Optimisation Algorithms - Week 2 Assignment

Ernests Kuznecovs - 17332791 - kuznecoe@tcd.ie

16th February 2022

Contents

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
  - δ: 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.

```python
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 symoblic 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.

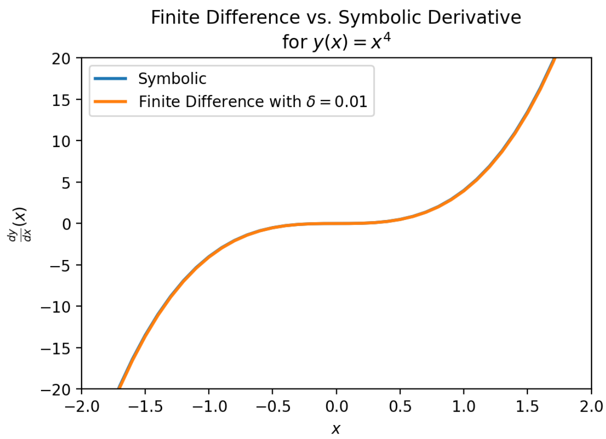

Figure 1

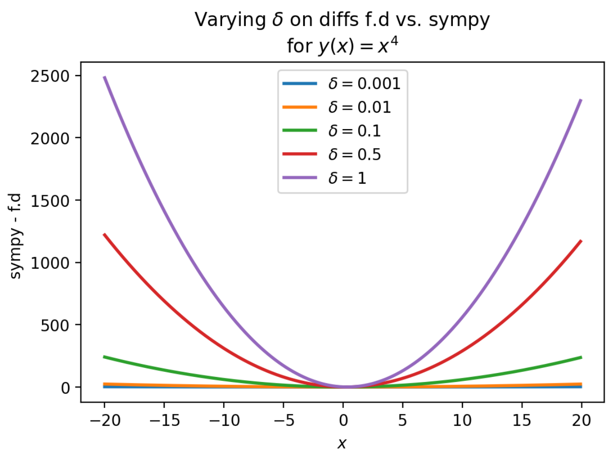

Figure 2

1.3 (iii) Varying δ on Finite Difference

The difference between symbolic derivative and finite difference is plotted. In Fig 2, as δ increases, we can see the error getting bigger an 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 $argmin_x f(x)$.

- 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 minumum of $f(x)$.
- Approximating is acomplished 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 α, in this case α 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 $f(x_i)$. $x_{i+1} = x_i + step$, 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.

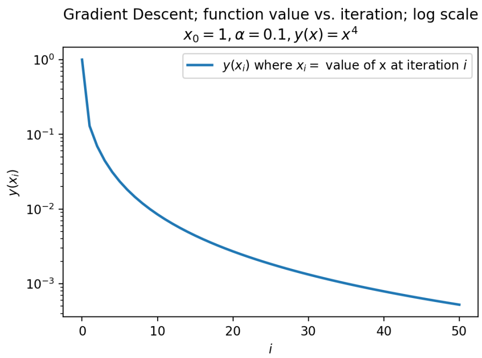

Figure 3: $y(x_i)$ on log scale

- 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 $argmin_x f(x) = 0$ 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 rapdily, 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 α, and the slope is important in the step as $step = \alpha * slope_{x_i}$.
- 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.

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 a 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.

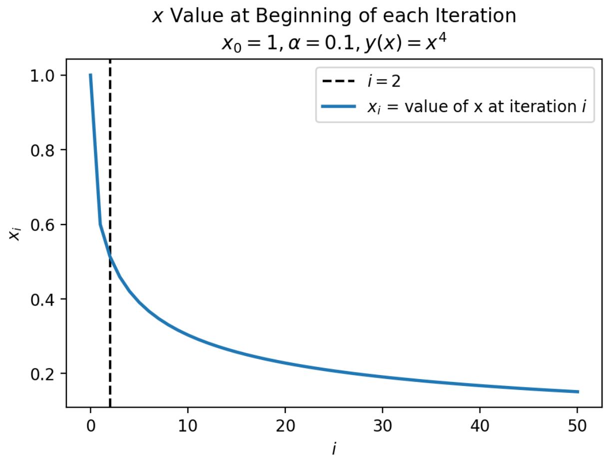

Figure 4

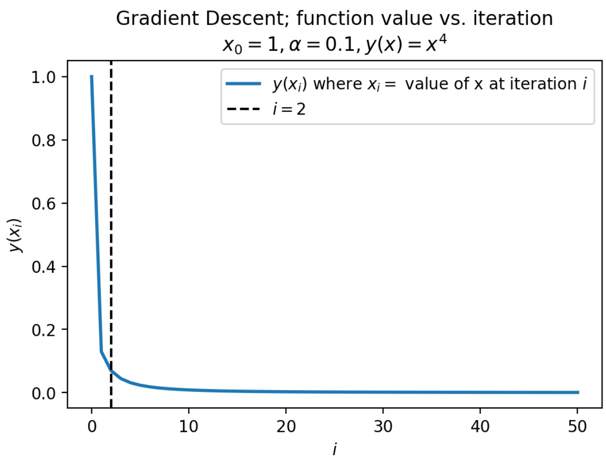

Figure 5

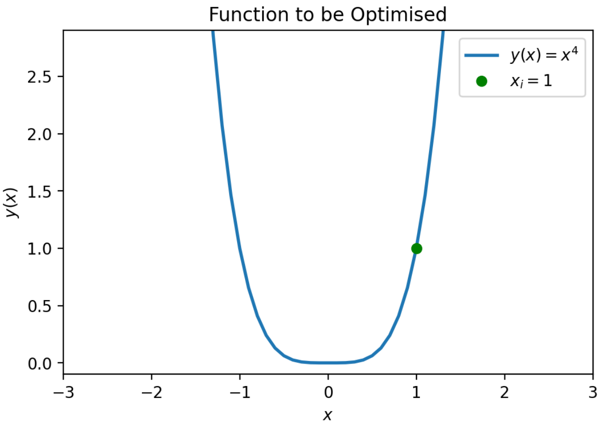

Figure 6

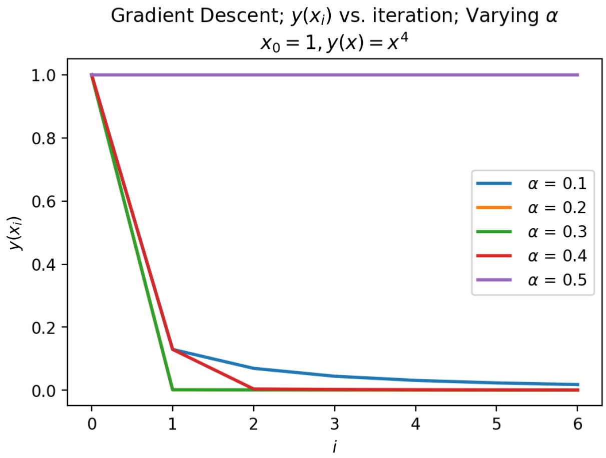

Figure 7

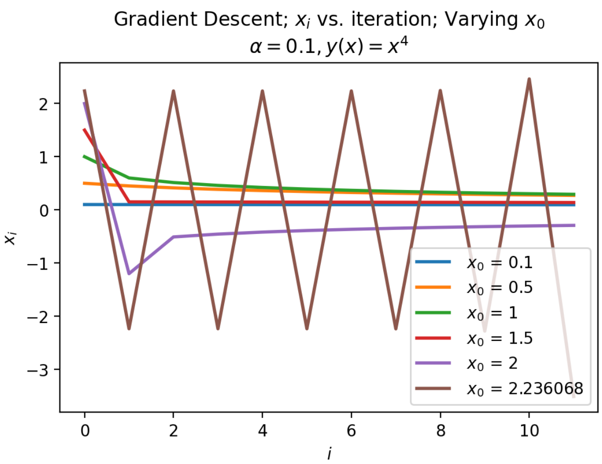

Figure 8

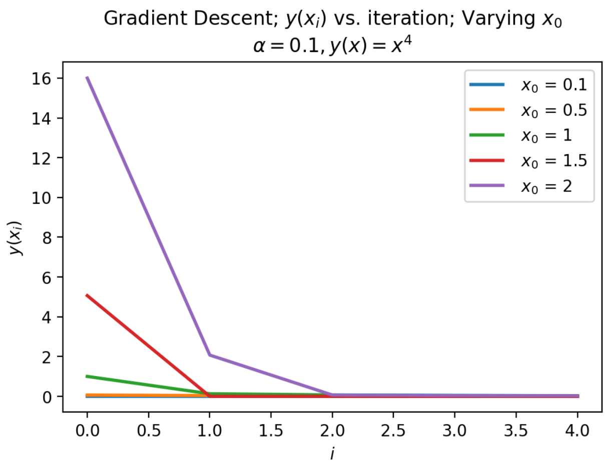

Figure 9

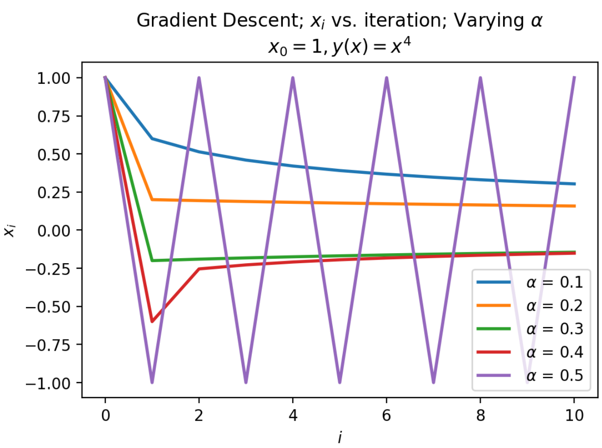

Figure 10

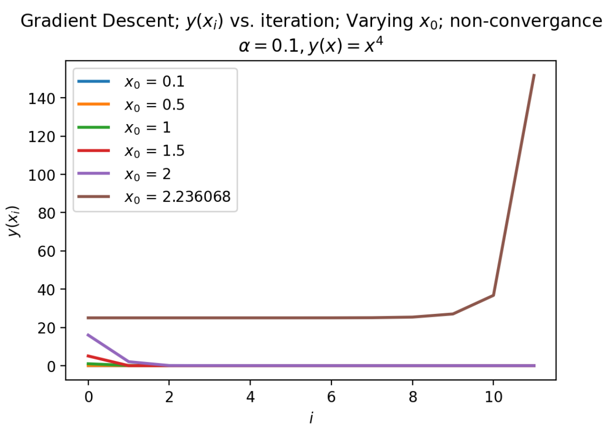

Figure 11

2.3 (iii) Varying Step Size α and x_0

- Varying x_0
  - Plotting x - Fig. 8 : We can see that $x_0 > 2.236068$ would lead to an explosive non-convergance; 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 α, and essentially stops making progress.
  - Plotting y that converge - Fig. 9 : We see that even though x_i dont converge on the same point for different x_0, they all converge on paractically 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 α
  - 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 α 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-convergance, 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 α that causes the slowdown of the convergance.

Both x_0 and α 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 α 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 wont 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 gammas 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 convergance 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 convergance 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 γ. Though α may need to 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.

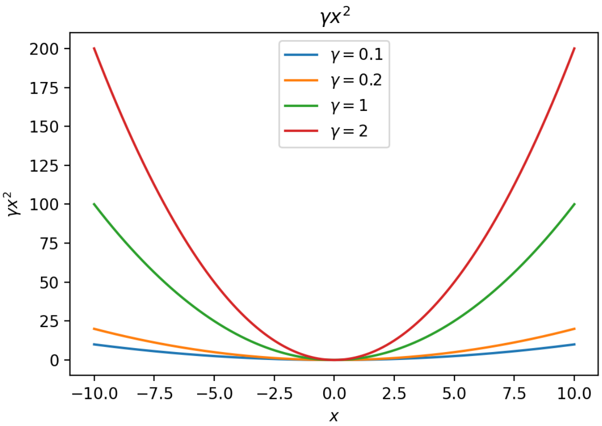

Figure 12

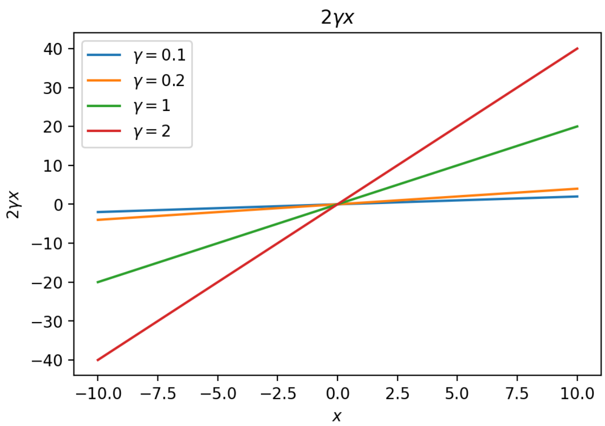

Figure 13

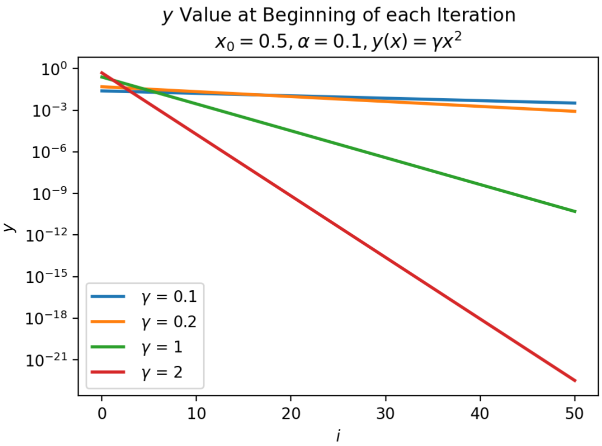

Figure 14

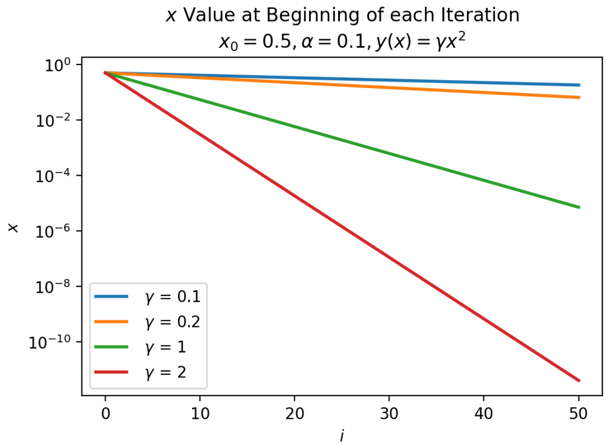

Figure 15

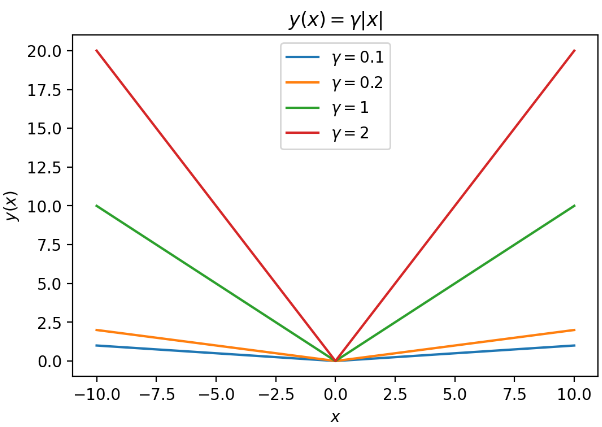

Figure 16

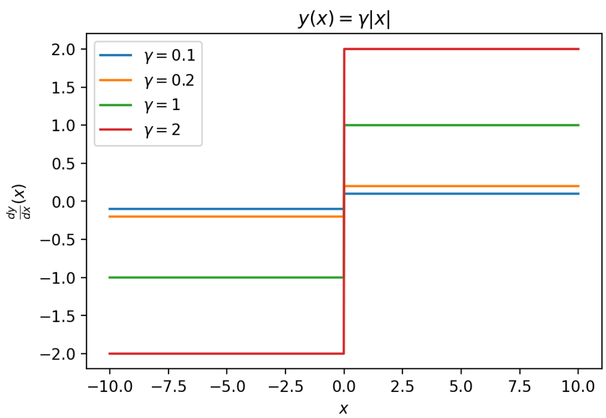

Figure 17

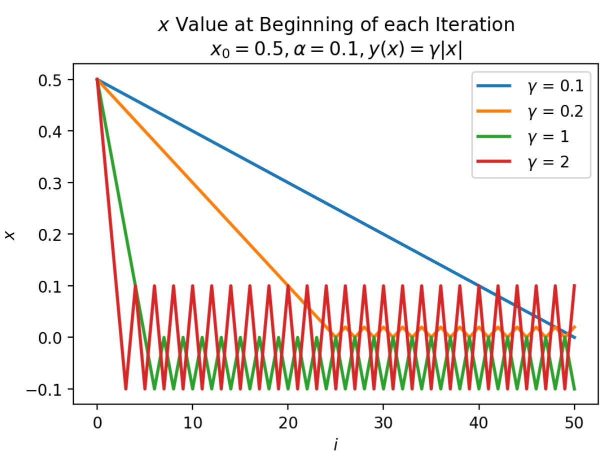

Figure 18

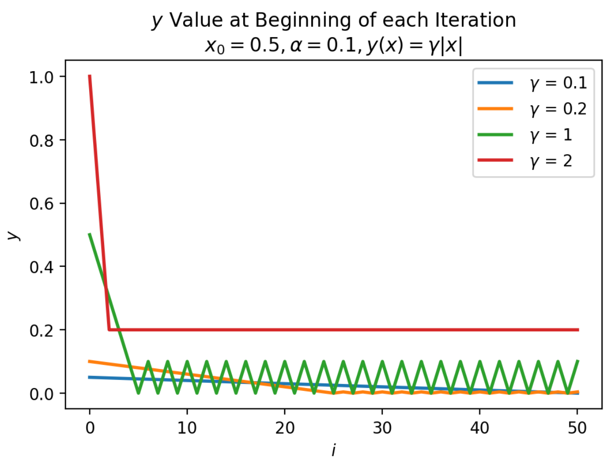

Figure 19

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, $\gamma = +/- \ slope$.

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/zigzaging in a loop for the 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 minumum. 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 thats 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 oscilate because its actually jumping between x values of the same magnitude and hence same function value.

4 Appendix

4.1 Code Listing

```
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 200
mpl.rcParams['figure.facecolor'] = '1'
import matplotlib.pyplot as plt

import numpy as np
import sympy

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

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

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

def finiteDiff(f, x, delta):
    return (f(x + delta) - f(x - delta)) / (2 * delta)
```

```python

def axset(ax, xrange, xoffset, yrange, yoffset):
    ax.set(xlim=(xoffset-xrange, xoffset+xrange),
           ylim=(yoffset-yrange, yoffset+yrange))

xs = np.arange(-20, 20, 0.1)

ys_sym = dydx(xs)

ys_finiteDiff = []
for x in xs:
    ys_finiteDiff.append(finiteDiff(y, x, 0.01))

fig, ax = plt.subplots()
ax.set_ylabel(r'$\frac{dy}{dx}(x)$')
ax.set_xlabel(r'$x$')
ax.set_title(r'Finite Difference vs. Symbolic Derivative'  "\n" r'for $y(x) = x^4$')

ax.plot(xs, ys_sym, linewidth=2.0)
ax.plot(xs, ys_finiteDiff, linewidth=2.0)
ax.legend(("Symbolic", r'Finite Difference with $\delta = 0.01$'))
axset(ax, xrange=2, xoffset=0, yrange=20, yoffset=0)

# fig.show()

# ax.set(
#     xlim=(-3, 3),
#     ylim=(-20, 20),
#     xticks=np.arange(1, 8),
#     yticks=np.arange(1, 8),
#      )

dydx = lambda x: 4 * x**3
y = lambda x: x**4

xs = np.arange(-20, 20, 0.1)

deltas = [0.001, 0.01, 0.1, 0.5, 1]
ys_dif = []
for delta in deltas:
  dif = []
  for x in xs:
      fd = finiteDiff(y, x, delta)
      ex = dydx(x)
      dif += [ex - fd]

  ys_dif += [(dif, delta)]

fig, ax = plt.subplots()
legend_labels = []
for (diff, delta) in ys_dif:
    legend_labels += [r'$\delta = $' + str(delta)]
    ax.plot(xs, diff, linewidth=2.0)

ax.set_title(r'Varying $\delta$ on diffs f.d vs. sympy'  "\n" r'for $y(x) = x^4$')
ax.set_ylabel(r'sympy - f.d ')
ax.set_xlabel(r'$x$')
ax.legend(legend_labels)
# axset(ax, xrange=3, xoffset=1.5, yrange=20, yoffset=10)

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

class QuadraticFn():
```

```
    def f(self, x):
        return x**2                          # function value f(x)

    def df(self, x):
        return x*2                           # derivative of f(x)

fn = QuadraticFn()

def gradDesc(fn, x0, alpha=0.15, num_iters=50):
    x = x0                                   # starting point
    X = np.array([x])                        # array of x history
    F = np.array(fn.f(x))                    # array of f(x) history
    for k in range(num_iters):
        step = alpha * fn.df(x)
        x = x - step
        X = np.append(X, [x], axis=0)        # add current x to history
        F = np.append(F, fn.f(x))            # add value of current f(x) to history
    return (X,F)

def gradDesc3(f, df, x0, alpha=0.15, num_iters=50):
    x = x0                                   # starting point
    X = np.array([x])                        # array of x history
    F = np.array(f(x))                    # array of f(x) history
    for k in range(num_iters):
        step = alpha * df(x)
        x = x - step
        # print(x)
        X = np.append(X, [x], axis=0)        # add current x to history
        F = np.append(F, f(x))            # add value of current f(x) to history
    return (X,F)

(X, F) = gradDesc(fn, 1)
x = gradient_descent(fn.df, 1)

xs = np.arange(-20, 20, 0.1)

ys = dydx(xs)
ys = y(xs)

fig, ax = plt.subplots()
ax.set_ylabel(r'$y(x)$')
ax.set_xlabel(r'$x$')

ax.set_title(r'Function to be Optimised')
ax.plot(xs, ys, linewidth=2.0)
ax.plot(1, y(1), 'go')
ax.legend(("$y(x) = x^4$", r'$x_i = 1$'))

# ax.axvline(x=1, color='k', linestyle='--')
axset(ax, xrange=3, xoffset=0, yrange=1.5, yoffset=1.4)

(_, F) = gradDesc3(y, dydx, x0=1, alpha=0.1)
iters = np.arange(0, len(F))

fig, ax = plt.subplots()
ax.set_ylabel(r'$y(x_{i})$')
ax.set_xlabel(r'$i$')
ax.set_title(r'Gradient Descent; function value vs. iteration' "\n"
             r'$x_0=1, \alpha=0.1 , y(x) = x^4$',)
ax.plot(iters, F, linewidth=2.0)
ax.axvline(x=2, color='k', linestyle='--')

ax.legend((r'$y(x_{i})$ where $x_i=$ value of x at iteration $i$', r'$i=2$', ))

(_, F) = gradDesc3(y, dydx, x0=1, alpha=0.1)
iters = np.arange(0, len(F))

fig, ax = plt.subplots()
```

```
ax.set_ylabel(r'$y(x_{i})$')
ax.set_xlabel(r'$i$')
ax.set_title(r'Gradient Descent; function value vs. iteration; log scale' "\n"
             r'$x_0=1, \alpha=0.1 , y(x) = x^4$',)

ax.semilogy(iters, F, linewidth=2.0)
ax.legend((r'$y(x_{i})$ where $x_i=$ value of x at iteration $i$',))

(X, _) = gradDesc3(lambda x : x**4, lambda x : 4*x**3, x0=1, alpha=0.1)
iters = np.arange(0, len(X))

fig, ax = plt.subplots()
ax.set_ylabel(r'$x_i$')
ax.set_xlabel(r'$i$')
ax.set_title(r'$x$ Value at Beginning of each Iteration' "\n"
            r'$x_0=1, \alpha=0.1 , y(x) = x^4$',)
ax.axvline(x=2, color='k', linestyle='--')
ax.plot(iters, X, linewidth=2.0)

ax.legend((r'$i=2$', r'$x_{i}$ = value of x at iteration $i$',))

# (X, _) = gradDesc3(y, dydx, x0=1, alpha=0.1)   # given a range of alphas, give back
    corresponding dimensions of answers, same for x0s
# perhaps it gives back objects that describe the shape of the output in detail,
   perhaps what dimension represents what, and how many there are

x0s = np.arange(0.1, 2, 0.1)
num_iters = 50

Xs = np.array([])
for x0 in x0s:
    (X, _) = gradDesc3(lambda x : x**4, lambda x : 4*x**3, x0=x0, alpha=0.1,
   num_iters=num_iters)
    if len(Xs) > 0:
        Xs = np.append(Xs, [X],  axis=0)
    else:
        Xs = np.array([X])

# fig, ax = plt.subplots()
# ax.set_ylabel(r'$x_i$')
# ax.set_xlabel(r'$i$')
# print(num_iters)

# print(Xs.shape)
# 0th index is x0 = 1.7
# [0,0] (x0=0.1,i=0)
# [0,1] (x0=0.1,i=1) 2 params input, Xs is the output

# [1,0] (x0=0.2,i=1)
# [1,1] (x0=0.2,i=1) 2 params input, Xs is the output

# indexes of inputs must correspond to position of output

itersY, x0sX = np.meshgrid(np.arange(num_iters+1), x0s)
# print(x0sX)
# print(itersY)
# print(Xs)

fig = plt.figure()
ax = plt.axes(projection='3d')
# ax.contour3D(x0sX, itersY, Xs, 100, cmap='binary')
ax.plot_surface(x0sX, itersY, Xs, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.view_init(12, 75)
# ax.view_init(12, 120)
ax.view_init(12, 30)
# ax.view_init(0, 0)
```

```
ax.set_xlabel(r'$x_0$')
ax.set_ylabel(r'$i$')
ax.set_zlabel(r'$x_i$')

# looks like i get slow on these kinds of problems
# probably practice will help
# and perhaps doing going slowly through them and
# understanding them will help

x0s = [0.1, 0.5, 1, 1.5, 2, 2.236068]
# 2.23607
num_iters = 11

Xs = np.array([])
for x0 in x0s:
    (X, _) = gradDesc3(lambda x : x**4,
                       lambda x : 4*x**3,
                       x0=x0,
                       alpha=0.1,
                       num_iters=num_iters)
    if len(Xs) > 0:
        Xs = np.append(Xs, [(X,x0)],  axis=0)
    else:
        Xs = np.array([(X, x0)])

fig, ax = plt.subplots()
ax.set_ylabel(r'$x_i$')
ax.set_xlabel(r'$i$')
ax.set_title(r'Gradient Descent; $x_i$ vs. iteration; Varying $x_0$' "\n"
              r'$ \alpha=0.1 , y(x) = x^4$',)
legend_labels = []
for (X, x0) in Xs:
    ax.plot(range(num_iters+1), X, linewidth=2.0)
    legend_labels += [(r' $x_{0}$ = ' + str(x0))]
ax.legend(legend_labels)

alphas = [0.1, 0.2, 0.3, 0.4, 0.5]
num_iters = 10

Xs = np.array([])
for alpha in alphas:
    (X, _) = gradDesc3(lambda x : x**4,
                       lambda x : 4*x**3,
                       x0=1,
                       alpha=alpha,
                       num_iters=num_iters)
    if len(Xs) > 0:
        Xs = np.append(Xs, [(X,alpha)],  axis=0)
    else:
        Xs = np.array([(X, alpha)])

fig, ax = plt.subplots()
ax.set_ylabel(r'$x_i$')
ax.set_xlabel(r'$i$')
ax.set_title(r'Gradient Descent; $x_i$ vs. iteration; Varying $\alpha$' "\n"
             r'$ x_0=1 , y(x) = x^4$',)
legend_labels = []
for (X, alpha) in Xs:
    ax.plot(range(num_iters+1), X, linewidth=2.0)
    legend_labels += [(r' $\alpha$ = ' + str(alpha))]
ax.legend(legend_labels)

x0s = [0.1, 0.5, 1, 1.5, 2]
num_iters = 4

Ys = np.array([])
for x0 in x0s:
    (_, Y) = gradDesc3(lambda x : x**4,
```

```
                           lambda x : 4*x**3,
                           x0=x0,
                           alpha=0.1,
                           num_iters=num_iters)
    if len(Ys) > 0:
        Ys = np.append(Ys, [(Y,x0)],  axis=0)
    else:
        Ys = np.array([(Y, x0)])

fig, ax = plt.subplots()
ax.set_ylabel(r'$y(x_i)$')
ax.set_xlabel(r'$i$')
ax.set_title(r'Gradient Descent; $y(x_i)$ vs. iteration; Varying $x_0$' "\n"
             r'$ \alpha=0.1 , y(x) = x^4$',)
legend_labels = []
for (Y, x0) in Ys:
    ax.plot(range(num_iters+1), Y, linewidth=2.0)
    legend_labels += [(r' $x_{0}$ = ' + str(x0))]
ax.legend(legend_labels)

x0s = [0.1, 0.5, 1, 1.5, 2, 2.236068]
num_iters = 11

Ys = np.array([])
for x0 in x0s:
    (_, Y) = gradDesc3(lambda x : x**4,
                           lambda x : 4*x**3,
                           x0=x0,
                           alpha=0.1,
                           num_iters=num_iters)
    if len(Ys) > 0:
        Ys = np.append(Ys, [(Y,x0)],  axis=0)
    else:
        Ys = np.array([(Y, x0)])

fig, ax = plt.subplots()
ax.set_ylabel(r'$y(x_i)$')
ax.set_xlabel(r'$i$')
ax.set_title(r'Gradient Descent; $y(x_i)$ vs. iteration; Varying $x_0$; non-
    convergance' "\n"
             r'$ \alpha=0.1 , y(x) = x^4$',)
legend_labels = []
for (Y, x0) in Ys:
    ax.plot(range(num_iters+1), Y, linewidth=2.0)
    legend_labels += [(r' $x_{0}$ = ' + str(x0))]
ax.legend(legend_labels)

x0s = [0.1, 0.5, 1, 1.5, 2]
num_iters = 12

Ys = np.array([])
for x0 in x0s:
    (_, Y) = gradDesc3(lambda x : x**4,
                           lambda x : 4*x**3,
                           x0=x0,
                           alpha=0.1,
                           num_iters=num_iters)
    if len(Ys) > 0:
        Ys = np.append(Ys, [(Y,x0)],  axis=0)
    else:
        Ys = np.array([(Y, x0)])

fig, ax = plt.subplots()
ax.set_ylabel(r'$x_i$')
ax.set_xlabel(r'$i$')
legend_labels = []
for (Y, x0) in Ys:
    ax.semilogy(range(num_iters+1), Y, linewidth=2.0)
```

```python
    legend_labels += [(r' $x_{0}$ = ' + str(x0))]
ax.legend(legend_labels)

alphas = [0.1, 0.2, 0.3, 0.4, 0.5]
num_iters = 6

Ys = np.array([])
for alpha in alphas:
    (_, Y) = gradDesc3(lambda x : x**4,
                       lambda x : 4*x**3,
                       x0=1,
                       alpha=alpha,
                       num_iters=num_iters)
    if len(Ys) > 0:
        Ys = np.append(Ys, [(Y,alpha)],  axis=0)
    else:
        Ys = np.array([(Y, alpha)])

fig, ax = plt.subplots()
ax.set_ylabel(r'$y(x_i)$')
ax.set_xlabel(r'$i$')
ax.set_title(r'Gradient Descent; $y(x_i)$ vs. iteration; Varying $\alpha$' "\n"
             r'$ x_0=1 , y(x) = x^4$',)
legend_labels = []
for (Y, alpha) in Ys:
    ax.plot(range(num_iters+1), Y, linewidth=2.0)
    legend_labels += [(r' $\alpha$ = ' + str(alpha))]
ax.legend(legend_labels)

from jax import grad
y = lambda x, gamma: gamma * x**2

# grad by default will take the derivative of the first parameter of the function that we pass
dydx = grad(y)

def visualise_fn(fn, l=-10, r=10, n=1000):
    xs = np.linspace(l, r, num=n)
    y = np.array([fn(x) for x in xs])
    plt.plot(xs,y)

def labels_fn(ax, legend, xaxis=r'$x$', yaxis=r'$y(x)$', title="Title"):
    ax.set_xlabel(xaxis)
    ax.set_ylabel(yaxis)
    ax.set_title(title)
    ax.legend(legend)

def visualise_fns(fns, labels_fn=labels_fn, l=-10, r=10, n=1000):
    xs = np.linspace(l, r, num=n)
    ys = []
    fig, ax = plt.subplots()
    for fn in fns:
        y = np.array([fn(x) for x in xs])
        ax.plot(xs,y)
    labels_fn(ax)

fns_gamma = (lambda fn, gammas: [(lambda x, gamma=gamma: fn(x, gamma)) for gamma in gammas])

gammas = [0.1, 0.2, 1, 2]
legend = [(r'$\gamma=$'+ str(gamma)) for gamma in gammas]
labels_y = lambda ax: labels_fn(ax, legend, yaxis=r'$\gamma x^2$', title=r'$\gamma x^2$')
labels_dy = lambda ax: labels_fn(ax, legend, yaxis=r'$2\gamma x$', title=r'$2 \gamma x$' )

visualise_fns(fns_gamma(dydx, gammas), labels_fn=labels_dy)

```

```
visualise_fns(fns_gamma(y, gammas), labels_fn=labels_y)

def gamma_grad(gamma, num_iters=40, x0=1, alpha=0.1):
    return gradDesc3(f=lambda x : y(x, gamma),
                     df=(lambda x : dydx(x, gamma)),
                     x0=x0,
                     alpha=0.1,
                     num_iters=num_iters)

gammas = [0.1, 0.2, 1, 2]
num_iters=50
Ys = np.array([])
# wonder how can generalise this for future ease of use
for gamma in gammas:
    (_, Y) = gamma_grad(gamma, num_iters=num_iters, x0=0.5, alpha=0.1)
    if len(Ys) > 0:
        Ys = np.append(Ys, [(Y,gamma)],  axis=0)
    else:
        Ys = np.array([(Y, gamma)])

fig, ax = plt.subplots()
legend_labels = []
for (Y, gamma) in Ys:
    # ax.plot(range(num_iters+1), Y, linewidth=2.0)
    ax.semilogy(range(num_iters+1), Y, linewidth=2.0)
    legend_labels += [(r' $\gamma$ = ' + str(gamma))]
ax.legend(legend_labels)
ax.set_ylabel(r'$y$')
ax.set_xlabel(r'$i$')
ax.set_title(r'$y$ Value at Beginning of each Iteration' "\n"
             r'$x_0=0.5, \alpha=0.1 , y(x) = \gamma x^2$',)

gammas = [0.1, 0.2, 1, 2]
num_iters=50
Xs = np.array([])
# wonder how can generalise this for future ease of use
for gamma in gammas:
    (X, _) = gamma_grad(gamma, num_iters=num_iters, x0=0.5, alpha=0.1)
    if len(Xs) > 0:
        Xs = np.append(Xs, [(X,gamma)],  axis=0)
    else:
        Xs = np.array([(X, gamma)])

fig, ax = plt.subplots()
legend_labels = []
for (X, gamma) in Xs:
    # ax.plot(range(num_iters+1), X, linewidth=2.0)
    ax.semilogy(range(num_iters+1), X, linewidth=2.0)
    legend_labels += [(r' $\gamma$ = ' + str(gamma))]
ax.legend(legend_labels)
ax.set_ylabel(r'$x$')
ax.set_xlabel(r'$i$')
ax.set_title(r'$x$ Value at Beginning of each Iteration' "\n"
             r'$x_0=0.5, \alpha=0.1 , y(x) = \gamma x^2$',)

y = lambda x, gamma: gamma * abs(x)
dydx = grad(y)

gammas = [0.1, 0.2, 1, 2]
legend = [(r'$\gamma=$'+ str(gamma)) for gamma in gammas]
labels_y = lambda ax: labels_fn(ax, legend, yaxis=r'$y(x)$', title=r'$y(x) = \gamma |
    x|$')
labels_dy = lambda ax: labels_fn(ax, legend, yaxis=r'$ \frac{dy}{dx}(x) $', title=r'
    $y(x) = \gamma |x|$')

gamma_grad

visualise_fns(fns_gamma(dydx, gammas), labels_fn=labels_dy)
```

```
visualise_fns(fns_gamma(y, gammas), labels_fn=labels_y)

def gamma_grad(gamma, num_iters=40, x0=1, alpha=0.1):
    return gradDesc3(f=lambda x : y(x, gamma),
                     df=(lambda x : dydx(x, gamma)),
                     x0=x0,
                     alpha=0.1,
                     num_iters=num_iters)

gammas = [0.1, 0.2, 1, 2]
num_iters=50
Ys = np.array([])
# wonder how can generalise this for future ease of use
for gamma in gammas:
    (_, Y) = gamma_grad(gamma, num_iters=num_iters, x0=0.5, alpha=0.1)
    if len(Ys) > 0:
        Ys = np.append(Ys, [(Y,gamma)],  axis=0)
    else:
        Ys = np.array([(Y, gamma)])

fig, ax = plt.subplots()
legend_labels = []
for (Y, gamma) in Ys:
    ax.plot(range(num_iters+1), Y, linewidth=2.0)
    # ax.semilogy(range(num_iters+1), Y, linewidth=2.0)
    legend_labels += [(r' $\gamma$ = ' + str(gamma))]
ax.legend(legend_labels)
ax.set_ylabel(r'$y$')
ax.set_xlabel(r'$i$')
ax.set_title(r'$y$ Value at Beginning of each Iteration' "\n"
             r'$x_0=0.5, \alpha=0.1 , y(x) = \gamma x^2$',)

gammas = [0.1, 0.2, 1, 2]
num_iters=50
Xs = np.array([])
# wonder how can generalise this for future ease of use
for gamma in gammas:
    (X, _) = gamma_grad(gamma, num_iters=num_iters, x0=0.5, alpha=0.1)
    if len(Xs) > 0:
        Xs = np.append(Xs, [(X,gamma)],  axis=0)
    else:
        Xs = np.array([(X, gamma)])

fig, ax = plt.subplots()
legend_labels = []
for (X, gamma) in Xs:
    ax.plot(range(num_iters+1), X, linewidth=2.0)
    # ax.semilogy(range(num_iters+1), X, linewidth=2.0)
    legend_labels += [(r' $\gamma$ = ' + str(gamma))]
ax.legend(legend_labels)
ax.set_ylabel(r'$x$')
ax.set_xlabel(r'$i$')
ax.set_title(r'$x$ Value at Beginning of each Iteration' "\n"
             r'$x_0=0.5, \alpha=0.1 , y(x) = \gamma x^2$',)
```

```
# Algorithms.py

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

import numpy as np

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

```python

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

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

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

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

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

```python
        mhat = (m / (1 - beta1**i))
        vhat = (v / (1 - beta2**i))
        x = x - alpha * (mhat / (np.sqrt(vhat) + eps))

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

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

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

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

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

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


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




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

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


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


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

# get_titles returns
# {
#   "Adam" : "Adam: beta1=0.1 eps=0.0001 iters=50 x0=[1, 1]",
```

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

import numpy as np

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

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

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

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

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

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

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

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

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

            b = t[key] == value
```

```
            # print(b)
            # print(type(b))
            if type(b) == np.ndarray:
                b = b.all()
            if not (b):
                diff_fields.add(key)


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

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

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


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

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

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

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

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

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

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

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

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

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

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

from OptimisationAlgorithmToolkit.DataType import create_labels, get_titles

from matplotlib.ticker import LogLocator

import numpy as np

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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


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

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



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

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

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

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


    ax.set_title(title)

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

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