Week 2 Optimisation for Machine Learning

Neimhin Robinson Gunning, 16321701

March 11, 2024

```
funcs.txt          Wed Feb 21 15:03:56 2024          1

function: 3*(x-5)^4+10*(y-9)^2
function: Max(x-5,0)+10*|y-9|
```

Figure 1: Two bivariate functions downloaded from `https://www.scss.tcd.ie/Doug.Leith/CS7DS2/week4.php`

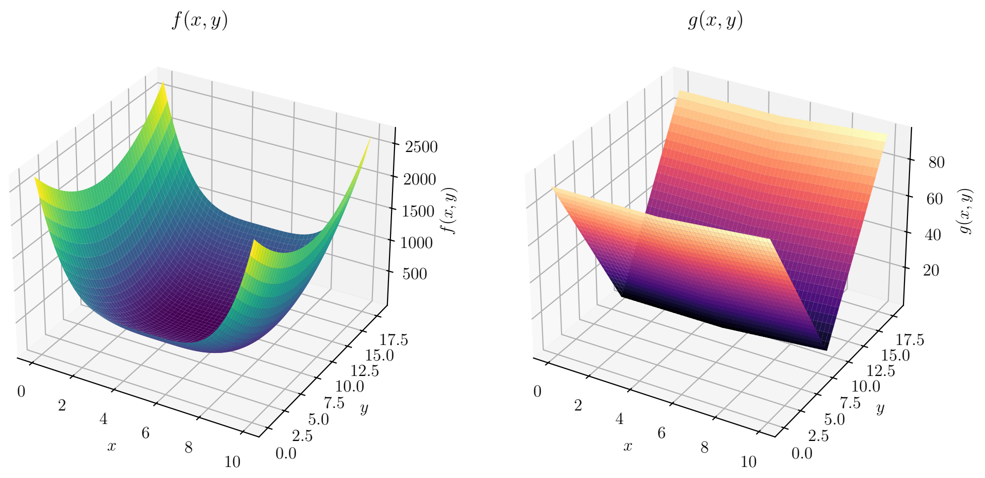

Figure 2

Let

$$f(x,y) = 3(x-5)^4 + 10(y-9)^2 \tag{1}$$

and

$$g(x,y) = \max(x-5,0) + 10|y-9| \tag{2}$$

Using `sympy` we find the derivatives:

$$\nabla f = [\frac{df}{dx}, \frac{df}{dy}] = [12(x-5)^3, 20y-180]$$

$$\nabla g = [\frac{dg}{dx}, \frac{dg}{dy}] = [\text{Heaviside}(x-5), 10\text{sign}(y-9)]$$

Clearly, the minimum of both $f(x,y)$ and $g(x,y)$ is 0 and they are both minimized by $x=5$, $y=9$.

The Polyak step size is

$$\alpha_{\text{Polyak}} = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x)} \tag{3}$$

where x is the parameter vector, $f(x)$ is the function to optimise, and $f^* \approx \min_x f(x)$.

Listing 1: A python function to calculate the Polyak step size on a `sympy` function.

src/polyak_step_size.py Wed Feb 14 15:12:30 2024 1

```
import numpy as np


def polyak_step_size(self, sp_func, sp_x, x, f_star):
    assert len(sp_x) == len(x)
    subs = {sp_xi: xi for sp_xi, xi in zip(sp_x, x)}
    fx = sp_func.subs(subs)
    grad = [sp_func.diff(sp_xi).subs(subs) for sp_xi in sp_x]
    grad = np.array(grad)
    denominator = sum(grad * grad)
    numerator = fx - f_star
    return numerator / denominator
```

```python
import lib
import numpy as np


def iterate(self):
    x_value = self._start
    old_x_value = None
    iteration = 0
    m = np.zeros(x_value.shape, dtype=np.float64)
    v = np.zeros(x_value.shape, dtype=np.float64)
    converged = False
    grad_value = self._gradient(x_value)

    def yielded():
        print(x_value)
        print(iteration)
        return {
            "iteration": iteration,
            "x": x_value,
            "f(x)": self._function(x_value),
            "sum": sum,
            "epsilon": self._epsilon,
            "converged": converged,
            "gradient": grad_value,
            "m": m,
            "v": v,
            "beta1": self._beta,
            "beta2": self._beta2,
            "alpha": self._step_size,
        }

    yield yielded()

    while not converged:
        if self._max_iter > 0 and iteration > self._max_iter:
            break
        grad_value = self._gradient(x_value)
        m = self._beta * m + (1-self._beta)*grad_value
        # grad_value * grad_value gives element-wise product of np array
        v = self._beta2 * v + (1-self._beta2) * (grad_value*grad_value)
        old_x_value = x_value
        iteration += 1
        m_hat = m / (1-(self._beta ** iteration))
        v_hat = np.array(v / (1-(self._beta2 ** iteration)))
        print('v', v, type(v))
        print('v_hat', v_hat, type(v_hat))
        print(np, type(np))
        v_hat_aug = v_hat**(0.5) + self._epsilon
        adam_grad = m_hat / v_hat_aug
        x_value = x_value - self._step_size * adam_grad
        converged = self._converged(x_value, old_x_value)
        yield yielded()


if __name__ == "__main__":
    adam = lib.GradientDescent()
    adam.epsilon(0.0001)
    adam.step_size(10**-2)
    adam.beta(0.8)
    adam.beta2(0.9)
    adam.max_iter(-1)
    adam.start(np.array([0, 0]))

    def converged(x1, x2):
        d = np.max(x1-x2)
        return d < 0.000001

    def fn(x):
        return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])

    def grad(x):
        return np.array(
            [lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
                for var in (lib.x, lib.y)])
    adam.converged(converged)
    adam.function(fn)
    adam.gradient(grad)
    adam.set_iterate(iterate)
    adam.run2csv("adam.csv")
```

```python
from sympy import symbols, diff, solve
import sympy as sp

# Define the symbolic variables
x, y = symbols('x y',real=True)

# Define the functions
f = 3 * (x - 5)**4 + 10 * (y - 9)**2
g = sp.Max(x - 5, 0) + 10 * sp.Abs(y - 9)

grad_f = [diff(f, var) for var in (x, y)]
argmin_f = solve(grad_f, (x, y))
print(f"Argmin of f(x, y): {argmin_f}")

grad_g = [diff(g, var) for var in (x, y)]
argmin_g = solve(grad_g, (x, y))
print(f"Argmin of g(x, y): {argmin_g}")
```

```python
import sympy as sp
import sys
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm
import seaborn as sns
import pandas as pd
from lib import GradientDescent

LINEWIDTH = 0.5
x = sp.symbols('x')
y = x**4
dydx = y.diff()

fig, ax = plt.subplots(1, 3, figsize=(12, 8))

blowup = 0.8

# alpha * (blowup ** 3) * 4 = 1.2

results = {
    "alpha": [],
    "start": [],
    "convergence time": [],
    "final guess": [],
}
iota = 0.00000000001
settings = [
            (0.1, 1),
            (0.03, 1),
            (0.5, 1),
            (0.25, 1),
            ((2*blowup)/((blowup**3)*4) + iota, blowup),
            ((2*blowup)/((blowup**3)*4) - iota, blowup),
            (0.05, 0.7),
            (0.1, 0.7),
            (0.15, 0.7),
            (0.1, 2),
]
color = cm.rainbow(np.linspace(0, 1, len(settings)))
settings_with_color = zip(settings, color)
for ((step_size, start), color) in settings_with_color:
    print(step_size, start, color)
    g = GradientDescent()
    g.max_iter(100)
    g.step_size(step_size)
    g.start(start)
    g.function(lambda x1: float(y.subs(x, x1)))
    y_diff = y.diff()
    g.gradient(lambda x1: float(y_diff.subs(x, x1)))
    g.debug(True)

    def is_inf(x):
        import math
        if x == math.inf or x == -math.inf:
            return True

    def converged(x1, x2):
        if is_inf(x1) or is_inf(x2):
            return True
        abs = np.abs(x1-x2)
        print(abs, x1, x2)
        return abs < 0.001
    g.converged(converged)
    iterations, estimates, y_of_x = zip(*[
        (x[0], x[1], x[2]) for x in g.iterate()])
    results["alpha"].append(step_size)
    results["start"].append(start)
    results["convergence time"].append(len(iterations))
    results["final guess"].append(estimates[-1])
    print('y_of_x', y_of_x)
    print('iterations', iterations)
    print('estimates', estimates)
    sns.lineplot(
        x=iterations,
        y=np.abs(np.array(estimates)),
        ax=ax[0],
        linewidth=LINEWIDTH,
        legend=False,
        color=color,
        label=f"$\\alpha={step_size}$, $x={start}$")
    sns.lineplot(
        x=iterations,
        y=y_of_x,
        ax=ax[1],
        linewidth=LINEWIDTH,
        color=color,
        label=f"$\\alpha={step_size}$, $x={start}$")
    ax[2].step(
        estimates,
        y_of_x,
        linewidth=LINEWIDTH,
        color=color,
        label=f"$\\alpha={step_size}$, $x={start}$")
    xs = np.arange(-2, 2, 0.01)
    ys = [y.subs(x, xi) for xi in xs]
    ax[2].plot(
        xs,
        ys,
        linewidth=LINEWIDTH,
```

```
        label="$x^4$",
        color='yellow',
    )
    ax[2].scatter(
        start,
        g._function(start),
        color=color)

ax[1].legend(framealpha=1)
ax[0].set_ylabel("$|\\hat x|$")
ax[0].set_xlabel("iteration")
ax[0].set_yscale('log')
ax[1].set_yscale('log')
ax[0].set_title("(a)")
ax[1].set_ylabel("$y(\\hat{x})$")
ax[1].set_xlabel("iteration")
ax[1].set_title("(b)")
ax[2].set_xlabel("$x$")
ax[2].set_ylabel("$y$")
ax[2].set_title("(c)")
ax[0].set_ylim([10**-2, 1.5])
ax[1].set_ylim([10**-6, 1.5])
ax[2].set_ylim([-0.2, 2.2])
ax[2].set_xlim([-2, 2])

plt.tight_layout()

outfile = "fig/gradient-descent-b1.pdf"
if len(sys.argv) > 1:
    outfile = sys.argv[1]
plt.savefig(outfile)
df = pd.DataFrame(results)
print(df)
df.to_csv("fig/gradient-descent-b1.csv")
```

```python
import sympy as sp
import sys
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from lib import GradientDescent

LINEWIDTH = 0.7
x = sp.symbols('x')
y = x**4
dydx = y.diff()

fig, ax = plt.subplots(1, 2)

for step_size in np.array([0.5]):
    for start in np.array([1.00001]):
        print(step_size, start)
        g = GradientDescent()
        g.max_iter(100)
        g.step_size(step_size)
        g.start(start)
        g.function(lambda x1: float(y.subs(x, x1)))
        y_diff = y.diff()
        g.gradient(lambda x1: float(y_diff.subs(x, x1)))
        g.debug(True)

        def converged(x1, x2):
            abs = np.abs(x1-x2)
            print(abs, x1, x2)
            return abs < 0.001
        g.converged(converged)
        iterations, estimates, y_of_x = zip(*[(x[0], x[1], x[2]) for x in g.iterate()])
        print('y_of_x', y_of_x)
        print('iterations', iterations)
        print('estimates', estimates)
        sns.lineplot(
            x=iterations,
            y=estimates,
            ax=ax[0],
            linewidth=LINEWIDTH,
            legend=False,
            label=f"$\\alpha={step_size}$, $x={start}$")
        sns.lineplot(
            x=iterations,
            y=y_of_x,
            ax=ax[1],
            linewidth=LINEWIDTH,
            label=f"$\\alpha={step_size}$, $x={start}$")

ax[0].set_ylabel("estimate of $\\mathrm{arg\\,min}_x x^4$")
ax[0].set_xlabel("iteration")
ax[1].set_ylabel("$y(\\hat{x})$")
ax[1].set_xlabel("iteration")
ax[0].set_ylim([-10000, 10000])
ax[1].set_ylim([-100, 10000])
plt.tight_layout()

outfile = "fig/gradient-descent-x^4-crazy.pdf"
if len(sys.argv) > 1:
    outfile = sys.argv[1]
print(outfile)
plt.savefig(outfile)
```

```python
import sympy as sp
import sys
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from lib import GradientDescent

LINEWIDTH = 0.5
x = sp.symbols('x')
y = x**4
dydx = y.diff()

fig, ax = plt.subplots(1, 3, figsize=(12, 4))

blowup = 0.8

# alpha * (blowup ** 3) * 4 = 1.2

results = {
    "alpha": [],
    "start": [],
    "convergence time": [],
    "final guess": [],
}
iota = 0.00000000001
for (step_size, start, color) in [
            (0.1, 1, 'gray'),
        ]:
    print(step_size, start, color)
    g = GradientDescent()
    g.max_iter(100)
    g.step_size(step_size)
    g.start(start)
    g.function(lambda x1: float(y.subs(x, x1)))
    y_diff = y.diff()
    g.gradient(lambda x1: float(y_diff.subs(x, x1)))
    g.debug(True)

    def is_inf(x):
        import math
        if x == math.inf or x == -math.inf:
            return True

    def converged(x1, x2):
        if is_inf(x1) or is_inf(x2):
            return True
        abs = np.abs(x1-x2)
        print(abs, x1, x2)
        return abs < 0.001
    g.converged(converged)
    iterations, estimates, y_of_x = zip(*[
        (x[0], x[1], x[2]) for x in g.iterate()])
    results["alpha"].append(step_size)
    results["start"].append(start)
    results["convergence time"].append(len(iterations))
    results["final guess"].append(estimates[-1])
    print('y_of_x', y_of_x)
    print('iterations', iterations)
    print('estimates', estimates)
    sns.lineplot(
        x=iterations,
        y=np.abs(np.array(estimates)),
        ax=ax[0],
        linewidth=LINEWIDTH,
        legend=False,
        color=color,
        label=f"$\\alpha={step_size}$, $x={start}$")
    sns.lineplot(
        x=iterations,
        y=y_of_x,
        ax=ax[1],
        linewidth=LINEWIDTH,
        color=color,
        label=f"$\\alpha={step_size}$, $x={start}$")
    ax[2].step(
        estimates,
        y_of_x,
        linewidth=LINEWIDTH,
        color=color,
        label=f"$\\alpha={step_size}$, $x={start}$")
    xs = np.arange(-2, 2, 0.01)
    ys = [y.subs(x, xi) for xi in xs]
    ax[2].plot(
        xs,
        ys,
        linewidth=LINEWIDTH,
        label="$x^4$",
        color='yellow',
    )
    ax[2].scatter(
        start,
        g._function(start),
        color=color)


ax[0].set_ylabel("$|\\hat x|$")
ax[0].set_xlabel("iteration")
ax[0].set_yscale('log')
ax[0].set_title("(a)")
```

```
ax[1].set_yscale('log')
ax[1].set_ylabel("$y(\\hat{x})$")
ax[1].set_xlabel("iteration")
ax[1].set_title("(b)")
ax[2].set_xlabel("$x$")
ax[2].set_ylabel("$y$")
ax[2].set_title("(c)")
# ax[0].set_ylim([-7, 7])
# ax[1].set_ylim([-1, 4])
ax[2].set_ylim([-0.2, 1.2])
# ax[2].set_xlim([-2, 2])
plt.tight_layout()

outfile = "fig/gradient-descent-bi.pdf"
if len(sys.argv) > 1:
    outfile = sys.argv[1]
plt.savefig(outfile)
df = pd.DataFrame(results)
print(df)
df.to_csv("fig/gradient-descent-bi.csv")
```

```python
import sympy as sp
import sys
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from lib import GradientDescent

LINEWIDTH = 0.1
x = sp.symbols('x')
y = x**4
dydx = y.diff()

fig, ax = plt.subplots(1, 3, figsize=(12,4))

blowup = 0.8

# alpha * (blowup ** 3) * 4 = 1.2

results = {
    "alpha": [],
    "start": [],
    "gamma": [],
    "$f(x)$": [],
    "convergence time": [],
    "final guess": [],
}
iota = 0.005
def run(gamma, color, max_iter=99, plot=True):
    g = GradientDescent()
    g.max_iter(max_iter)
    alpha = 1
    start = 1
    g.step_size(alpha)
    g.start(start)
    y = gamma * (x**2)
    g.function(lambda x1: float(y.subs(x, x1)))
    y_diff = y.diff()
    g.gradient(lambda x1: float(y_diff.subs(x, x1)))
    g.debug(True)

    def is_inf(x):
        import math
        if x == math.inf or x == -math.inf:
            return True

    def converged(x1, x2):
        if is_inf(x1) or is_inf(x2):
            return True
        abs = np.abs(x1-x2)
        print(abs, x1, x2)
        return abs < 0.001
    g.converged(converged)
    iterations, estimates, y_of_x = zip(*[
        (x[0], x[1], x[2]) for x in g.iterate()])
    results["alpha"].append(alpha)
    results["gamma"].append(gamma)
    results["$f(x)$"].append(str(y))
    results["start"].append(start)
    results["convergence time"].append(len(iterations))
    results["final guess"].append(estimates[-1])
    if plot:
        sns.lineplot(
            x=iterations,
            y=estimates,
            ax=ax[0],
            linewidth=LINEWIDTH,
            legend=False,
            color=color,
            label=f"$\\gamma={gamma}$")
        sns.lineplot(
            x=iterations,
            y=y_of_x,
            ax=ax[1],
            linewidth=LINEWIDTH,
            color=color,
            label=f"$\\gamma={gamma}$")
        ax[2].step(
            estimates,
            y_of_x,
            linewidth=LINEWIDTH,
            color=color,
            label=f"$\\gamma={gamma}$")
        xs = np.arange(-2, 2, 0.01)
        ys = [y.subs(x, xi) for xi in xs]
        ax[2].plot(
            xs,
            ys,
            linewidth=LINEWIDTH,
            label="$\\gamma x^2$",
            color='yellow',
        )
        ax[2].scatter(
            start,
            g._function(start),
            color=color)


for (gamma, color) in [
        ( 0.01, 'green'),
```

```
            ( 0.1, 'blue'),
            ( 1 - iota, 'black'),
            (1 + iota, 'orange'),
            ( 1, 'red'),
            (-0.05, 'purple'),
        ]:
    run(gamma, color)

run(-1000, 'pink', max_iter=10000, plot=False)

ax[0].set_ylabel("$x$")
ax[0].set_xlabel("iteration")
ax[0].set_title("(a)")
ax[1].set_ylabel("$y(\\hat{x})$")
ax[1].set_xlabel("iteration")
ax[1].set_title("(b)")
ax[2].set_xlabel("$x$")
ax[2].set_ylabel("$y$")
ax[2].set_title("(c)")
ax[0].set_ylim([-7, 7])
ax[1].set_ylim([-1, 4])
ax[2].set_ylim([-1, 2.2])
ax[2].set_xlim([-2, 2])
plt.tight_layout()

outfile = "fig/gradient-descent-ci.pdf"
if len(sys.argv) > 1:
    outfile = sys.argv[1]
plt.savefig(outfile)
df = pd.DataFrame(results)
print(df)
df.to_csv("fig/gradient-descent-ci.csv")
```

```python
#!/usr/bin/env python

import pandas as pd
import sys
import subprocess
import os


def csv_to_latex_pdf(input_csv, output_pdf="output.pdf"):
    # Read the CSV file into a pandas DataFrame
    df = pd.read_csv(input_csv, dtype=str)

    # Convert the DataFrame to LaTeX tabular format
    df_to_latex_pdf(df, output_pdf=output_pdf)


def format_float(x):
    if isinstance(x, float):
        import math
        if x == math.inf:
            return "$\\infty$"
        if x == -math.inf:
            return "$-\\infty"
        if x == math.nan:
            return "NaN"
        return ("\\num{{{0:.2g}}}".format(x))


def df_to_latex_pdf(df, output_pdf="output.pdf"):
    # Create the tmp directory if it doesn't exist
    if not os.path.exists("tmp"):
        os.makedirs("tmp")
    latex_tabular = df.to_latex(float_format=format_float)

    # Wrap the tabular code in a LaTeX document
    latex_document = r"""\documentclass{article}
\usepackage{booktabs}
\usepackage{siunitx}
\begin{document}
\thispagestyle{empty}
    """ + latex_tabular + r"""\end{document}"""

    output_tex = "tmp/output.tex"

    # Save the LaTeX code to a file
    with open(output_tex, 'w') as f:
        f.write(latex_document)

    # Compile the LaTeX file using pdflatex
    subprocess.run(["pdflatex", "-jobname=tmp/output", output_tex])
    subprocess.run(["pdfcrop", "tmp/output.pdf", output_pdf])

    print(f"PDF generated as {output_pdf}")


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: python script_name.py input.csv output.pdf")
        sys.exit(1)

    input_csv = sys.argv[1]
    output_pdf = sys.argv[2]
    csv_to_latex_pdf(input_csv, output_pdf)
```

```
import lib
import sys
import argparse
import numpy as np
import rms2
import adam
import hb


def converged(x1, x2):
    d = np.max(x1-x2)
    return d < 0.001


parser = argparse.ArgumentParser(
    prog="Run Gradient Descent A Step Size Algorithm")

parser.add_argument('-al', '--algorithm', choices=[
    'rmsprop', 'adam', 'polyak', 'heavy_ball'], required=True)

parser.add_argument('-b', '--beta', type=float)
parser.add_argument('-b2', '--beta2', type=float)
parser.add_argument('-a', '--alpha', type=float)
parser.add_argument('filename')

args = parser.parse_args()

print(args.filename)

gd = lib.GradientDescent()
if 0:
    pass
elif args.algorithm == 'rmsprop':
    gd.set_iterate(rms2.iterate)
elif args.algorithm == 'adam':
    gd.set_iterate(adam.iterate)
elif args.algorithm == 'heavy_ball':
    gd.set_iterate(hb.iterate)
else:
    print("no algorithm")
    sys.exit(1)

gd.start(np.array([4, 8]))
gd.converged(converged)
gd.step_size(args.alpha)
gd.beta(args.beta)
gd.beta2(args.beta2)
gd.epsilon(0.0001)
gd.max_iter(-1)


def fn(x):
    return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])


def grad(x):
    return np.array([
        lib.f.diff(var).subs(
            lib.x, x[0]
        ).subs(
            lib.y, x[1]
        ) for var in (lib.x, lib.y)])


gd.converged(converged)
gd.function(fn)
gd.gradient(grad)
gd.run2csv(args.filename)
```

```python
class GradientDescent():
# ...
    def iterate(self):
        import math
        x_value = self._start
        old_x_value = None
        iteration = 0
        while True:
            yield [iteration, float(x_value), float(self._function(x_value))]
            iteration += 1
            if self._max_iter > 0 and iteration > self._max_iter:
                break
            grad_value = self._gradient(x_value)
            x_value -= self._step_size * grad_value  # Update step
            if old_x_value is not None and self._converged(x_value, old_x_value):
                yield [iteration, float(x_value), float(self._function(old_x_value))]
                print("converged")
                break
            old_x_value = x_value
```

```python
import lib


def iterate(self):
    x_value = self._start
    old_x_value = None
    iteration = 0
    converged = False
    grad_value = self._gradient(x_value)

    z = 0

    def yielded():
        print(x_value)
        print(iteration)
        return {
            "iteration": iteration,
            "z": z,
            "x": x_value,
            "f(x)": self._function(x_value),
            "sum": sum,
            "epsilon": self._epsilon,
            "converged": converged,
            "gradient": grad_value,
        }

    yield yielded() # yield initial values

    while not converged:
        iteration += 1
        if self._max_iter > 0 and iteration > self._max_iter:
            break
        grad_value = self._gradient(x_value)
        old_x_value = x_value
        z = self._beta * z + self._step_size * grad_value
        x_value = x_value - z
        converged = self._converged(x_value, old_x_value)
        yield yielded()


if __name__ == "__main__":
    import numpy as np
    hb = lib.GradientDescent()
    hb.step_size(10**-3)
    hb.beta(0.5)
    hb.max_iter(-1)
    hb.start(np.array([0, 0]))

    def converged(x1, x2):
        d = np.max(x1-x2)
        return d < 0.000001

    def fn(x):
        return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])

    def grad(x):
        return np.array([
            lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
            for var in (lib.x, lib.y)])
    hb.converged(converged)
    hb.function(fn)
    hb.gradient(grad)
    hb.set_iterate(iterate)
    hb.run2csv("hb.csv")
```

```python
import lib


def iterate(self):
    x_value = self._start
    old_x_value = None
    iteration = 0
    converged = False
    grad_value = self._gradient(x_value)

    z = 0

    def yielded():
        print(x_value)
        print(iteration)
        return {
            "iteration": iteration,
            "z": z,
            "x": x_value,
            "f(x)": self._function(x_value),
            "sum": sum,
            "epsilon": self._epsilon,
            "converged": converged,
            "gradient": grad_value,
        }

    yield yielded() # yield initial values

    while not converged:
        iteration += 1
        if self._max_iter > 0 and iteration > self._max_iter:
            break
        grad_value = self._gradient(x_value)
        old_x_value = x_value
        z = self._beta * z + self._step_size * grad_value
        x_value = x_value - z
        converged = self._converged(x_value, old_x_value)
        yield yielded()


if __name__ == "__main__":
    import numpy as np
    hb = lib.GradientDescent()
    hb.step_size(10**-3)
    hb.beta(0.5)
    hb.max_iter(-1)
    hb.start(np.array([0, 0]))

    def converged(x1, x2):
        d = np.max(x1-x2)
        return d < 0.000001

    def fn(x):
        return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])

    def grad(x):
        return np.array([
            lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1])
            for var in (lib.x, lib.y)])
    hb.converged(converged)
    hb.function(fn)
    hb.gradient(grad)
    hb.set_iterate(iterate)
    hb.run2csv("hb.csv")
```

```python
import sympy as sp
import functools

x, y = sp.symbols('x y', real=True)
f = 3 * (x - 5)**4 + (10 * ((y - 9)**2))
g = sp.Max(x - 5, 0) + (10 * sp.Abs(y - 9))


class GradientDescent():
    def __init__(self):
        self._max_iter = 1000
        self._debug = False
        self._converged = lambda x1, x2: False
        self._epsilon = 0.0001
        self._beta = 0

    def step_size(self, a):
        self._step_size = a
        return self

    def beta(self, b):
        self._beta = b
        return self

    def beta2(self, b):
        self._beta2 = b
        return self

    def epsilon(self, e):
        self._epsilon = e
        return self

    def function(self, f):
        self._function = f
        return self

    def gradient(self, g):
        self._gradient = g
        return self

    def max_iter(self, m):
        self._max_iter = m
        return self

    def start(self, s):
        self._start = s
        return self

    def debug(self, d):
        self._debug = d
        return self

    def converged(self, c):
        self._converged = c
        return self

    def set_iterate(self, f):
        self.iterate = functools.partial(f, self)
        return self

    def iterate(self):
        x_value = self._start
        old_x_value = None
        iteration = 0
        while True:
            yield [iteration, x_value, self._function(x_value)]
            iteration += 1
            if self._max_iter > 0 and iteration > self._max_iter:
                break
            grad_value = self._gradient(x_value)
            print(x_value, type(x_value))
            print(grad_value, type(grad_value))
            x_value -= self._step_size * grad_value  # Update step
            if old_x_value is not None and self._converged(x_value, old_x_value):
                yield [iteration, float(x_value), float(self._function(old_x_value))]
                print("converged")
                break
            old_x_value = x_value

    def run2csv(self, fname, summarise=True):
        import pandas as pd
        iterations = list(self.iterate())
        df = pd.DataFrame(iterations)
        df.to_csv(fname)
        if(summarise):
            with open(fname + ".summary", "w") as f:
                print(f"iterations: {len(df)}", file=f)
                print(f"start: {df['x'][0]}", file=f)
                print(f"final: {df['x'][len(df) - 1]}", file=f)


if __name__ == "__main__":
    print(f.diff(x), f.diff(y))
    print(g.diff(x), g.diff(y))
```

```python
import numpy as np


def polyak_step_size(self, sp_func, sp_x, x, f_star):
    assert len(sp_x) == len(x)
    subs = {sp_xi: xi for sp_xi, xi in zip(sp_x, x)}
    fx = sp_func.subs(subs)
    grad = [sp_func.diff(sp_xi).subs(subs) for sp_xi in sp_x]
    grad = np.array(grad)
    denominator = sum(grad * grad)
    numerator = fx - f_star
    return numerator / denominator
```

```python
import lib


def iterate(self):
    import numpy as np
    x_value = self._start
    old_x_value = None
    iteration = 0
    sum = np.zeros(x_value.shape)
    alpha_n = np.zeros(x_value.shape)
    alpha_n.fill(self._step_size)
    converged = False
    grad_value = self._gradient(x_value)

    def yielded():
        print(x_value)
        print(iteration)
        return {
            "iteration": iteration,
            "x": x_value,
            "f(x)": self._function(x_value),
            "sum": sum,
            "epsilon": self._epsilon,
            "converged": converged,
            "gradient": grad_value,
            "alpha_n": alpha_n,
        }

    yield yielded()

    while not converged:
        iteration += 1
        if self._max_iter > 0 and iteration > self._max_iter:
            break
        grad_value = self._gradient(x_value)
        old_x_value = x_value
        print(grad_value, type(grad_value))
        print(alpha_n, type(alpha_n))
        print(x_value, type(x_value))
        x_value = x_value - alpha_n * grad_value
        sum = self._beta * sum + (1-self._beta) * (grad_value**2)
        alpha_n = self._step_size / (sum**0.5+self._epsilon)
        converged = self._converged(x_value, old_x_value)
        yield yielded()


def rms_gradient_descent():
    rms = lib.GradientDescent()
    rms.set_iterate(iterate)
    return rms


if __name__ == "__main__":
    import numpy as np
    rms = lib.GradientDescent()
    rms.epsilon(0.0001)
    rms.step_size(10**-2)
    rms.beta(0.1)
    rms.max_iter(-1)
    rms.start(np.array([0, 0]))

    def converged(x1, x2):
        d = np.max(x1-x2)
        return d < 0.000001

    def fn(x):
        return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])

    def grad(x):
        return np.array([lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1]) for var in (lib.x, lib.y)])
    rms.converged(converged)
    rms.function(fn)
    rms.gradient(grad)
    rms.set_iterate(iterate)
    rms.run2csv("rms2.csv")
```

```python
class RMSPropGradientDescent():
    def __init__(self):
        self._max_iter = 1000
        self._debug = False
        self._converged = lambda x1, x2: False
        self._sum = 0
        self._epsilon = 0.0001

    def step_size(self, a):
        self._step_size = a
        return self

    def epsilon(self, e):
        self._epsilon = e
        return self

    def beta(self, b):
        self._beta = b
        return self

    def function(self, f):
        self._function = f
        return self

    def gradient(self, g):
        self._gradient = g
        return self

    def max_iter(self, m):
        self._max_iter = m
        return self

    def start(self, s):
        self._start = s
        return self

    def debug(self, d):
        self._debug = d
        return self

    def converged(self, c):
        self._converged = c
        return self

    def run2csv(self, fname):
        import pandas as pd
        iterations = list(self.iterate())
        df = pd.DataFrame(iterations)
        df.to_csv(fname)

    def iterate(self):
        import math
        import numpy as np
        x_value = self._start
        old_x_value = None
        iteration = 0
        sum = np.zeros(x_value.shape)
        alpha_n = np.zeros(x_value.shape)
        alpha_n.fill(self._step_size)
        converged = False
        grad_value = self._gradient(x_value)

        def yielded():
            print(x_value)
            print(iteration)
            return {
                "iteration": iteration,
                "x": x_value,
                "f(x)": self._function(x_value),
                "sum": sum,
                "epsilon": self._epsilon,
                "converged": converged,
                "gradient": grad_value,
                "alpha_n": alpha_n,
            }

        yield yielded()

        while not converged:
            iteration += 1
            if self._max_iter > 0 and iteration > self._max_iter:
                break
            grad_value = self._gradient(x_value)
            old_x_value = x_value
            print(grad_value, type(grad_value))
            print(alpha_n, type(alpha_n))
            print(x_value, type(x_value))
            x_value = x_value - alpha_n * grad_value
            sum = self._beta * sum + (1-self._beta) * (grad_value**2)
            alpha_n = self._step_size / (sum**0.5+self._epsilon)
            converged = self._converged(x_value, old_x_value)
            yield yielded()


if __name__ == "__main__":
    import numpy as np
    import lib
    rms = RMSPropGradientDescent()
    rms.epsilon(0.0001)
    rms.step_size(10**-2)
```

```
    rms.beta(0.1)
    rms.max_iter(-1)
    rms.start(np.array([0, 0]))

    def converged(x1, x2):
        d = np.max(x1-x2)
        return d < 0.000001

    def fn(x):
        return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])

    def grad(x):
        return np.array([lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1]) for var in (lib.x, lib.y)])
    rms.converged(converged)
    rms.function(fn)
    rms.gradient(grad)
    rms.run2csv("rms.csv")
```

```python
import lib


def iterate(self):
    import numpy as np
    x_value = self._start
    old_x_value = None
    iteration = 0
    sum = np.zeros(x_value.shape)
    alpha_n = np.zeros(x_value.shape)
    alpha_n.fill(self._step_size)
    converged = False
    grad_value = self._gradient(x_value)

    def yielded():
        print(x_value)
        print(iteration)
        return {
            "iteration": iteration,
            "x": x_value,
            "f(x)": self._function(x_value),
            "sum": sum,
            "epsilon": self._epsilon,
            "converged": converged,
            "gradient": grad_value,
            "alpha_n": alpha_n,
        }

    yield yielded()

    while not converged:
        iteration += 1
        if self._max_iter > 0 and iteration > self._max_iter:
            break
        grad_value = self._gradient(x_value)
        old_x_value = x_value
        print(grad_value, type(grad_value))
        print(alpha_n, type(alpha_n))
        print(x_value, type(x_value))
        x_value = x_value - alpha_n * grad_value
        sum = self._beta * sum + (1-self._beta) * (grad_value**2)
        alpha_n = self._step_size / (sum**0.5+self._epsilon)
        converged = self._converged(x_value, old_x_value)
        yield yielded()


def rms_gradient_descent():
    rms = lib.GradientDescent()
    rms.set_iterate(iterate)
    return rms


if __name__ == "__main__":
    import numpy as np
    rms = lib.GradientDescent()
    rms.epsilon(0.0001)
    rms.step_size(10**-2)
    rms.beta(0.1)
    rms.max_iter(-1)
    rms.start(np.array([0, 0]))

    def converged(x1, x2):
        d = np.max(x1-x2)
        return d < 0.000001

    def fn(x):
        return lib.f.subs(lib.x, x[0]).subs(lib.y, x[1])

    def grad(x):
        return np.array([lib.f.diff(var).subs(lib.x, x[0]).subs(lib.y, x[1]) for var in (lib.x, lib.y)])
    rms.converged(converged)
    rms.function(fn)
    rms.gradient(grad)
    rms.set_iterate(iterate)
    rms.run2csv("rms2.csv")
```

```python
import sympy as sp

x = sp.symbols('x')
print(x)
f = x ** 4
print(f)
print(f.diff())
print(f.subs(x, x**2))
print(f.conjugate())
print(f)
print(f.subs())
```

```python
import matplotlib.pyplot as plt
import numpy as np
import sys


def f(x, y):
    return 3 * (x - 5)**4 + 10 * (y - 9)**2


def g(x, y):
    return np.maximum(x - 5, 0) + 10 * np.abs(y - 9)


def main(outfile):
    x = np.linspace(0, 10, 400)
    y = np.linspace(0, 18, 400)
    X, Y = np.meshgrid(x, y)
    Z_f = f(X, Y)
    Z_g = g(X, Y)

    fig = plt.figure(figsize=(12, 6))

    ax = fig.add_subplot(1, 2, 1, projection='3d')
    ax.plot_surface(X, Y, Z_f, cmap='viridis')
    ax.set_title('$f(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_zlabel('$f(x, y)$')

    ax = fig.add_subplot(1, 2, 2, projection='3d')
    ax.plot_surface(X, Y, Z_g, cmap='magma')
    ax.set_title('$g(x, y)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_zlabel('$g(x, y)$')

    plt.savefig(outfile)
    plt.show()


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python script.py <output_file>")
        sys.exit(1)

    outfile = sys.argv[1]
    main(outfile)

```