x_i}$ .
- Figure 5 plots  $y$  against  $i$ ; the majority of the optimisation happens in 2 iterations, and very little progress is made after  $i = 2$ , it essentially comes to a flat line 3 iterations onward.

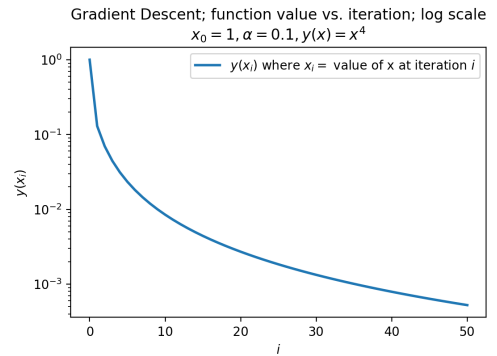


Figure 3:  $y(x_i)$  on log scale

We see that  $x_i$  takes longer to become a flat line than  $y(x_i)$ , this is because of the flat shape of the bottom of  $x^4$ . Once  $x_i$  reaches the bottom,  $x_i$  itself can still move a bit, but will not have an equally proportional impact on  $y(x_i)$ . Even on a log scale (fig 3) the optimisation is seen to slow down due to the  $4x^3$  slope.

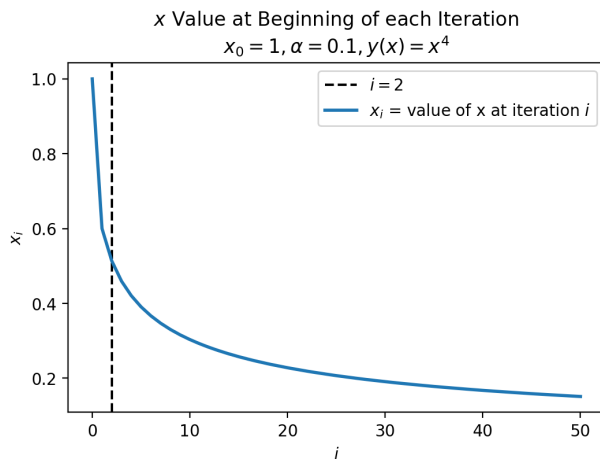


Figure 4

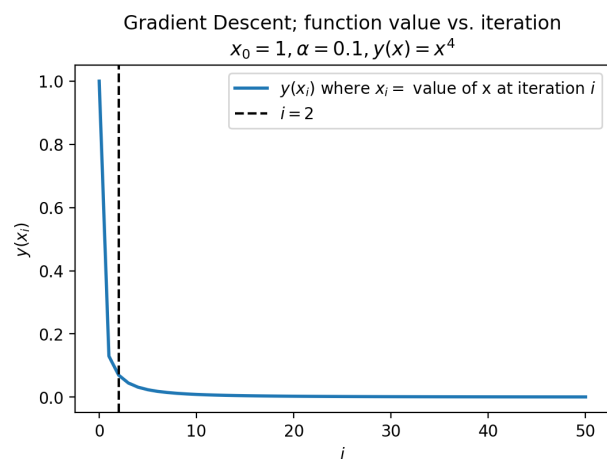


Figure 5

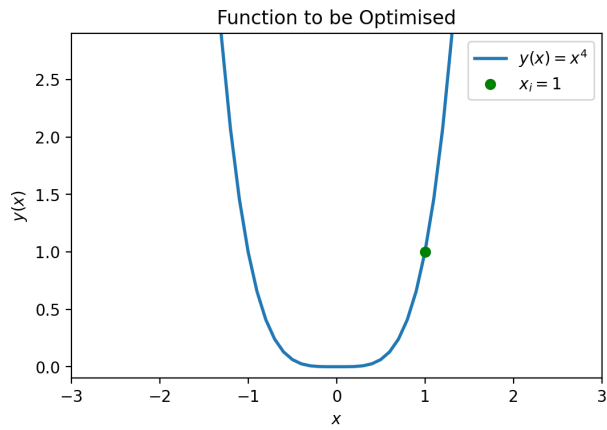


Figure 6

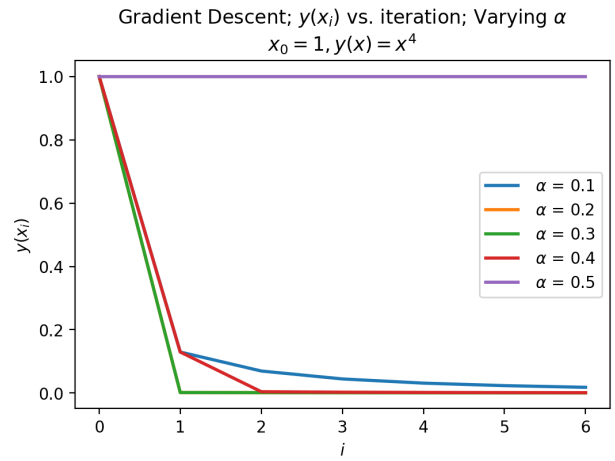


Figure 7

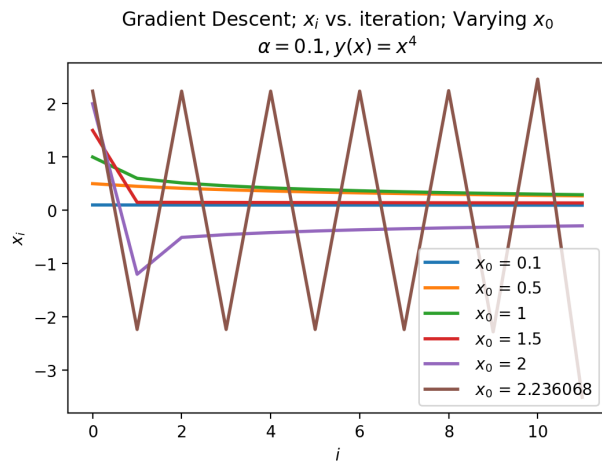


Figure 8

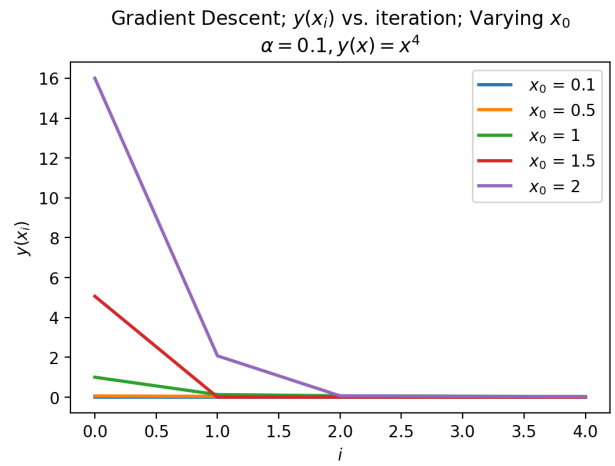


Figure 9

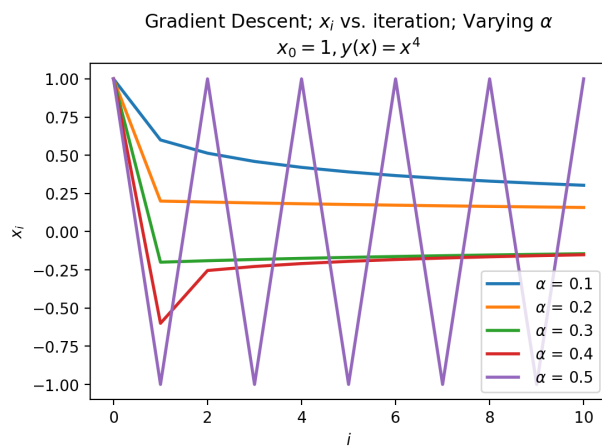


Figure 10

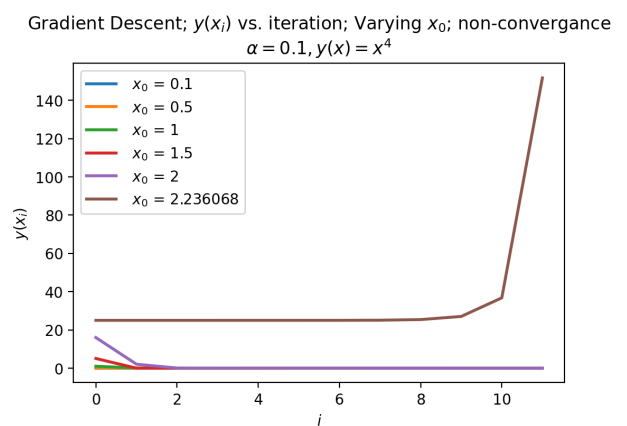


Figure 11

## 2.3 (iii) Varying Step Size $\alpha$ and $x_0$

- Varying  $x_0$ 
  - Plotting  $x$  - Fig. 8 : We can see that  $x_0 > 2.236068$  would lead to an explosive non-convergence; it keeps jumping over to the other side of the curve, higher than what it was before. This is because the slope at  $x > 2.236068$  is too high in magnitude for the combination with  $\alpha = 0.1$  and therefore results in too large of a step size.  $x_0 = 2$  jumps over to the other side, but not higher than it was before and still manages to converge. Once the  $x_i$  reaches within  $-0.5 < x < 0.5$ , the size of the slope is tiny relative to the  $\alpha$ , and essentially stops making progress.
  - Plotting  $y$  that converge - Fig. 9 : We see that even though  $x_i$  don't converge on the same point for different  $x_0$ , they all converge on practically the same  $y$  value, and all of them within only 2 iterations.
  - Plotting  $y$  that doesn't converge - Fig. 11 : We can see that  $x_0 > 2.236068$  will not converge, the function value keeps increasing due to the larger and larger jumps to each side of the convex function.
- Varying  $\alpha$ 
  - Plotting  $x$  - Fig.10 : An  $\alpha > 0.5$  would lead to an explosive non convergence, as it would cause jumps to the other side to a higher  $y$  value. Rest of the  $\alpha$  converge, but it seems like the very first jump determines where its going to get stuck in the flat region.
  - Plotting  $y$  - Fig. 7 :  $\alpha > 0.5$  shows non-convergence, and the rest of the  $\alpha$ 's converge closely to each other.  $\alpha = 0.1$  makes keeps making progress even after 5 iterations in, seems like it's the nature of the rapidly flattening function rather than a small constant  $\alpha$  that causes the slowdown of the convergence.

Both  $x_0$  and  $\alpha$  cause un-forgiving explosions if not chosen small enough, but as long as the first step size is small enough, they converge to practically the same  $y$  value. The functions rapidly decreasing slope, rather than the chosen constant  $\alpha$  value, is what causes the quicksand behaviour towards the minimum, a small  $\alpha$  will allow a bit more flexible placement of  $x_i / x_0$  as it'll be a tiny bit less likely to shoot off exponentially, while still being able to converge. But since  $x^4$ 's slope decreases *and* increases rapidly, it won't give that much flexibility.

## 3 (c) Optimising $y(x) = \gamma x^2$ and $y(x) = \gamma|x|$

### 3.1 (i) Optimising $y(x) = \gamma x^2$

We have  $y(x) = \gamma x^2$ .  $y(x)$  and  $y'(x)$  are plotted (fig 12, 13). It is a strongly convex curve. Larger  $\gamma$ 's have a steeper curve, and the derivatives are just straight lines at certain slopes.

$x$  and  $y$  are plotted against iteration (fig 15, 14). We first observe that the optimisations stay at a constant rate on the log scale. We can see that for higher  $\gamma$ 's, the rate of convergence is higher, and stays constant logarithmically even though  $y$  is getting down to  $10^{-21}$ , this stable rate of optimisation must be due to the derivative of the function being a straight line, and hence the step size is always scaling with the logarithm.

The rate of convergence for the lower  $\gamma$ 's is much slower as the slope is much smaller, although the optimisation already started at a small value and so doesn't need to move, we could say that the  $\alpha$  value is appropriately moderate for such a small slope at that  $x_0$  and  $\gamma$ . Though  $\alpha$  may need to be bumped up for larger  $x_0$  since it looks like slope will still be low even far away, or else it might take too long to converge.

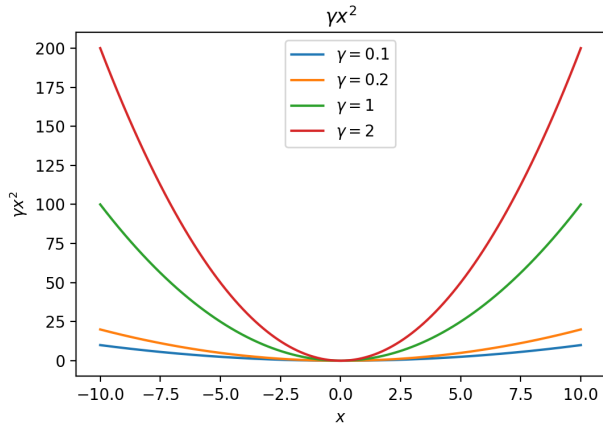


Figure 12

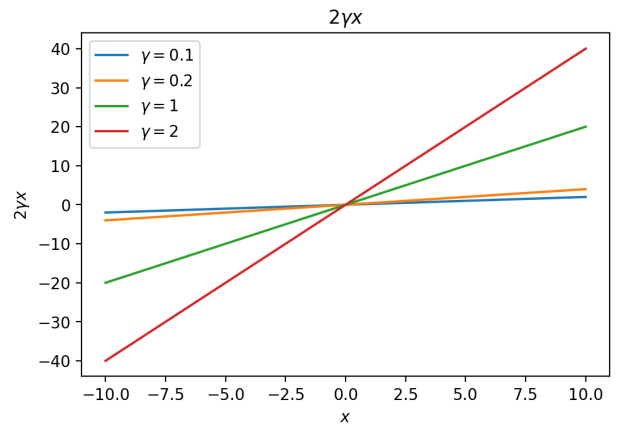


Figure 13

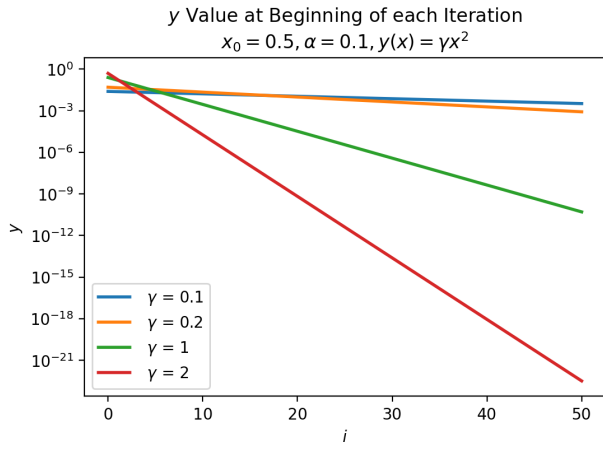


Figure 14

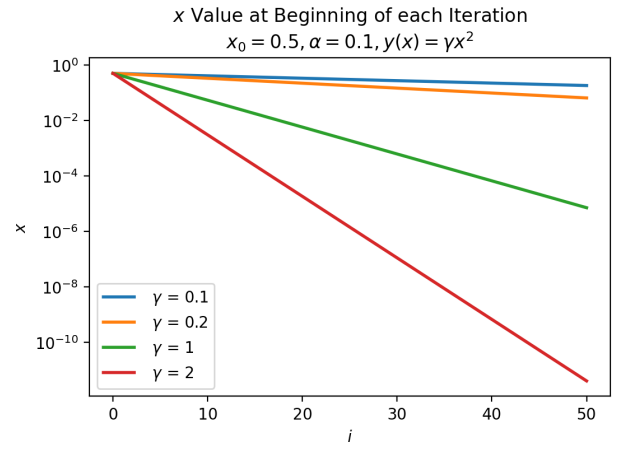


Figure 15

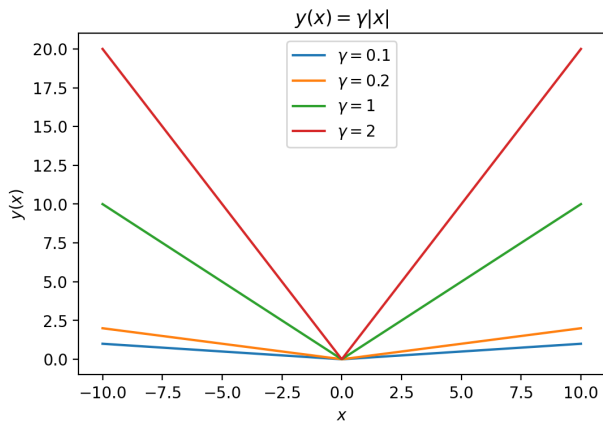


Figure 16

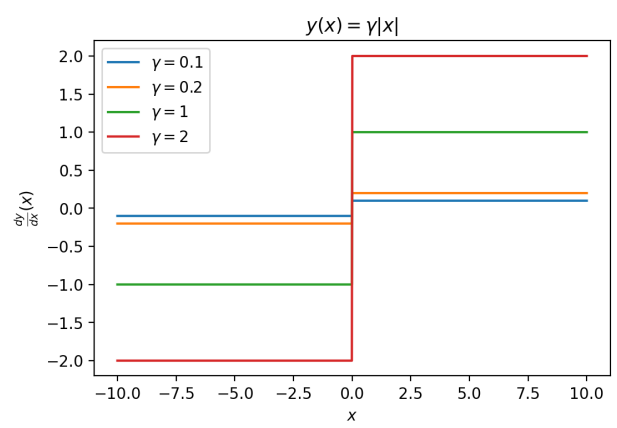


Figure 17

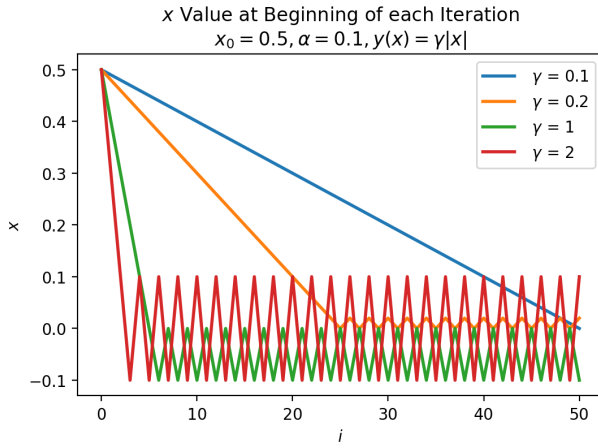


Figure 18

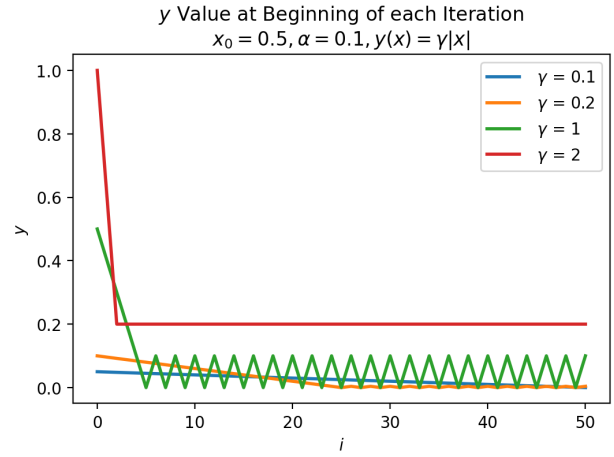


Figure 19

### 3.2 (ii) Optimising $y(x) = \gamma|x|$

We have  $y(x) = \gamma|x|$ .  $y(x)$  and  $y'(x)$  are plotted (fig 16, 17). The functions have kinks in them at  $x = 0$ . For larger  $\gamma$ 's the slope is bigger,  $\gamma = +/ - \text{slope}$ .

$x$  and  $y$  are plotted against iteration (fig 18, 19). We can see the higher  $\gamma$ 's move towards  $x = 0$  faster due to the larger slope. We can observe chattering/zigzagging in a loop for all of the  $\gamma$ 's (except for  $\gamma = 0.1$  since it hasn't reached the chattering stage at iteration 50 yet). This exactly repeated loop happens because once the  $x_i$  jumps to the other side of the kink, on the next iteration it will try jump back towards the minimum. It would need to get exactly on  $x = 0$ , though most likely it will fall on the slope - it is at this point it enters the loop; the slope is constant, and so is alpha, so it will jump back and forth by the same amount (depending on slope and alpha) across the kink.

We can see the gap that's jumped by the larger  $\gamma$ 's is larger, this is because the loop can be entered from a larger value of  $x_i$  since the slope is larger therefore jumps are bigger. Conversely, smaller step sizes end up chattering closer to the optimum value because it inched closer to the kink before jumping over and entering the loop. We can see  $\gamma = 2$  y value doesn't oscillate because it's actually jumping between x values of the same magnitude and hence same function value.

## 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
6 import numpy as np
7 import sympy
8
9 x = sympy.symbols('x', real=True)
10 y = x**4
11 dydx = sympy.diff(y,x)
12 print(dydx)
13
14 y = sympy.lambdify(x, y)
15 dydx = sympy.lambdify(x, dydx)
16
17 def finiteDiff(f, x, delta):
18     return (f(x) - f(x - delta)) / delta
19
20 def finiteDiff(f, x, delta):
21     return (f(x + delta) - f(x - delta)) / (2 * delta)

```

```

22
23 def axset(ax, xrange, xoffset, yrange, yoffset):
24     ax.set(xlim=(xoffset-xrange, xoffset+xrange),
25            ylim=(yoffset-yrange, yoffset+yrange))
26
27 xs = np.arange(-20, 20, 0.1)
28
29 ys_sym = dydx(xs)
30
31 ys_finiteDiff = []
32 for x in xs:
33     ys_finiteDiff.append(finiteDiff(y, x, 0.01))
34
35 fig, ax = plt.subplots()
36 ax.set_ylabel(r'$\frac{dy}{dx}(x)$')
37 ax.set_xlabel(r'$x$')
38 ax.set_title(r'Finite Difference vs. Symbolic Derivative' "\n" r'for $y(x) = x^4$')
39
40 ax.plot(xs, ys_sym, linewidth=2.0)
41 ax.plot(xs, ys_finiteDiff, linewidth=2.0)
42 ax.legend(("Symbolic", r'Finite Difference with $\delta = 0.01$'))
43 axset(ax, xrange=2, xoffset=0, yrange=20, yoffset=0)
44
45 # fig.show()
46
47 # ax.set(
48 #     xlim=(-3, 3),
49 #     ylim=(-20, 20),
50 #     xticks=np.arange(1, 8),
51 #     yticks=np.arange(1, 8),
52 # )
53
54 dydx = lambda x: 4 * x**3
55 y = lambda x: x**4
56
57 xs = np.arange(-20, 20, 0.1)
58
59 deltas = [0.001, 0.01, 0.1, 0.5, 1]
60 ys_dif = []
61 for delta in deltas:
62     dif = []
63     for x in xs:
64         fd = finiteDiff(y, x, delta)
65         ex = dydx(x)
66         dif += [ex - fd]
67
68     ys_dif += [(dif, delta)]
69
70 fig, ax = plt.subplots()
71 legend_labels = []
72 for (diff, delta) in ys_dif:
73     legend_labels += [r'$\delta = $' + str(delta)]
74     ax.plot(xs, diff, linewidth=2.0)
75
76 ax.set_title(r'Varying $\delta$ on diffs f.d vs. sympy' "\n" r'for $y(x) = x^4$')
77 ax.set_ylabel(r'sympy - f.d ')
78 ax.set_xlabel(r'$x$')
79 ax.legend(legend_labels)
80 # axset(ax, xrange=3, xoffset=1.5, yrange=20, yoffset=10)
81
82 def gradient_descent(df, x0, alpha=0.15, i_max=50):
83     x = x0
84     for k in range(i_max):
85         step = alpha * -df(x)
86         x = x + step
87     return x
88
89 class QuadraticFn():

```



```

90     def f(self, x):
91         return x**2                                # function value f(x)
92
93     def df(self, x):
94         return x*2                                # derivative of f(x)
95
96 fn = QuadraticFn()
97
98 def gradDesc(fn, x0, alpha=0.15, num_iters=50):
99     x = x0                                          # starting point
100    X = np.array([x])                              # array of x history
101    F = np.array(fn.f(x))                          # array of f(x) history
102    for k in range(num_iters):
103        step = alpha * fn.df(x)
104        x = x - step
105        X = np.append(X, [x], axis=0)              # add current x to history
106        F = np.append(F, fn.f(x))                  # add value of current f(x) to history
107    return (X,F)
108
109 def gradDesc3(f, df, x0, alpha=0.15, num_iters=50):
110     x = x0                                          # starting point
111     X = np.array([x])                              # array of x history
112     F = np.array(f(x))                            # array of f(x) history
113     for k in range(num_iters):
114         step = alpha * df(x)
115         x = x - step
116         # print(x)
117         X = np.append(X, [x], axis=0)              # add current x to history
118         F = np.append(F, f(x))                    # add value of current f(x) to history
119     return (X,F)
120
121 (X, F) = gradDesc(fn, 1)
122 x = gradient_descent(fn.df, 1)
123
124 xs = np.arange(-20, 20, 0.1)
125
126 ys = dydx(xs)
127 ys = y(xs)
128
129 fig, ax = plt.subplots()
130 ax.set_ylabel(r'$y(x)$')
131 ax.set_xlabel(r'$x$')
132
133 ax.set_title(r'Function to be Optimised')
134 ax.plot(xs, ys, linewidth=2.0)
135 ax.plot(1, y(1), 'go')
136 ax.legend(("y(x) = x^4", r'$x_i = 1$'))
137
138 # ax.axvline(x=1, color='k', linestyle='--')
139 axset(ax, xrange=3, xoffset=0, yrange=1.5, yoffset=1.4)
140
141 (_, F) = gradDesc3(y, dydx, x0=1, alpha=0.1)
142 iters = np.arange(0, len(F))
143
144 fig, ax = plt.subplots()
145 ax.set_ylabel(r'$y(x_{i})$')
146 ax.set_xlabel(r'$i$')
147 ax.set_title(r'Gradient Descent; function value vs. iteration' "\n"
148             r'$x_0=1, \alpha=0.1$, y(x) = x^4$,')
149 ax.plot(iters, F, linewidth=2.0)
150 ax.axvline(x=2, color='k', linestyle='--')
151
152 ax.legend((r'$y(x_{i})$ where $x_i$ = value of x at iteration $i$', r'$i=2$', ))
153
154 (_, F) = gradDesc3(y, dydx, x0=1, alpha=0.1)
155 iters = np.arange(0, len(F))
156
157 fig, ax = plt.subplots()

```

```

158 ax.set_ylabel(r'$y(x_{i})$')
159 ax.set_xlabel(r'$i$')
160 ax.set_title(r'Gradient Descent; function value vs. iteration; log scale' "\n"
161             r'$x_0=1, \alpha=0.1, y(x) = x^4$',)
162
163 ax.semilogy(iters, F, linewidth=2.0)
164 ax.legend((r'$y(x_{i})$ where $x_i=$ value of x at iteration $i$',))
165
166 (X, _) = gradDesc3(lambda x : x**4, lambda x : 4*x**3, x0=1, alpha=0.1)
167 iters = np.arange(0, len(X))
168
169 fig, ax = plt.subplots()
170 ax.set_ylabel(r'$x_i$')
171 ax.set_xlabel(r'$i$')
172 ax.set_title(r'$x$ Value at Beginning of each Iteration' "\n"
173             r'$x_0=1, \alpha=0.1, y(x) = x^4$',)
174 ax.axvline(x=2, color='k', linestyle='--')
175 ax.plot(iters, X, linewidth=2.0)
176
177 ax.legend((r'$i=2$', r'$x_{i}$ = value of x at iteration $i$',))
178
179 # (X, _) = gradDesc3(y, dydx, x0=1, alpha=0.1) # given a range of alphas, give back
180 # corresponding dimensions of answers, same for x0s
181 # perhaps it gives back objects that describe the shape of the output in detail,
182 # perhaps what dimension represents what, and how many there are
183
184 x0s = np.arange(0.1, 2, 0.1)
185 num_iters = 50
186
187 Xs = np.array([])
188 for x0 in x0s:
189     (X, _) = gradDesc3(lambda x : x**4, lambda x : 4*x**3, x0=x0, alpha=0.1,
190                       num_iters=num_iters)
191     if len(Xs) > 0:
192         Xs = np.append(Xs, [X], axis=0)
193     else:
194         Xs = np.array([X])
195
196 # fig, ax = plt.subplots()
197 # ax.set_ylabel(r'$x_i$')
198 # ax.set_xlabel(r'$i$')
199 # print(num_iters)
200
201 # print(Xs.shape)
202 # 0th index is x0 = 1.7
203 # [0,0] (x0=0.1,i=0)
204 # [0,1] (x0=0.1,i=1) 2 params input, Xs is the output
205
206 # [1,0] (x0=0.2,i=1)
207 # [1,1] (x0=0.2,i=1) 2 params input, Xs is the output
208
209 # indexes of inputs must correspond to position of output
210
211 itersY, x0sX = np.meshgrid(np.arange(num_iters+1), x0s)
212 # print(x0sX)
213 # print(itersY)
214 # print(Xs)
215
216 fig = plt.figure()
217 ax = plt.axes(projection='3d')
218 # ax.contour3D(x0sX, itersY, Xs, 100, cmap='binary')
219 ax.plot_surface(x0sX, itersY, Xs, rstride=1, cstride=1,
220               cmap='viridis', edgecolor='none')
221 ax.view_init(12, 75)
222 # ax.view_init(12, 120)
223 ax.view_init(12, 30)
224 # ax.view_init(0, 0)

```

```

223 ax.set_xlabel(r'$x_0$')
224 ax.set_ylabel(r'$i$')
225 ax.set_zlabel(r'$x_i$')
226
227 # looks like i get slow on these kinds of problems
228 # probably practice will help
229 # and perhaps doing going slowly through them and
230 # understanding them will help
231
232 x0s = [0.1, 0.5, 1, 1.5, 2, 2.236068]
233 # 2.23607
234 num_iters = 11
235
236 Xs = np.array([])
237 for x0 in x0s:
238     (X, _) = gradDesc3(lambda x : x**4,
239                        lambda x : 4*x**3,
240                        x0=x0,
241                        alpha=0.1,
242                        num_iters=num_iters)
243     if len(Xs) > 0:
244         Xs = np.append(Xs, [(X,x0)], axis=0)
245     else:
246         Xs = np.array([(X, x0)])
247
248 fig, ax = plt.subplots()
249 ax.set_ylabel(r'$x_i$')
250 ax.set_xlabel(r'$i$')
251 ax.set_title(r'Gradient Descent; $x_i$ vs. iteration; Varying $x_0$' "\n"
252             r'$\alpha=0.1$, $y(x) = x^4$')
253 legend_labels = []
254 for (X, x0) in Xs:
255     ax.plot(range(num_iters+1), X, linewidth=2.0)
256     legend_labels += [(r'$x_{0}$ = ' + str(x0))]
257 ax.legend(legend_labels)
258
259 alphas = [0.1, 0.2, 0.3, 0.4, 0.5]
260 num_iters = 10
261
262 Xs = np.array([])
263 for alpha in alphas:
264     (X, _) = gradDesc3(lambda x : x**4,
265                        lambda x : 4*x**3,
266                        x0=1,
267                        alpha=alpha,
268                        num_iters=num_iters)
269     if len(Xs) > 0:
270         Xs = np.append(Xs, [(X,alpha)], axis=0)
271     else:
272         Xs = np.array([(X, alpha)])
273
274 fig, ax = plt.subplots()
275 ax.set_ylabel(r'$x_i$')
276 ax.set_xlabel(r'$i$')
277 ax.set_title(r'Gradient Descent; $x_i$ vs. iteration; Varying $\alpha$' "\n"
278             r'$x_0=1$, $y(x) = x^4$')
279 legend_labels = []
280 for (X, alpha) in Xs:
281     ax.plot(range(num_iters+1), X, linewidth=2.0)
282     legend_labels += [(r'$\alpha$ = ' + str(alpha))]
283 ax.legend(legend_labels)
284
285 x0s = [0.1, 0.5, 1, 1.5, 2]
286 num_iters = 4
287
288 Ys = np.array([])
289 for x0 in x0s:
290     (_, Y) = gradDesc3(lambda x : x**4,

```

```

291         lambda x : 4*x**3,
292         x0=x0,
293         alpha=0.1,
294         num_iters=num_iters)
295     if len(Ys) > 0:
296         Ys = np.append(Ys, [(Y,x0)], axis=0)
297     else:
298         Ys = np.array([(Y, x0)])
299
300 fig, ax = plt.subplots()
301 ax.set_ylabel(r'$y(x_i)$')
302 ax.set_xlabel(r'$i$')
303 ax.set_title(r'Gradient Descent; $y(x_i)$ vs. iteration; Varying $x_0$' "\n"
304             r'$\alpha=0.1$, $y(x) = x^4$')
305 legend_labels = []
306 for (Y, x0) in Ys:
307     ax.plot(range(num_iters+1), Y, linewidth=2.0)
308     legend_labels += [(r'$x_{0}$ = ' + str(x0))]
309 ax.legend(legend_labels)
310
311 x0s = [0.1, 0.5, 1, 1.5, 2, 2.236068]
312 num_iters = 11
313
314 Ys = np.array([])
315 for x0 in x0s:
316     (_, Y) = gradDesc3(lambda x : x**4,
317                        lambda x : 4*x**3,
318                        x0=x0,
319                        alpha=0.1,
320                        num_iters=num_iters)
321     if len(Ys) > 0:
322         Ys = np.append(Ys, [(Y,x0)], axis=0)
323     else:
324         Ys = np.array([(Y, x0)])
325
326 fig, ax = plt.subplots()
327 ax.set_ylabel(r'$y(x_i)$')
328 ax.set_xlabel(r'$i$')
329 ax.set_title(r'Gradient Descent; $y(x_i)$ vs. iteration; Varying $x_0$; non-
330             convergence' "\n"
331             r'$\alpha=0.1$, $y(x) = x^4$')
332 legend_labels = []
333 for (Y, x0) in Ys:
334     ax.plot(range(num_iters+1), Y, linewidth=2.0)
335     legend_labels += [(r'$x_{0}$ = ' + str(x0))]
336 ax.legend(legend_labels)
337
338 x0s = [0.1, 0.5, 1, 1.5, 2]
339 num_iters = 12
340
341 Ys = np.array([])
342 for x0 in x0s:
343     (_, Y) = gradDesc3(lambda x : x**4,
344                        lambda x : 4*x**3,
345                        x0=x0,
346                        alpha=0.1,
347                        num_iters=num_iters)
348     if len(Ys) > 0:
349         Ys = np.append(Ys, [(Y,x0)], axis=0)
350     else:
351         Ys = np.array([(Y, x0)])
352
353 fig, ax = plt.subplots()
354 ax.set_ylabel(r'$x_i$')
355 ax.set_xlabel(r'$i$')
356 legend_labels = []
357 for (Y, x0) in Ys:
358     ax.semilogy(range(num_iters+1), Y, linewidth=2.0)

```

```

358     legend_labels += [(r' $x_{0}$ = ' + str(x0))]
359 ax.legend(legend_labels)
360
361 alphas = [0.1, 0.2, 0.3, 0.4, 0.5]
362 num_iters = 6
363
364 Ys = np.array([])
365 for alpha in alphas:
366     (_, Y) = gradDesc3(lambda x : x**4,
367                        lambda x : 4*x**3,
368                        x0=1,
369                        alpha=alpha,
370                        num_iters=num_iters)
371     if len(Ys) > 0:
372         Ys = np.append(Ys, [(Y,alpha)], axis=0)
373     else:
374         Ys = np.array([(Y, alpha)])
375
376 fig, ax = plt.subplots()
377 ax.set_ylabel(r'$y(x_i)$')
378 ax.set_xlabel(r'$i$')
379 ax.set_title(r'Gradient Descent; $y(x_i)$ vs. iteration; Varying $\alpha$' "\n"
380             r'$x_0=1$, $y(x) = x^4$',)
381 legend_labels = []
382 for (Y, alpha) in Ys:
383     ax.plot(range(num_iters+1), Y, linewidth=2.0)
384     legend_labels += [(r' $\alpha$ = ' + str(alpha))]
385 ax.legend(legend_labels)
386
387 from jax import grad
388 y = lambda x, gamma: gamma * x**2
389
390 # grad by default will take the derivative of the first parameter of the function
391 # that we pass
392 dydx = grad(y)
393
394 def visualise_fn(fn, l=-10, r=10, n=1000):
395     xs = np.linspace(l, r, num=n)
396     y = np.array([fn(x) for x in xs])
397     plt.plot(xs,y)
398
399 def labels_fn(ax, legend, xaxis=r'$x$', yaxis=r'$y(x)$', title="Title"):
400     ax.set_xlabel(xaxis)
401     ax.set_ylabel(yaxis)
402     ax.set_title(title)
403     ax.legend(legend)
404
405 def visualise_fns(fns, labels_fn=labels_fn, l=-10, r=10, n=1000):
406     xs = np.linspace(l, r, num=n)
407     ys = []
408     fig, ax = plt.subplots()
409     for fn in fns:
410         y = np.array([fn(x) for x in xs])
411         ax.plot(xs,y)
412         labels_fn(ax)
413
414 fns_gamma = (lambda fn, gammas: [(lambda x, gamma=gamma: fn(x, gamma)) for gamma in
415                                gammas])
416
417 gammas = [0.1, 0.2, 1, 2]
418 legend = [(r'$\gamma$=' + str(gamma)) for gamma in gammas]
419 labels_y = lambda ax: labels_fn(ax, legend, yaxis=r'$\gamma x^2$', title=r'$\gamma x^2$')
420 labels_dy = lambda ax: labels_fn(ax, legend, yaxis=r'$2\gamma x$', title=r'$2 \gamma x$')
421
422 visualise_fns(fns_gamma(dydx, gammas), labels_fn=labels_dy)

```

```

422 visualise_fns(fns_gamma(y, gammas), labels_fn=labels_y)
423
424 def gamma_grad(gamma, num_iters=40, x0=1, alpha=0.1):
425     return gradDesc3(f= $\lambda x : y(x, \text{gamma})$ ,
426                     df= $\lambda x : \text{dydx}(x, \text{gamma})$ ),
427                     x0=x0,
428                     alpha=0.1,
429                     num_iters=num_iters)
430
431 gammas = [0.1, 0.2, 1, 2]
432 num_iters=50
433 Ys = np.array([])
434 # wonder how can generalise this for future ease of use
435 for gamma in gammas:
436     (_, Y) = gamma_grad(gamma, num_iters=num_iters, x0=0.5, alpha=0.1)
437     if len(Ys) > 0:
438         Ys = np.append(Ys, [(Y, gamma)], axis=0)
439     else:
440         Ys = np.array([(Y, gamma)])
441
442 fig, ax = plt.subplots()
443 legend_labels = []
444 for (Y, gamma) in Ys:
445     # ax.plot(range(num_iters+1), Y, linewidth=2.0)
446     ax.semilogy(range(num_iters+1), Y, linewidth=2.0)
447     legend_labels += [(r' $\gamma =$ ' + str(gamma))]
448 ax.legend(legend_labels)
449 ax.set_ylabel(r' $y$ ')
450 ax.set_xlabel(r' $i$ ')
451 ax.set_title(r' $y$  Value at Beginning of each Iteration' "\n"
452             r' $x_0=0.5, \alpha=0.1, y(x) = \gamma x^2$ ',)
453
454 gammas = [0.1, 0.2, 1, 2]
455 num_iters=50
456 Xs = np.array([])
457 # wonder how can generalise this for future ease of use
458 for gamma in gammas:
459     (X, _) = gamma_grad(gamma, num_iters=num_iters, x0=0.5, alpha=0.1)
460     if len(Xs) > 0:
461         Xs = np.append(Xs, [(X, gamma)], axis=0)
462     else:
463         Xs = np.array([(X, gamma)])
464
465 fig, ax = plt.subplots()
466 legend_labels = []
467 for (X, gamma) in Xs:
468     # ax.plot(range(num_iters+1), X, linewidth=2.0)
469     ax.semilogy(range(num_iters+1), X, linewidth=2.0)
470     legend_labels += [(r' $\gamma =$ ' + str(gamma))]
471 ax.legend(legend_labels)
472 ax.set_ylabel(r' $x$ ')
473 ax.set_xlabel(r' $i$ ')
474 ax.set_title(r' $x$  Value at Beginning of each Iteration' "\n"
475             r' $x_0=0.5, \alpha=0.1, y(x) = \gamma x^2$ ',)
476
477 y =  $\lambda x, \text{gamma} : \text{gamma} * \text{abs}(x)$ 
478 dydx = grad(y)
479
480 gammas = [0.1, 0.2, 1, 2]
481 legend = [(r' $\gamma =$ ' + str(gamma)) for gamma in gammas]
482 labels_y =  $\lambda \text{ax} : \text{labels\_fn}(\text{ax}, \text{legend}, \text{yaxis}=\text{r}'y(x)'$ , title=r' $y(x) = \gamma |$ 
483              $x|$ ')
484 labels_dy =  $\lambda \text{ax} : \text{labels\_fn}(\text{ax}, \text{legend}, \text{yaxis}=\text{r}'\frac{dy}{dx}(x)'$ , title=r' $y(x) = \gamma |x|$ ')
485 gamma_grad
486
487 visualise_fns(fns_gamma(dydx, gammas), labels_fn=labels_dy)

```

```

488
489 visualise_fns(fns_gamma(y, gammas), labels_fn=labels_y)
490
491 def gamma_grad(gamma, num_iters=40, x0=1, alpha=0.1):
492     return gradDesc3(f=lambda x : y(x, gamma),
493                     df=(lambda x : dydx(x, gamma)),
494                     x0=x0,
495                     alpha=0.1,
496                     num_iters=num_iters)
497
498 gammas = [0.1, 0.2, 1, 2]
499 num_iters=50
500 Ys = np.array([])
501 # wonder how can generalise this for future ease of use
502 for gamma in gammas:
503     (_, Y) = gamma_grad(gamma, num_iters=num_iters, x0=0.5, alpha=0.1)
504     if len(Ys) > 0:
505         Ys = np.append(Ys, [(Y,gamma)], axis=0)
506     else:
507         Ys = np.array([(Y, gamma)])
508
509 fig, ax = plt.subplots()
510 legend_labels = []
511 for (Y, gamma) in Ys:
512     ax.plot(range(num_iters+1), Y, linewidth=2.0)
513     # ax.semilogy(range(num_iters+1), Y, linewidth=2.0)
514     legend_labels += [(r' $\gamma$ = ' + str(gamma))]
515 ax.legend(legend_labels)
516 ax.set_ylabel(r'$y$')
517 ax.set_xlabel(r'$i$')
518 ax.set_title(r'$y$ Value at Beginning of each Iteration' "\n"
519             r'$x_0=0.5, \alpha=0.1, y(x) = \gamma x^2$',)
520
521 gammas = [0.1, 0.2, 1, 2]
522 num_iters=50
523 Xs = np.array([])
524 # wonder how can generalise this for future ease of use
525 for gamma in gammas:
526     (X, _) = gamma_grad(gamma, num_iters=num_iters, x0=0.5, alpha=0.1)
527     if len(Xs) > 0:
528         Xs = np.append(Xs, [(X,gamma)], axis=0)
529     else:
530         Xs = np.array([(X, gamma)])
531
532 fig, ax = plt.subplots()
533 legend_labels = []
534 for (X, gamma) in Xs:
535     ax.plot(range(num_iters+1), X, linewidth=2.0)
536     # ax.semilogy(range(num_iters+1), X, linewidth=2.0)
537     legend_labels += [(r' $\gamma$ = ' + str(gamma))]
538 ax.legend(legend_labels)
539 ax.set_ylabel(r'$x$')
540 ax.set_xlabel(r'$i$')
541 ax.set_title(r'$x$ Value at Beginning of each Iteration' "\n"
542             r'$x_0=0.5, \alpha=0.1, y(x) = \gamma x^2$',)
543
544
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 class OptimisationAlgorithm:
11     def __init__(self, algorithm, algorithm_name):
12         self.algorithm = algorithm
13         self.algorithm_name = algorithm_name

```

```

14
15     arguments = algorithm.__code__.co_varnames[:algorithm.__code__.co_argcount]
16     self.all_parameters = arguments
17     self.standard_parameters = ("x0", "f", "iters")
18     self.hyperparameters = list(filter(lambda arg: arg not in self.
standard_parameters, arguments))
19
20     def __type_check_parameters(self, input_record):
21         for key in input_record.keys():
22             if key not in self.all_parameters:
23                 raise NameError(key + " is not one of: " + str(self.all_parameters))
24         for key in self.all_parameters:
25             if key not in input_record:
26                 raise NameError(key + " is missing from input: " + str(list(
input_record.keys()))))
27
28     def set_parameters(self, **input_record):
29         self.__type_check_parameters(input_record)
30         self.parameter_values = input_record
31         return self
32
33     def run(self):
34         inputs = self.__make_input()
35         for input in inputs:
36             input["X"], input["Y"] = self.algorithm(**input)
37             input["X"] = np.array(input["X"])
38             input["Y"] = np.array(input["Y"])
39             input["algorithm"] = self
40         return inputs
41
42     def __make_input(self):
43         kwargs = self.parameter_values.copy()
44         expected_vector = { "x0" }
45         for key, value in kwargs.items():
46             if key in expected_vector:
47                 value = np.array(value)
48                 if value.ndim == 1:
49                     kwargs[key] = [value]
50             else:
51                 if type(value) is not list:
52                     kwargs[key] = [value]
53
54         keys = kwargs.keys()
55         partial_dicts = [{}]
56         for key in keys:
57             partial_dicts_new = []
58             for partial_dict in partial_dicts:
59                 for value in kwargs[key]: # making a new partial dict for each value
60                     partial_dict_new = partial_dict.copy()
61                     partial_dict_new[key] = value
62                     partial_dicts_new += [partial_dict_new]
63             partial_dicts = partial_dicts_new
64         return partial_dicts
65
66     def polyak(x0, f, f_star, eps, iters):
67         dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]
68
69         for _ in range(iters):
70             fdif = f(*x) - f_star
71             df_squared_sum = np.sum(np.array([df(*x)**2 for df in dfs]))
72             alpha = fdif / (df_squared_sum + eps)
73             x = x - alpha * np.array([df(*x) for df in dfs])
74
75             X += [x] ; Y += [f(*x)]
76         return X, Y
77
78     Polyak = OptimisationAlgorithm(algorithm=polyak,
79                                     algorithm_name="Polyak")

```



```

80
81 def constant_step(x0, alpha, f, iters):
82     dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]
83
84     for _ in range(iters):
85         step = alpha * np.array([df(*x) for df in dfs])
86         x = x - step
87
88         X += [x] ; Y += [f(*x)]
89     return X, Y
90
91 ConstantStep = OptimisationAlgorithm(algorithm=constant_step,
92                                     algorithm_name="Constant")
93
94 def adagrad(x0, f, alpha0, eps, iters):
95     dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]
96
97     df_vector_sum = np.zeros(len(dfs))
98     for _ in range(iters):
99         df_vec = np.array([df(*x) for df in dfs])
100         df_vector_sum += df_vec**2
101         alphas = alpha0 / (np.sqrt(df_vector_sum) + eps)
102         x = x - (alphas * df_vec)
103
104         X += [x] ; Y += [f(*x)]
105     return X, Y
106
107 Adagrad = OptimisationAlgorithm(algorithm=adagrad,
108                                algorithm_name="Adagrad")
109
110 def rmsprop(x0, f, alpha0, beta, eps, iters):
111     dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]
112
113     sum = np.zeros(len(dfs)) ; alpha = alpha0
114     for _ in range(iters):
115         x = x - (alpha * np.array([df(*x) for df in dfs]))
116         sum = beta * sum + (1 - beta) * np.array([df(*x)**2 for df in dfs])
117         alpha = alpha0 / (np.sqrt(sum) + eps)
118
119         X += [x] ; Y += [f(*x)]
120     return X, Y
121
122 RMSProp = OptimisationAlgorithm(algorithm=rmsprop,
123                                algorithm_name="RMSProp")
124
125
126 def heavy_ball(x0, f, alpha, beta, iters):
127     dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]
128
129     z = np.zeros(len(dfs))
130     for _ in range(iters):
131         z = beta * z + alpha * np.array([df(*x) for df in dfs])
132         x = x - z
133
134         X += [x] ; Y += [f(*x)]
135     return X, Y
136
137 HeavyBall = OptimisationAlgorithm(algorithm=heavy_ball,
138                                  algorithm_name="Heavy Ball")
139
140 def adam(x0, f, eps, beta1, beta2, alpha, iters):
141     dfs = f.partial_derivatives ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]
142
143     m = np.zeros(len(dfs)) ; v = np.zeros(len(dfs))
144     for k in range(iters):
145         i = k + 1
146         m = beta1 * m + (1 - beta1) * np.array([df(*x) for df in dfs])
147         v = beta2 * v + (1 - beta2) * np.array([(df(*x)**2) for df in dfs])

```

```

148         mhat = (m / (1 - beta1**i))
149         vhat = (v / (1 - beta2**i))
150         x = x - alpha * (mhat / (np.sqrt(vhat) + eps))
151
152         X += [x] ; Y += [f(*x)]
153     return X,Y
154
155 Adam = OptimisationAlgorithm(algorithm=adam,
156                             algorithm_name="Adam")

```

```

1 # Each record should contain its label depending on what are the other records in the
   list.
2
3 # The user semi-manually inputs what the title should be.
4 # - Have utility functions to extract pieces of the title from the list of records.
5
6 # Function that takes in a list of records.
7 # - For each record determines the label based on what is in the list of records.
8
9 # Perhaps there should be a function that calculates the meta information that is used
   by both
10 # - utility functions that extract pieces of title
11 # - function that assigns the labels to each individual record
12
13
14 # MetaInfo: extracts:
15 # - Which optimisation functions there are
16 # - For each optimisation function
17 #   - What are the parameters that are not varying and what values do they have
18 #   - What are the parameters that are varying and what values do they have
19
20
21
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         title += f' {p}={r[p]}'
82     return title
83
84 def create_labels(records):
85     m = meta(records)
86     for r in records:
87         r['label'] = create_label(r, m)
88
89 # e.g: Adam      beta1=0.2  beta2=0.4
90 def create_label(record, meta):
91     alg_name = record['algorithm'].algorithm_name
92     differing_fields = meta[alg_name]
93     label = f'{alg_name}'
94     for f in differing_fields:
95         label += f' {f}={record[f]}'
96     return label
97
98 # {
99 #   "Adam"      : ["eps", "beta1"]
100 #   "RMSProp"   : ["eps", "alpha0"]
101 # }
102 def meta(records):
103     mr = {}
104     algs = get_algs(records)
105     for a in algs:
106         a_records = alg(records, a)
107         mr[a] = differing_fields(a_records)
108     return mr
109
110 def differing_fields(records):
111     diff_fields = set({})
112     t = records[0]
113     for r in records:
114         for key, value in r.items():
115             # print("a")
116             # print(t[key])
117             # print(type(value))
118             # print(isinstance(value, list))
119
120             if isinstance(value, list):
121                 value = np.array(value)
122             if isinstance(t[key], list):
123                 t[key] = np.array(t[key])
124
125         b = t[key] == value

```

```

126         # print(b)
127         # print(type(b))
128         if type(b) == np.ndarray:
129             b = b.all()
130         if not (b):
131             diff_fields.add(key)
132
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 #   - Parital 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 OptimisableFunction:
11     def __init__(self, sympy_function, sympy_symbols, function_name):
12         self.sympy_symbols = sympy_symbols
13         self.function_name = function_name
14
15         self.sympy_function = sympy_function
16         self.function = lambdify(sympy_symbols, sympy_function, modules="numpy")
17
18         self.sympy_partial_derivatives = [sympy_function.diff(symbol) for symbol in
19 sympy_symbols]
20         self.partial_derivatives = [lambdify(sympy_symbols, p, modules="numpy") for p
21 in self.sympy_partial_derivatives]
22
23     def __parameters_string(self):
24         s = map(latex, self.sympy_symbols)
25         return ",".join(s)
26
27     def latex(self):
28         return self.function_name + "(" + self.__parameters_string() + ") = " + latex
29 (simplify(self.sympy_function))
30
31     def partials_latex(self):
32         s = map(latex, self.sympy_symbols)
33         z = zip(self.sympy_partial_derivatives, s)
34         return [ "\\frac{\\partial " + self.function_name + "\\}{\\partial " +
35 partial_wrt_name + "}" "=" + latex(simplify(partial))
36               for (partial, partial_wrt_name) in z]
37
38     def print_partials_latex(self):
39         for p in self.partials_latex():
40             print(p)

```

```

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 def plot_contour(records, x1r, x2r, log=False):
14     create_labels(records)
15     t = get_titles(records)
16
17     f = records[0]['f'].function;
18     f_name = records[0]['f'].function_name;
19     f_latex = records[0]['f'].latex()
20
21     X1, X2 = np.meshgrid(x1r, x2r)
22     Z = np.vectorize(f)(X1, X2)
23
24     if log:
25         plt.contourf(X1, X2, Z, locator=LogLocator(), cmap='RdGy')
26     else:
27         plt.contourf(X1, X2, Z, cmap='RdGy')
28     xlim = plt.xlim()
29     ylim = plt.ylim()
30
31     for (X, label) in dicts_collect(("X", "label"), records):
32         plt.plot(X.T[0], X.T[1], linewidth=2.0, label=label)
33
34     plt.xlabel(r'$x_1$')
35     plt.ylabel(r'$x_2$')
36
37     title = rf'${f_latex}$' + " \n " + title_string(records)
38     plt.title(title)
39
40     plt.xlim(xlim)
41     plt.ylim(ylim)
42     plt.legend()
43     plt.colorbar()
44
45 def plot_path(records, xr):
46     create_labels(records)
47     f = records[0]['f'].function;
48     function_name = records[0]['f'].function_name
49     f_latex = records[0]['f'].latex()
50
51     yr = [f(x) for x in xr]
52     plt.plot(xr, yr)
53     xlim = plt.xlim()
54     ylim = plt.ylim()
55
56     for (X, label) in dicts_collect(("X", "label"), records):
57         xs = X.flatten()
58         ys = [f(x) for x in xs]
59         plt.plot(xs, ys, linewidth=2.0, label=label)
60
61     plt.xlim(xlim)
62     plt.ylim(ylim)
63     plt.legend()
64     title = rf'${f_latex}$' + "\n" + title_string(records)
65     plt.title(title)
66     plt.ylabel(f'${function_name}$')
67     plt.xlabel(r'$x$')
68

```

```

69 def plot_step_size(records, mean=True):
70     create_labels(records)
71     fig, ax = plt.subplots()
72     f_latex = records[0]['f'].latex()
73     for (X, label) in dicts_collect(("X", "label"), records):
74         if mean:
75             s = np.array([np.mean(x) for x in step_sizes(X).T])
76             ax.plot(np.arange(1, len(s)+1), s, linewidth=2.0, label=label)
77         else:
78             sX = step_sizes(X)
79             for i in range(len(sX)):
80                 x = i + 1
81                 s = sX[i]
82                 ax.plot(np.arange(1, len(s)+1), s, linewidth=2.0, label=label + f'
83 $x_{x}$ step$')
84     ax.legend()
85
86     title = rf'${f_latex}$' + " \n " + title_string(records)
87     if mean:
88         ax.set_title("Mean Step Across x's \n" + title)
89     else:
90         ax.set_title("Mean Step Across x's \n" + title)
91     ax.set_ylabel(f'Step Size')
92     ax.set_xlabel(r'$i$')
93
94 def title_string(records):
95     title = ""
96     t = get_titles(records)
97     for _, v in t.items():
98         title += v + '\n'
99     return title
100
101 # [[x11 x21 x31 ...] [x12 x22 x32 ...] ...] -> [[x12-x11 x13-x12 ...] [x22-x21 x23-
102 x22 ...] ...]
103 def step_sizes(X):
104     return np.array([(x[1:] - x[:-1]) for x in X.T])
105
106
107 def ploty(records):
108     create_labels(records)
109     t = get_titles(records)
110     f_latex = records[0]['f'].latex()
111
112     fig, ax = plt.subplots()
113     for (X, Y, label) in dicts_collect(("X", "Y", "label"), records):
114         ax.plot(range(len(Y)), Y, linewidth=2.0, label=label)
115
116     f = records[0]['f']
117     function_name = f.function_name
118
119     title = rf'${f_latex}$' + " \n " + title_string(records)
120
121
122     ax.set_title(title)
123
124     ax.set_ylabel(f'${function_name}$')
125     ax.set_xlabel(r'$i$')
126     ax.legend()
127     return ax
128
129 def dicts_collect(keys, dicts):
130     values = []
131     for dict in dicts:
132         values += [[dict[key] for key in keys]]
133     return values

```