COMPSCI 527 Homework 2

Group Members:

Problem 0 (3 points)

Part 1: Optimization Basics

Problem 1.1 (Exam Style)

$$f(x,y) = (x^3 - 3x + 3)(2y^2 + 1)$$

$$g(x,y) = \nabla f = ((2y^2+1)(3x^2-3), (x^3-3x+3)(4y))$$

$$H(x,y) = \begin{pmatrix} (2y^2+1)(6x) & (12y)(x^2-1) \\ (12y)(x^2-1) & 4(x^3-3x+3) \end{pmatrix}$$

Problem 1.2 (Exam Style)

$$\frac{\partial f}{\partial x} = 0$$

$$\therefore (x^3 - 3x + 3)(2y^2 + 1) = 0$$

$$x = \pm 1$$

$$\frac{\partial f}{\partial y} = 0$$

$$\therefore (x^3 - 3x + 3)(4y) = 0$$

$$y = 0$$

or

$$(x^3 - 3x + 3) = 0$$

$\therefore (1,0)$ and $(-1,0)$ are the only two stationary points.

$$f(1,0) = 1$$

$$f(-1,0) = 5$$

$$H(1,0) = \begin{pmatrix} 6 & 0 \\ 0 & 4 \end{pmatrix}$$

det $H = 24 > 0$

$\therefore (1,0)$ is a minimum point

$$H(-1,0) = \begin{pmatrix} -6 & 0 \\ 0 & 20 \end{pmatrix}$$

det $H = -120 < 0$

$\therefore (-1,0)$ is a saddle point

Problem 1.3 (Exam Style)

One step of ordinary gradient descent:

$$\begin{gathered} -\nabla f(2,0) = -(9,0) \\ z_1 = z_0 + \alpha_0(-\nabla f(z_0)) \\ = (2,0) + 0.01(-9,0) \\ = (1.91,0) \end{gathered}$$

Problem 1.4 (Exam Style)

Gradient descent with momentum:

$$\begin{gathered} v_1 = (0.8)(0) - (0.01)(\nabla f(2,0)) \\ = (-0.09,0) \end{gathered}$$

$$\begin{gathered} z_1 = z_0 + v_1 \\ = (1.91,0) \end{gathered}$$

$$\begin{gathered} v_2 = 0.8(-0.09,0) - (0.01)(\nabla f(1.91,0)) \\ = (-0.072,0) - 0.01(3(1.91^2) - 3,0) \\ = (-0.151443,0) \end{gathered}$$

$$\begin{gathered} z_2 = (1.91,0) + (-0.151443,0) \\ = (1.758557,0) \end{gathered}$$

Problem 1.5 (Exam Style)

Newton's method:

$$
\begin{aligned}
z_0 &= (2, 0) \\
z_1 &= z_0 + \triangle z_0 \\
&= z_0 - H_0^{-1} g_0 \\
&= z_0 - H_0^{-1}(z_0) \nabla f(z_0) \\
\nabla f(z_0) &= (9, 0) \\
H_0^{-1}(z_0) &= \frac{1}{12 \times 20 - 0} \begin{pmatrix} 20 & 0 \\ 0 & 12 \end{pmatrix} \\
&= \begin{pmatrix} \frac{1}{12} & 0 \\ 0 & \frac{1}{20} \end{pmatrix} \\
z_1 &= (2, 0) - \begin{pmatrix} \frac{1}{12} & 0 \\ 0 & \frac{1}{20} \end{pmatrix} \begin{pmatrix} 9 \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} \frac{5}{4} \\ 0 \end{pmatrix}
\end{aligned}
$$

Problem 1.6 (Exam Style)

1. From ordinary gradient descent, distance from local minimum $(1, 0)$ is 0.91. From Newton's method, distance from local minimum $(1, 0)$ is 0.25 $\therefore$ Newton's method gets closer to the minimum.
2. No. $f(x, y)$ is not a sum of differentiable squares of the form $f(z) = \sum_{n=1}^{N} \phi_n^2(z)$

Problem 1.7 (Exam Style)

$\vec{x_0} = (-1, 1)$. Gradient descent on f:

$$
\begin{aligned}
\nabla f(-1, y) &= (0, 20y) \\
f(-1, y) &= 5(2y^2 + 1) \\
\therefore y_{k+1} &= y_k + \alpha_k(-20y_k) \\
&= (1 - 20\alpha_k) y_k
\end{aligned}
$$

For $0 < \alpha_k < 0.05$,

$$
|1 - 20\alpha_k| < 1
$$

$\therefore$ y converges geometrically to 0. The minimum value of y is 5 since iteration starts at $(-1, 1)$, the value of x never changes and the algorithm consequently converges to $(-1, 0)$.

Problem 1.8 (Exam Style)

$$f(x,y) \;=\; \left(x^3 \;-\; 3x \;+\; 3\right)\left(2y^2 \;+\; 1\right).$$

Its gradient is:

$$\nabla f(x,y) \;=\; \begin{pmatrix} \frac{\partial f}{\partial x}(x,y) \\ \frac{\partial f}{\partial y}(x,y) \end{pmatrix} = \begin{pmatrix} \left(3x^2 - 3\right)\left(2y^2 + 1\right) \\ \left(x^3 - 3x + 3\right)\left(4y\right) \end{pmatrix}.$$

For any y, $\frac{\partial f}{\partial x}(-1,y) \;=\; \left(3(-1)^2 - 3\right)\left(2y^2 + 1\right) \;=\; \left(3 - 3\right)\left(2y^2 + 1\right) \;=\; 0$ If gradient descent is started at $\left(x_0, y_0\right) = (-1, 1)$, the x-coordinate will not deviate away from -1 under exact arithmetic because its partial derivative in x is identically zero.

For $x = -1$, the partial derivative in y is
$\frac{\partial f}{\partial y}(-1,y) \;=\; \left((-1)^3 - 3(-1) + 3\right)\left(4y\right) \;=\; \left(-1 + 3 + 3\right)\left(4y\right) \;=\; 5\left(4y\right) \;=\; 20\,y.$
Thus, gradient descent updates will drive y to 0. Eventually, $(x,y) \to (-1,0)$. A Hessian check shows that $(-1,0)$ is a saddle point.

In floating-point arithmetic, $3(-1)^2 - 3 \;=\; 3 - 3 \;=\; 0$ cannot be represented as exactly zero. Numerical roundoff errors will produce a small nonzero value. As such, the partial derivative in x is not exactly zero, and small updates in x accumulate over many iterations, pushing x away from -1.

Once x deviates from -1, the gradient no longer vanishes, and the algorithm will begin descending away from the saddle at $(-1,0)$, since $(-1,0)$ is a saddle point. Since $\left(x^3 - 3x + 3\right)$ has a global minimum at $x = 1$ in that region, gradient descent will push x toward 1, eventually converging to $\left(1,0\right)$.

Therefore, the convergence behavior changes under finite-precision arithmetic. In exact arithmetic, the algorithm stays on $x = -1$ and converges to the saddle point $(-1,0)$. However, in finite precision, small roundoff errors are produced that make the update in x non-zero, eventually pushing x away from -1. The iteration then converges to the lower-valued minimum at $\left(1,0\right)$.

Part 2: Optimization Code

In [1]:
```
def f(z):
    x, y = z[0], z[1]
    return (x ** 3 - 3 * x + 3) * (2 * y ** 2 + 1)
```

In [2]:
```
class State:
    def __init__(self, z, fct):
        self.z = z
        output = fct(z)
        assert len(output) >= 2, 'at least two outputs are needed from fct'
        self.fz = output[0]
```

```
        self.gradient = output[1]
        self.Hessian = None if len(output) < 3 else output[2]
```

In [3]:
```
import numpy as np
from copy import deepcopy
```

In [4]:
```
class Descender:
    def __init__(
            self, fct, z0, step_fct, delta=1.e-4, max_iterations=1000, record=True,
    ):
        self.function = fct
        self.step = step_fct
        self.delta = delta
        self.max_iterations = max_iterations
        self.state = State(z0, fct)
        self.old_state = None
        self.history = [deepcopy(self.state)] if record else None
        self.quiet = quiet

    def successful_termination(self):
        if np.linalg.norm(self.state.gradient) == 0.:
            return True
        if self.old_state is not None:
            return np.linalg.norm(self.state.z - self.old_state.z) < self.delta

    def descend(self):
        found = False
        for k in range(self.max_iterations):
            if self.successful_termination():
                found = True
                break
            self.old_state = deepcopy(self.state)
            z = self.step(self.state)
            self.state = State(z, self.function)
            if self.history is not None:
                self.history.append(self.state)
        if not self.quiet:
            if not found:
                print('Warning: maximum number of iterations exceeded')
        return (self.state.z, self.state.fz) if self.history is None else self.hist
```

Problem 2.1: Function Wrappers

In [5]:
```
def fg(z):
    fz = f(z)
    x, y = z[0], z[1]
    gz = np.stack([
        3 * (x ** 2 - 1) * (2 * y ** 2 + 1),
        4 * y * (x ** 3 - 3 * x + 3)
    ], axis=0)
    return fz, gz
```

In [6]:
```
def fgh(z):
    fz, gz = fg(z)
```

```
    x, y = z[0], z[1]

    hxx = 6 * x * (2 * y ** 2 + 1)
    hxy = 12 * y * (x ** 2 - 1)
    hyy = 4 * (x ** 3 - 3 * x + 3)

    hz = np.array([[hxx, hxy],
                   [hxy, hyy]])
    return fz, gz, hz

z_init = np.array([-0.5, 2])

fz, gz, hz = fgh(z_init)
print("z_init :", z_init)
print("fz :", fz)
print("gz :", gz)
print("hz :", hz)
```

```
z_init : [-0.5  2. ]
fz : 39.375
gz : [-20.25  35.  ]
hz : [[-27.  -18. ]
 [-18.   17.5]]
```

Problem 2.2: Stepper Functions

In [7]:
```
class GDStepper:
    def __init__(self, alpha):
        self.alpha = alpha

    def step(self, s):
        return s.z - self.alpha * s.gradient

class MomentumStepper:
    def __init__(self, alpha, momentum):
        self.alpha = alpha
        self.momentum = momentum
        self.vel = None

    def step(self, s):
        if self.vel is None:
            self.vel = np.zeros(shape = 2)

        self.vel = self.momentum * self.vel - self.alpha * s.gradient
        return s.z + self.vel
```

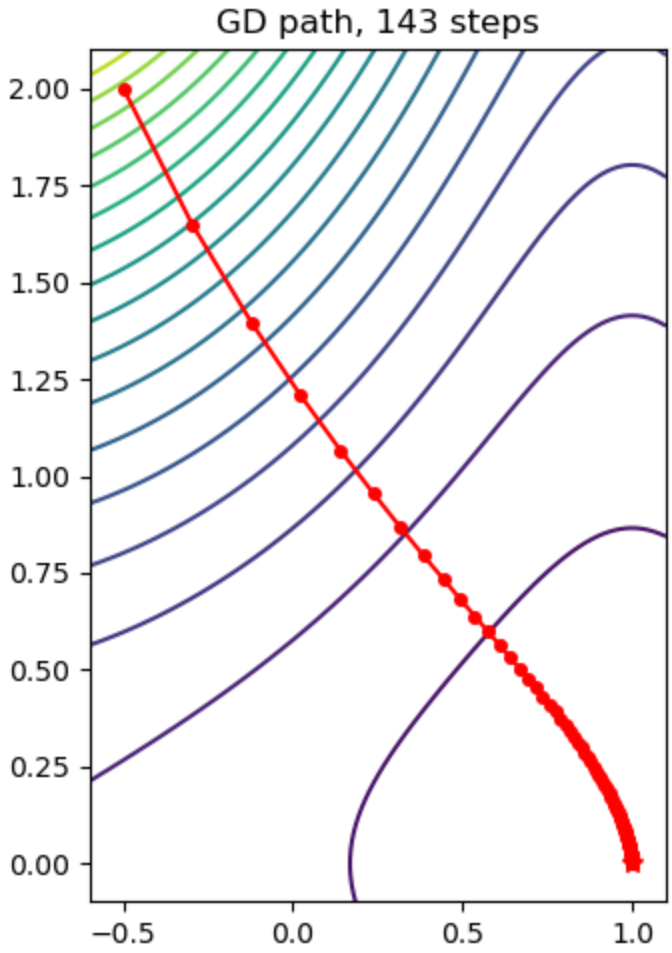
In [8]:
```
learning_rate = 0.01
d = Descender(fg, z_init, GDStepper(learning_rate).step)
history = d.descend()

momentum = 0.8
m = Descender(fg, z_init, MomentumStepper(learning_rate, momentum).step)
history_2  = m.descend()
```

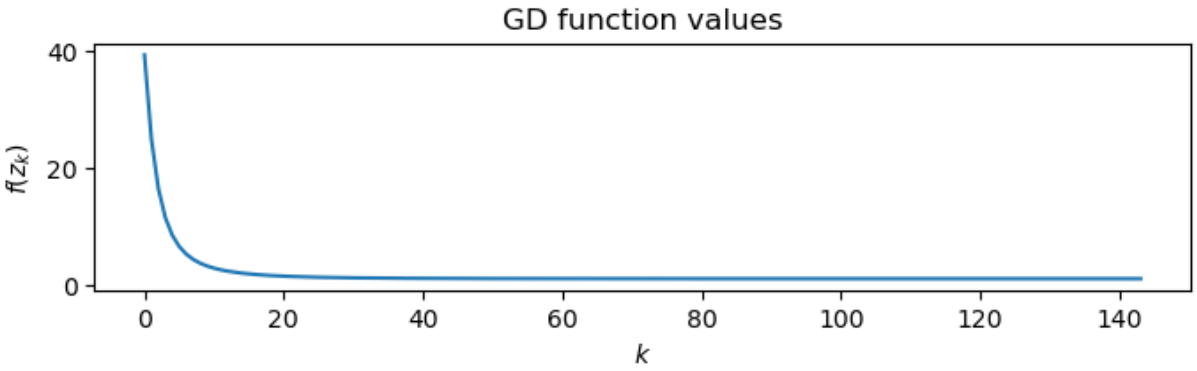
In [9]:
```
import urllib.request
import ssl
from os import path as osp
import shutil


def retrieve(file_name, semester='spring25', homework=2):
    if osp.exists(file_name):
        print('Using previously downloaded file {}'.format(file_name))
    else:
        context = ssl._create_unverified_context()
        fmt = 'https://www2.cs.duke.edu/courses/{}/compsci527/homework/{}/{}'
        url = fmt.format(semester, homework, file_name)
        with urllib.request.urlopen(url, context=context) as response:
            with open(file_name, 'wb') as file:
                shutil.copyfileobj(response, file)
        print('Downloaded file {}'.format(file_name))
```

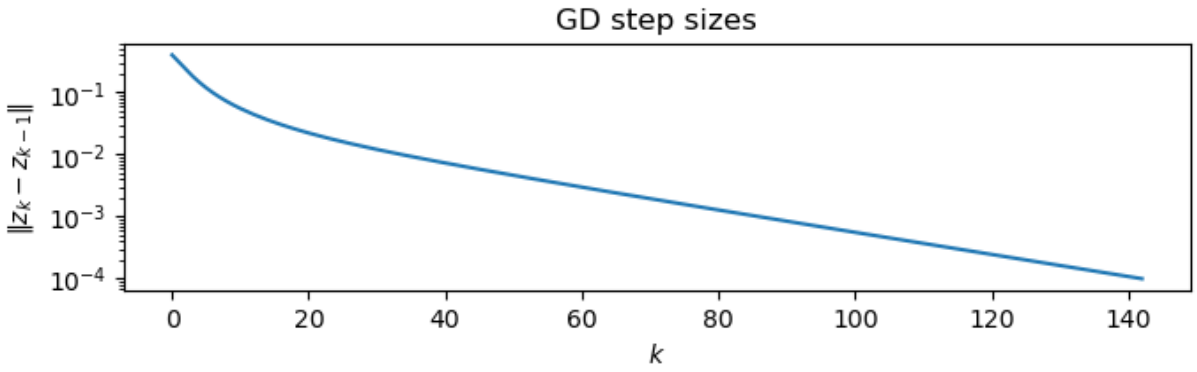
In [10]:
```
retrieve('gd_plot.py')
```

```
Using previously downloaded file gd_plot.py
```

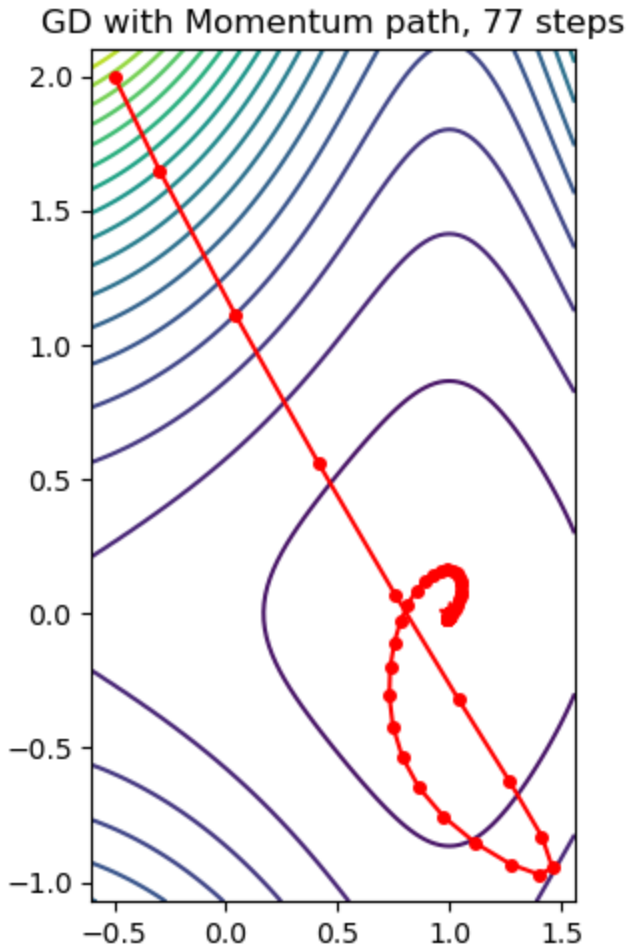
In [11]:
```
from gd_plot import plot_path, plot_progress
from matplotlib import pyplot as plt
%matplotlib inline

plot_path("GD", history, f)
plot_progress('GD', history)

plot_path("GD with Momentum", history_2, f)
plot_progress("GD with Momentum", history_2)
```

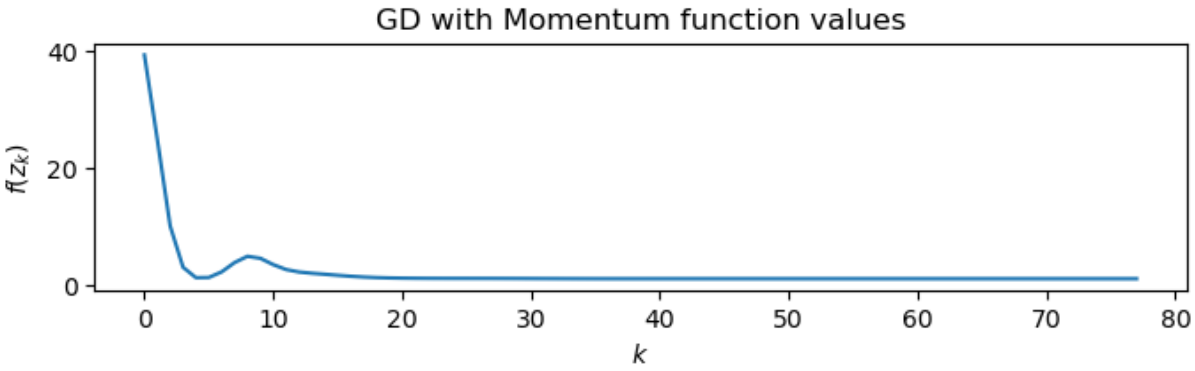


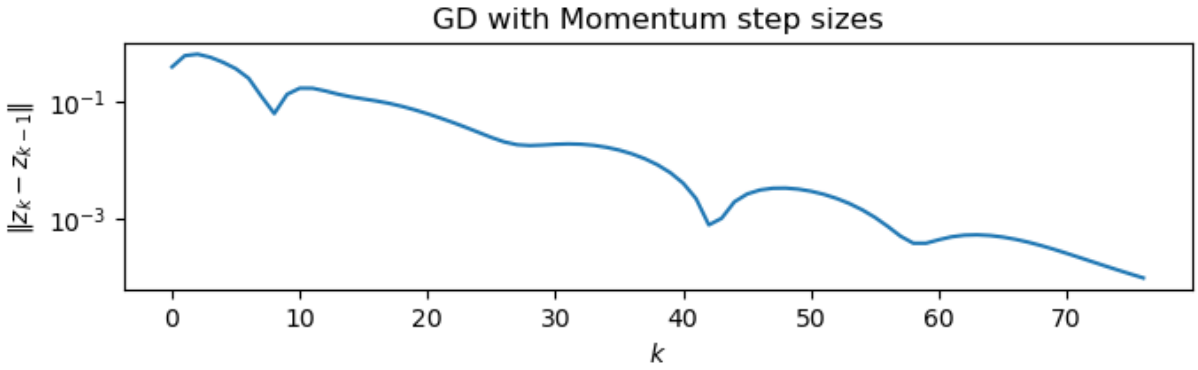





```
GD: from f(-0.50000, 2.00000) = 39.37500 to f(0.99991, 0.00237) = 1.00001 in 143 ite
rations
```








```
GD with Momentum: from f(-0.50000, 2.00000) = 39.37500 to f(0.99987, -0.00048) = 1.0
0000 in 77 iterations
```

1. With the inclusion of an update step $\mathbf{v}_{k+1} = \mu_k \mathbf{v}_k - \alpha_k \nabla f(\mathbf{z}_k)$ that is dependent on and derived from past steps / velocities being used to compute new estimates of the minimum, each update is not simply the learning rate multiplied by the gradient of the function at the previous estimate of the minimum i.e. $\mathbf{v}_{k+1} = \alpha_k \nabla f(\mathbf{z}_k)$. Therefore, "gradient descent" is not accurate since we are no longer descending along the instantaneous steepest direction at each estimate of the minimum. The update step is altered by the addition of a term that depends on past update steps.

2. Yes. In the case of momentum, the number of iterations needed for successful termination is 77, compared to 143 iterations for gradient descent using the same learning rate α.

3. Using momentum can cause overshooting of the minimum and oscillations if the momentum coefficient μ is too large and inadequately tuned. We can see this in the contour plot above, where the iterates circle around the minimum instead of steadily converging towards the minimum. This may cause the estimates to be unable to converge to a minimum, or even diverge.

In [12]:
```
import numpy as np

def run_momentum_experiments(fgh, z_init, alpha=0.01):
    mus = [0.0, 0.5, 0.8, 0.9]
    results = []

    for mu in mus:
        stepper = MomentumStepper(alpha=alpha, momentum=mu)

        d = Descender(
            fct=fgh,
            z0=z_init,
            step_fct=stepper.step,
            delta=1e-4,
            max_iterations=1000,
            record=True,
            quiet=True
        )

        history = d.descend()
        final_state = history[-1]
        z_final = final_state.z
        f_final = final_state.fz
        iteration_count = len(history) - 1
        results.append((mu, z_final, f_final, iteration_count))

    return results

z_init = np.array([-0.5, 2.0])
alpha = 0.01
out = run_momentum_experiments(fgh, z_init, alpha)
print(" momentum |      final z      |  f(z)    | iters")
```

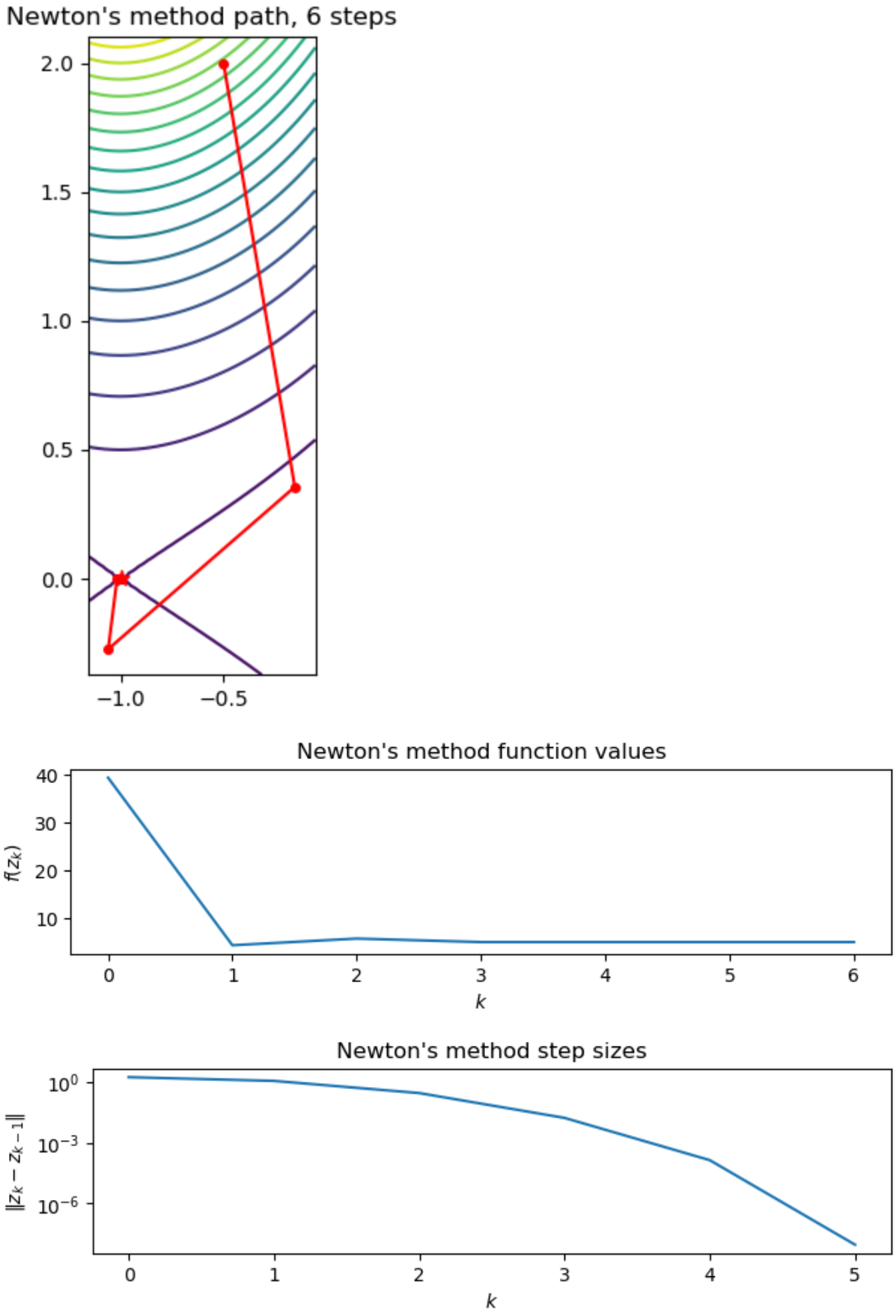
```
for row in out:
    mu, z_final, f_final, icount = row
    print(f"{mu:3.2f}: {z_final}, {f_final}, {icount} iterations")


print("From the above experiments, I conclude that a momentum value of 0.8 is suita
```

```
 momentum |      final z      |  f(z)    | iters
0.00: [0.99990509 0.0023656 ], 1.0000112191797441, 143 iterations
0.50: [9.99956237e-01 9.70877341e-04], 1.0000018909511257, 62 iterations
0.80: [ 9.99869125e-01 -4.79294004e-04], 1.0000005108279129, 77 iterations
0.90: [1.00065738 0.00133318], 1.0000048514712516, 146 iterations
From the above experiments, I conclude that a momentum value of 0.8 is suitable. It
produces a reasonable number of iterations, 77, and yields a value sufficiently clos
e to the minimum point (1, 0).
```

Problem 2.3

In [13]:

```
def newton(s):
    H = s.Hessian
    g = s.gradient
    return s.z - np.linalg.inv(H) @ g

z_init = np.array([-0.5, 2.0])
n = Descender(fgh, z_init, newton)
history_3 = n.descend()

plot_path("Newton's method", history_3, f)
plot_progress("Newton's method", history_3)
```




```
Newton's method: from f(-0.50000, 2.00000) = 39.37500 to f(-1.00000, -0.00000) = 5.0
0000 in 6 iterations
```

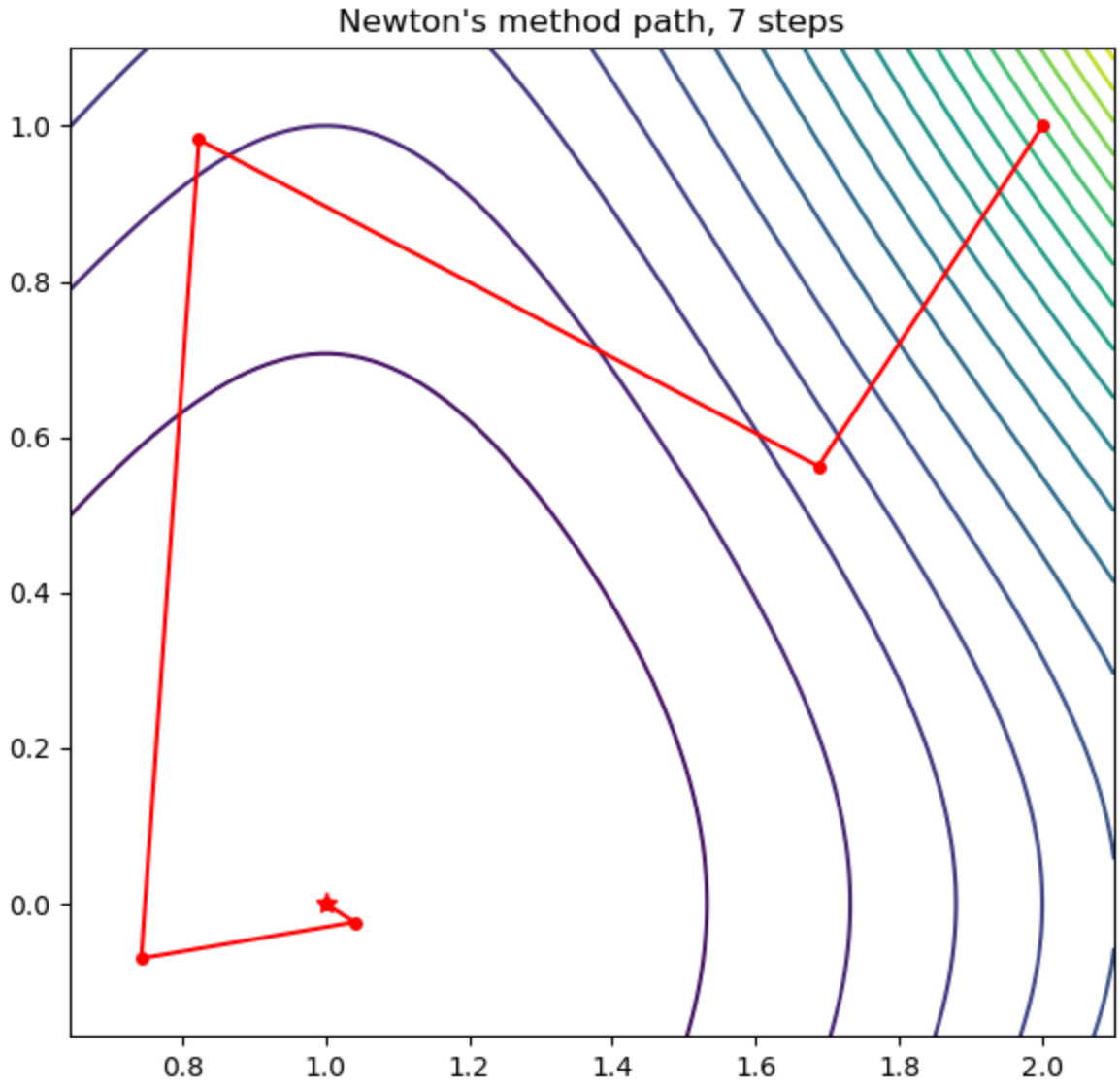
What goes wrong, and why, in your experiment? Specifically, for the "why" part, what

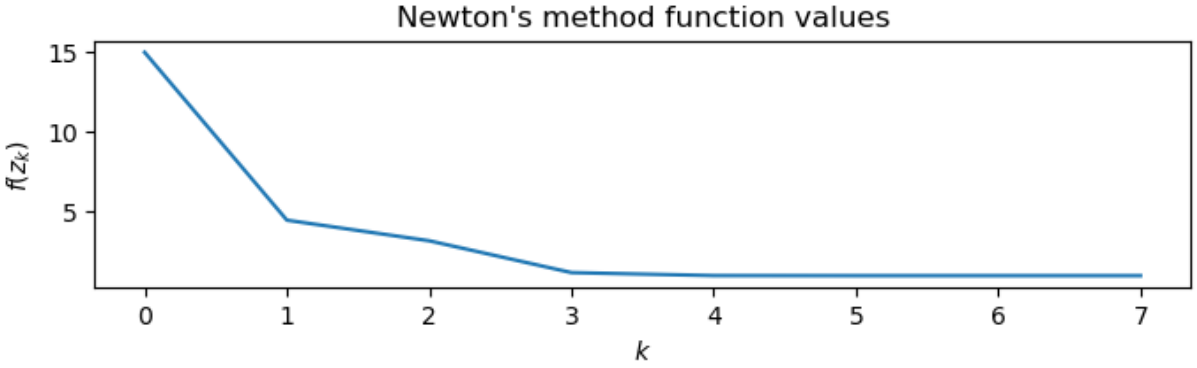
assumptions underlying Newton's method for function minimization are violated?

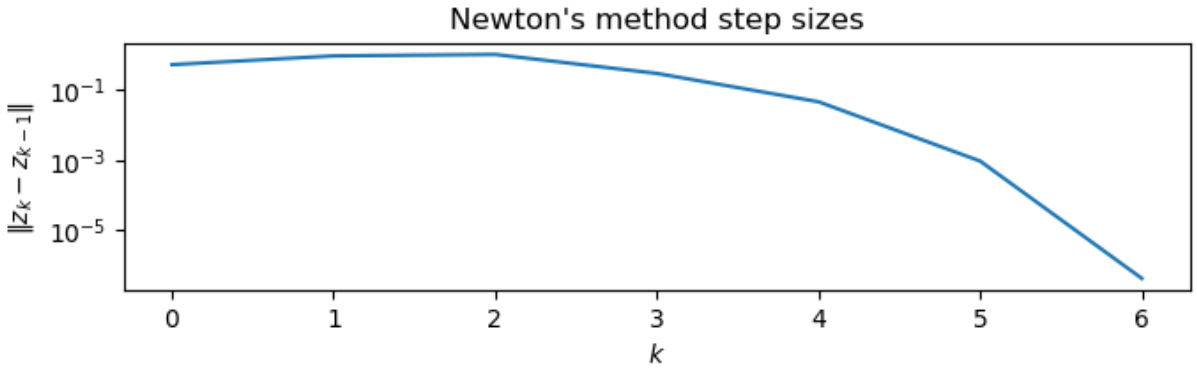
The method converges to $(-1, 0)$ which is a saddle point, not $(1, 0)$, a minimum point. Newton's method is a root-finding method, not a minimum-finding method, it only involves setting the gradient of the second-order Taylor approximation of the function to 0, and then solving the resulting linear system in z. Since a saddle point satisfies Newton's method's requirement that the gradient of the second-order Taylor approximation is equal to 0, just like a minimum point or maximum point does, and our initial point $(-0.5, 2)$ lies in the basin of attraction of the saddle point, Newton's method converges to the saddle point instead of the actual minimum point.

Problem 2.4

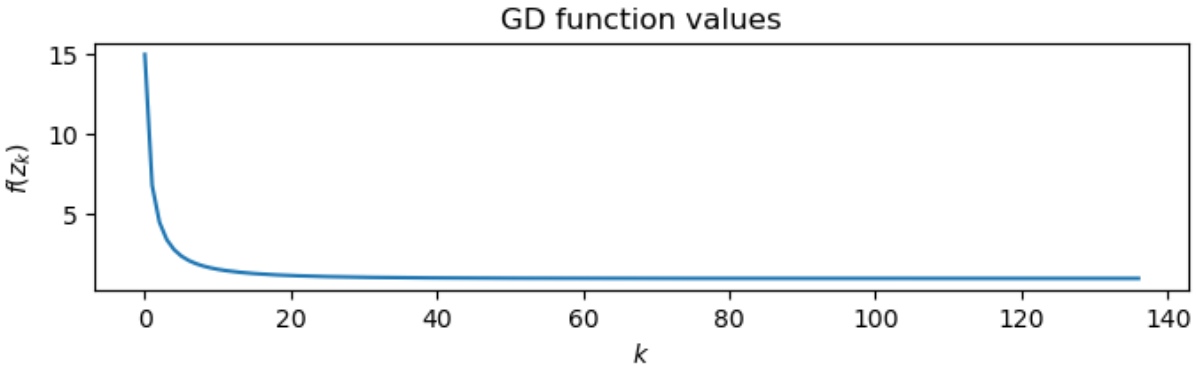
In [14]:
```
new_z_init = np.array([2, 1], dtype=float)
n_2 = Descender(fgh, new_z_init, newton)
history_4 = n_2.descend()

plot_path("Newton's method", history_4, f)
plot_progress("Newton's method", history_4)

learning_rate = 0.01
d_gd_2 = Descender(fg, new_z_init, GDStepper(learning_rate).step)
history_gd_2 = d_gd_2.descend()

plot_progress("GD", history_gd_2)
```

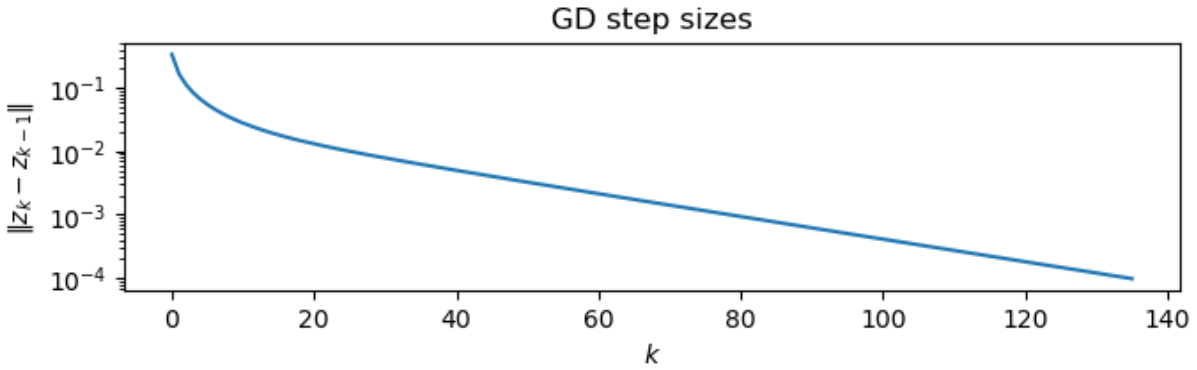

Newton's method path, 7 steps
1.0
0.8
0.6
0.4
0.2
0.0
0.8
1.0
1.2
1.4
1.6
1.8
2.0


Newton's method function values
15
10
5
$f(z_k)$
0
1
2
3
4
5
6
7
k




```
Newton's method: from f(2.00000, 1.00000) = 15.00000 to f(1.00000, -0.00000) = 1.000
00 in 7 iterations
```






```
GD: from f(2.00000, 1.00000) = 15.00000 to f(1.00006, 0.00236) = 1.00001 in 136 iter
ations
```

1. Yes. Newton's method converges to $(1, 0)$, which is the expected minimum point.

2. Yes, Newton's method converges in fewer iterations than gradient descent. Newton's method can be seen to have locally quadratic convergence from the plot of $f(z_k)$ above. Gradient descent can be seen to have approximately linear convergence from the plot of $f(z_k)$ above. We can see from the plots of $f(z_k)$ that Newton's method approaches the neighborhood of the minimum point $(1, 0)$ in fewer steps than gradient descent.

Part 3: Basics of Stochastic Gradient Descent

Problem 3.1 (Exam Style)

We run one step of Stochastic Gradient Descent (SGD) with mini-batch size 1, i.e. $|B_1| = 1$, on the first training sample $(x_1, y_1) = (1, 6)$. The mini-batch average f_1 is an estimate of the entire average f,

$$f_1(z) = \frac{1}{|B_1|}\phi_1(z) = (b + w - 6)^2$$

The partial derivatives with respect to b and w for that sample are:

$$\frac{\partial}{\partial b}\Big[(b + w - 6)^2\Big] \;=\; 2\big(b + w - 6\big),$$

$$\frac{\partial}{\partial w}\Big[(b + w - 6)^2\Big] \;=\; 2\big(b + w - 6\big)$$

$$\nabla f(z) \approx \nabla f_1(z) = \big(2\big(b + w - 6\big), 2\big(b + w - 6\big)\big)$$

Given $z_0 = (0, 3)$, $\alpha = \frac{1}{6}$,

$$z_1 = z_0 + \alpha \nabla f_1(z_0) \;=\; (0, 3) + \frac{1}{6}(-6, -6) \;=\; (-1, 2)$$

Part 4: Automatic Differentiation and Stochastic Gradient Descent

In [15]:
```
from autograd import grad
```

In [16]:
```
from autograd import numpy as anp
import numpy as snp
```

In [17]:
```
class LRRisk:
    def __init__(self, x, y, batch_size=None):
        assert x.ndim == 2 and y.ndim == 1 and x.shape[0] == y.size, \
            'x must be n by d and y must be a vector of length n'
        self.x, self.y = x, y
        self.n, self.d = x.shape
        self.batch_size = batch_size
        if batch_size is not None:
            assert batch_size <= self.n, 'batch size can be at most n'
            self.rng = snp.random.default_rng(5)
            self.indices = self.rng.permutation(self.n)
            self.batch_interval = (0, batch_size)

    def _next_batch(self):
        if self.batch_size is None:
            return self.x, self.y
        a, b = self.batch_interval
```

```
        idx = self.indices[a:b]
        x_batch, y_batch = self.x[idx], self.y[idx]
        self.batch_interval = (a + self.batch_size, snp.minimum(b + self.batch_size
        if b == self.n:
            self.batch_interval = (0, self.batch_size)
            self.indices = self.rng.permutation(self.n)
        return x_batch, y_batch

    def risk(self, z):
        b, w = z[0], z[1:]
        assert w.size == self.d, 'z must have d + 1 entries'
        x, y = self._next_batch()
        losses = (b + anp.dot(x, w) - y) ** 2
        return anp.sum(losses) / y.size
```

In [18]:
```
rng = snp.random.default_rng(3)
n, d = 50, 2
b_star, w_star = rng.uniform(low=-1, high=1), rng.uniform(low=-1, high=1, size=d)
xx = rng.uniform(size=(n, d))
data = {}
for name, sigma in zip(('clean', 'noisy'), (0, 0.01)):
    yy = b_star + snp.dot(xx, w_star) + sigma * rng.normal(size=n)
    data[name] = {'x': xx, 'y': yy}
```

In [19]:
```
def make_risk_function(data_points, batch_size):
    risk_instance = LRRisk(data_points['x'], data_points['y'], batch_size=batch_siz

    def fg(z):
        risk = risk_instance.risk
        g_risk = grad(risk)
        return risk(z), g_risk(z)

    return fg
```

In [20]:
```
def print_result(name, z_ideal, z_actual):
    err = snp.linalg.norm(z_actual - z_ideal)
    print(name + ':')
    with snp.printoptions(precision=6):
        if z_ideal.size < 5:
            print('\tideal solution  {}'.format(z_ideal))
            print('\tactual solution {}'.format(z_actual))
        print('\terror {:.3g}'.format(err))
```

In [21]:
```
z_star = snp.concatenate(([b_star], w_star))

# Initialize z
dim = data['clean']['x'].shape[1]
z0 = anp.zeros(dim + 1)

# Initialize the Descender
learning_rate = 0.1
stepper = GDStepper(learning_rate).step
risk_and_grad = make_risk_function(data['clean'], batch_size=None)
d = Descender(risk_and_grad, z0, stepper, delta=1.e-5, record=False)
```

```
# Run the Descender and print results
z_ast, fz_ast = d.descend()
print_result('first experiment', z_star, z_ast)
```

```
first experiment:
        ideal solution  [-0.828702 -0.526379  0.602549]
        actual solution [-0.827898 -0.527118  0.601787]
        error 0.00133
```

Problem 4.1 (Exam Style)

No, I would not expect different results from the ones in the first experiment above.

One sentence:
Since batch_size = None in the first experiment, when risk_instance = LRRisk(data_points['x'], data_points['y'], batch_size=batch_size) creates an LRRisk instance, and _next_batch() is called, x and y are simply set to the entire dataset of parameters, data_points['x'], data_points['y'], effectively yielding just one batch.

Problem 4.2

In [22]:
```
def SGD(data_points, risk_fn, z_init, batch_size, delta, alpha=0.1, max_epochs=3000
    n = data_points['x'].shape[0]
    if batch_size is None:
        steps_per_epoch = 1
    else:
        steps_per_epoch = int(np.ceil(n / batch_size))

    stepper = GDStepper(alpha).step
    d = Descender(risk_fn, z_init, stepper, delta = 0, max_iterations=steps_per_epo

    z_prev = z_init.copy()
    for epoch in range(max_epochs):
        z_epoch, fz_epoch = d.descend()
        if np.linalg.norm(z_epoch - z_prev) < delta:
            return z_epoch

        z_prev = z_epoch
        d = Descender(risk_fn, z_epoch, stepper, delta = 0, max_iterations=steps_pe

    print(f"Reached max_epochs={max_epochs} without satisfying ||Δz||<{delta}")
    return z_prev


z0 = anp.zeros(dim + 1)

for dp_name in ["clean", "noisy"]:
    for bs in [None, 10]:
        fg = make_risk_function(data[dp_name], bs)
        z_sgd = SGD(data[dp_name], fg, z0, batch_size=bs, delta=1e-5, alpha=0.1, ma

        label = f"{dp_name}, bs={bs}"
        print_result(label, z_star, z_sgd)
```

```
clean, bs=None:
        ideal solution  [-0.828702 -0.526379  0.602549]
        actual solution [-0.827898 -0.527118  0.601787]
        error 0.00133
clean, bs=10:
        ideal solution  [-0.828702 -0.526379  0.602549]
        actual solution [-0.828538 -0.526532  0.602399]
        error 0.00027
noisy, bs=None:
        ideal solution  [-0.828702 -0.526379  0.602549]
        actual solution [-0.830988 -0.525068  0.608933]
        error 0.00691
Reached max_epochs=3000 without satisfying ||Δz||<1e-05
noisy, bs=10:
        ideal solution  [-0.828702 -0.526379  0.602549]
        actual solution [-0.831238 -0.524245  0.608521]
        error 0.00683
```

The SGD algorithm gives better results on clean data, by a factor of approximately 5. On noisy data, although SGD seems to give slightly better results, the error is approximately of the same order of magnitude as the error from "plain" GD.