ELEN90088 System Optimisation and Machine Learning, 2025

Exercise 1

Due date: 23:59, Monday the 31st March, 2025

Submission guidline:

- One submission per group by the due date on LMS.
- Answer the exercise questions in this Python notebook itself.
- Export your notebook file (.ipynb) as a PDF file, on which we give marks and comments. This means that each group should submit two versions of the exercise report (.ipynb file and PDF).
- Demonstrators will conduct a brief oral assessment for selected groups in subsequent workshop. Details will be announced on LMS.
- Regarding the use of LLM and other generative AI tools: refer to information in the introductory slides.
- This exercise contains one question (Question 7) with 2 bonus marks. So the highest marks you could obtain from this Exercise are $10 + 2 = 12$ marks.

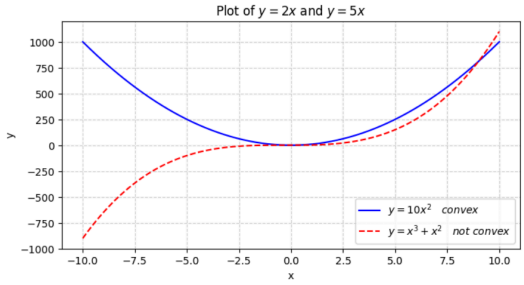
Question 1 (plot convex and non-convex functions) (Mark: 1 point)

Practice Question

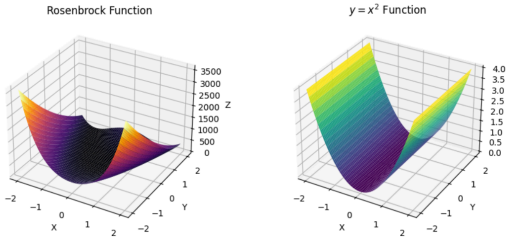
Plot one concave and one non-convex (neither convex nor concave) function of your choosing (preferably one in 2 dimensions and the other in 3 dimensions so that it can be visualised, check e.g. this tutorial for hints). Provide their formulas below.

Hint: an interesting and well-known function is [Rosenbrock function]. (https://en.wikipedia.org/wiki/Rosenbrock_function) It is already built-in to Scipy optimize as a benchmark. *You can keep your answer simple and don't need to spend too much time on this practice question.*

Two-dimensional functions:



Three-dimensional functions:



In [ ]:

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# 3D plots
# Define the Rosenbrock function
def rosenbrock(x, y, a=1, b=100):
    return (a - x)**2 + b * (y - x**2)**2

# Define y = x^2 function
def parabola(x):
    return x**2

# Create grid
x = np.linspace(-2, 2, 400)
y = np.linspace(-2, 2, 400)
X, Y = np.meshgrid(x, y)

# Compute Rosenbrock function values
Z_rosenbrock = rosenbrock(X, Y)  # non-convex nor concave

# Compute y = x^2 function values
Z_parabola = parabola(X) #convex

# Create 3D plot
fig = plt.figure(figsize=(10, 4))

# Plot Rosenbrock function
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(X, Y, Z_rosenbrock, cmap='inferno')
ax1.set_title('Rosenbrock Function')
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('Z')

# Plot y = x^2 function
ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(X, Y, Z_parabola, cmap='viridis')
ax2.set_title('$y = x^2$ Function')
ax2.set_xlabel('X')
```

```
ax2.set_ylabel('Y')
ax2.set_zlabel('Z')

# Show plot
plt.tight_layout()
plt.show()

# 2D plots
# Define x values
x = np.linspace(-10, 10, 400)

# Define y expressions
y1 = 10 * x ** 2  # convex
y2 = x ** 3 + x ** 2  # non-convex function

# Create 2D plot
plt.figure(figsize=(8, 4))
plt.plot(x, y1, label=r'$y = 10x^2 \quad convex$', linestyle='-', color='b')
plt.plot(x, y2, label=r'$y = x^3 + x^2 \quad not\ convex$', linestyle='--', colo

# Add labels and title
plt.xlabel('x')
plt.ylabel('y')
plt.title('Plot of $y = 10x^2$ and $y = x^3 + x^2$')

# Add grid
plt.grid(True, linestyle='--', alpha=0.6)

# Add legend
plt.legend()

# Show plot
plt.show()
```

Question 2 (decide if a polynomial is convex) (Mark: 2 points)

Consider a polynomial with 2 variable

$$p(x) = \sum_{i,j:i+j\leq d} c_{ij} x_1^i x_2^j$$

where the sum is over all nonnegative integer pairs i, j whose sum is less or equal to d. We call d the *degree* of the polynomial.

To understand the notation better, consider a polynomial of degree 2 given by $h(x) = x_1 x_2 + 2x_2^2 + 3x_1 - 1$. You should convince yourself that $h(x)$ can be obtained by setting $c_{11} = 1, c_{02} = 2, c_{10} = 3, c_{00} = -1$ and all other coefficients c_{ij} to 0 in $p(x)$.

Answer the following questions:

1. how to decide the convexity of a first-order ($d = 1$) polynomial?
2. how to decide the convexity of a quadratic ($d = 2$) polynomial?

3. how to decide the convexity of a cubic ($d = 3$) polynomial?
4. extend the result in (3) to all odd-order (d is odd) polynomial.
5. how to decide the convexity of a quartic ($d = 4$) polynomial? What is the difference/difficulty compared to the quadratic case? In particular, try to plot a convex quartic polynomial and a concave quartic polynomial in 1 variable.

Comment: as shown in this paper, unless NP=P, it is computationally hard to check the convexity of a polynomial with order equal or larger than 4. So in general, it is computationaly difficult to decide if a function is convex.

1

The first-order polynomial function can be called a linear function. Linear functions are convex and concave because the second derivative is zero, satisfying both the convexity and concavity conditions.

2

The convexity of a quadratic function can be determined using the Hessian matrix (the matrix of second-order partial derivatives). The polynomial is convex if and only if the Hessian is positive semidefinite.

3

Cubic polynomials can have inflection points where the convexity changes. For single-variable case, compute the second derivative $f''(x)$. If it is non-negative for all x, the polynomial is convex, but cubics always fail this due to inflection points. \ For multivariable case, construct the Hessian matrix $H(x)$; check if it is positive semi-definite for all x (though cubics cannot satisfy this globally since $H(x)$ varies linearly with x). No cubic is globally convex; only piecewise analysis is possible.

4

Odd-order polynomials (degree 3, 5, etc.) generally have regions where they are convex and regions where they are concave due to the presence of inflection points. They are not convex or concave globally.

5

Determining convexity requires checking the second derivative over the entire domain, which can be computationally intensive.

Question 3 (Optimization package) (Mark: 1 point)

CVXPY is an open source Python-embedded modeling language for convex optimization problems. This exercise aims to familiarze you with this package.

*Note: you are **not** required to use CVXPY for your exercises or project. There are other Python-based optimization package you can use, including Scipy or Pyomo. If you decide to use a different optimization package, you can solve this exercise with one you choose. CVXPY lets you express your problem in a natural way that follows the math, rather than in the restrictive standard form required by solvers, but it has its own restrictions. In particular, CVXPY does not support non-convex optimization whereas other packages (e.g. Scipy) does.*

Before addressing this problem, we suggest you read through the first section of the User Guide and a few Basic examples. They should already give you a pretty good idea how a convex optimization problem can be solved using CVXPY.

Consider a simple scalar linear dynamical system

$$x_{t+1} = ax_t + u_t, \quad t = 1, 2, \dots$$

where $|a| > 1$. We call x_t the state of the system, and u_t the control input. The objective is to minimize the cost C defined as

$$C = \sum_{t=1}^{T} x_t^2 + \sum_{t=1}^{T-1} \alpha u_t^2.$$

Given the parameters $x_1 = 1, a = 1.5, \alpha = 2$, and $T = 3$, find the optimal control input u_1, u_2 that minimizes the cost by rewriting the problem as a quadratic optimization problem.

Hint: there are at least two ways to formulate your problem: you can formulate a problem where the optimization variable is just (u_1, u_2); you can also formulate a problem where the optimization variable is $(u_1, u_2, x_1, x_2, x_3)$. See if you can use the second (more interesting) problem formulation.

$C = \sum_{t=1}^{T} x_t^2 + \sum_{t=1}^{T-1} \alpha u_t^2, \quad \alpha = 2, T = 3$

Based on this equation, $C = \sum_{t=1}^{3} x_t^2 + \sum_{t=1}^{2} 2u_t^2$

$C = x_1^2 + x_2^2 + x_3^2 + 2u_1^2 + 2u_2^2$

$x_{t+1} = ax_t + u_t, \quad t = 1, 2, \dots, \quad x_1 = 1, a = 1.5, \alpha = 2$, and $T = 3$

When $t = 1, \quad x_2 = 1.5x_1 + u_1 = 1.5 + u_1$

When $t = 2, \quad x_3 = 15x_2 + u_2 = 1.5^2 + 1.5u_1 + u_2 = 1.5^2 + 1.5u_1 + u_2$

Therefore, $C = 1^2 + (1.5 + u_1)^2 + (1.5^2 + 1.5u_1 + u_2)^2 + 2u_1^2 + 2u_2^2$

In [ ]:
```
import cvxpy as cp
import numpy as np
```

```
# Define the variables
u_1 = cp.Variable()
u_2 = cp.Variable()

# Define the objective function
c = 1 + (1.5 + u_1)**2 + (2.25 + 1.5*u_1 + u_2)**2 + 2*u_1**2 + 2*u_2**2

# Formulate the problem
objective = cp.Minimize(c)
problem = cp.Problem(objective)

# Solve the problem
problem.solve()

# Print the results
print("Optimal value of u_1:", u_1.value)
print("Optimal value of u_2:", u_2.value)
print("Minimum value of c:", c.value)
```

In [ ]:

```
import numpy as np
import cvxpy as cp

# Define parameters
T = 3
a = 1.5
alpha = 2

def compute_c_function(T):
    # Define variables
    u = cp.Variable(T-1)  # Control variable u_t
    x = cp.Variable(T)  # State variable x_t

    # Constraints
    # As we know the value of x, it can be seen as a constraint
    constraints = [x[0] == 1]  # Initial condition
    for t in range(1, T):
        constraints.append(x[t] == a * x[t-1] + u[t-1])  # Add new constraints t

    # Define the cost function
    c = cp.sum_squares(x) + alpha*cp.sum_squares(u)

    return c, u, constraints  # Return function, control variable, and constrain

# Get function c(u), variable u, and constraints
c, u, constraints = compute_c_function(T)

# Solve the optimization problem
prob = cp.Problem(cp.Minimize(c), constraints)
prob.solve()

print("Optimal value of u:", u.value)
print("Minimum value of c:", c.value)
```

Question 4 (Aloha communication protocol) (Mark: 2 points)

Aloha is a well-known random access or *MAC* (Media/multiple Access Control) communication protocol. It enables multiple nodes to share a broadcast channel without any additional signaling in a distributed manner. Unlike *FDMA* or *TDMA* (frequency or time-division multiple access), the channel is not divided into segments beforehand and collisions of packets due to simultaneous transmissions by nodes are allowed.

In slotted Aloha, the nodes can only transmit at the beginning of time slots, which are kept by a global/shared clock. The operations of slotted ALOHA in each node are decribed as follows:

- If there is only one frame in a slot, then the transmission is successful and the slot is said to be a successful slot.
- If two or more frames collide in a slot, then the transmission is failed and a re-transmission is considered for each frame involved in the collision. The node will retransmit its frame in each subsequent slot with probability p until the frame is transmitted without a collision, where $0 \leq p \leq 1$.

Assume that there are N nodes and each node independently attempts to transmit a frame in each slot with probability p. For one slot, let S denote the *success probability* that this slot is a successful slot.

Answer the following questions:

1. Provide an expression of S defined above.
2. To maximise the S, define the optimisation problem to find the optimal value p. Note that the constraint $0 \leq p \leq 1$ should be taken into consideration. Clearly identify the objective and decision variable(s). Is the objective convex or concave? Show through derivation. Find the optimality conditions for this problem.
3. When N tends to infinity, what is the maximal success probability S?

1

$S = N \cdot p \cdot (1-p)^{N-1}$

2

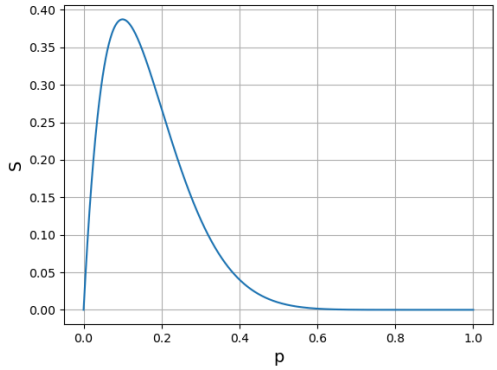
Objective function: $S(p) = N \cdot p \cdot (1-p)^{N-1}$

Decision variable: p

Constraint: $0 \leq p \leq 1$

Because the first derivative $S'(p) = N \cdot (1-p)^{N-2} \cdot (1-pN)$ and the second derivative $S''(p) = N \cdot (1-p)^{N-3} \cdot (N-1)(pN-2)$. When $p > \frac{2}{N}$, $S''(p) > 0$, $S(p)$ is convex. When $p < \frac{2}{N}$, $S''(p) < 0$, $S(p)$ is concave. However, the optimal solution is $S'(p) = 0$ when $p = \frac{1}{N}$.

In conclusion, $S(p)$ is concave when $0 \leq p \leq \frac{1}{N}$ and $p = \frac{1}{N}$ is a global maximum. $S(p)$ is convex when $\frac{2}{N} < p \leq 1$.



3

When $p = \frac{1}{N}$, $S_{max} = (1 - \frac{1}{N})^{N-1}$.
$\lim_{N\to\infty}(1 - \frac{1}{N})^{N-1} = \lim_{N\to\infty}[(1 - \frac{1}{N})^{N} \cdot (1 - \frac{1}{N})^{-1}] = \frac{1}{e}$. The maximum success probability $S = \frac{1}{e}$.

```
(ELEN90088) (base) mjh@maojunhengdeMBP exercise1 % python3 question4.py
The maximum value of S ≈ 0.368063
```

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt

def plot_graph(N):
    p = np.linspace(0, 1, 1000)
    S = N * p * (1 - p)**(N - 1)
    plt.plot(p, S, label=f'N = {N}')
    plt.xlabel('p', fontsize=14)
    plt.ylabel('S', fontsize=14)
    plt.grid(True)
    plt.show()

def maximum_s(N):
    p = 1 / N
    S_max = N * p * (1 - p) ** (N - 1)
    print(f"The maximum value of S ≈ {S_max:.6f}")

if __name__ == "__main__":
    plot_graph(10)
    maximum_s(10000)
```

Question 5 (Geometric programming: power control in wireless network) (Mark: 2 points)

Example: Power Control in Wireless Communication

Adapted from Boyd, Kim, Vandenberghe, and Hassibi, "A Tutorial on Geometric Programming."

The power control problem in wireless communications aims to minimise the total transmitter power available across N trasmitters while concurrently achieving good (or a pre-defined minimum) performance.

The technical setup is as follows. Each transmitter i transmits with a power level P_i bounded below and above by a minimum and maximum level. The power of the signal received from transmitter j at receiver i is $G_{ij}P_j$, where $G_{ij} > 0$ represents the path gain (often loss) from transmitter j to receiver i. The signal power at the intended receiver i is $G_{ii}P_i$, and the interference power at receiver i from other transmitters is given by $\sum_{k\neq i} G_{ik}P_k$. The (background) noise power at receiver i is σ_i. Thus, the *Signal to Interference and Noise Ratio (SINR)* of the ith receiver-transmitter pair is

$$S_i = \frac{G_{ii}P_i}{\sum_{k\neq i} G_{ik}P_k + \sigma_i}.$$

The minimum SINR represents a performance lower bound for this system, $S^{\min}$.

The resulting optimisation problem is formulated as

$$\begin{array}{ll} \min_P & \sum_{i=1}^N P_i \\ \text{subject to} & P^{min} \leq P_i \leq P^{max},\ \forall i \\ & \dfrac{G_{ii}P_i}{\sigma_i + \sum_{k\neq i} G_{ik}P_k} \geq S^{min},\ \forall i \end{array}$$

Answer the following questions: Wireless Power Control

Let $N = 10$, $P^{min} = 0.1$, $P^{max} = 5$, $\sigma = 0.2$ (same for all). Create a random path loss matrix G, where off-diagonal elements are between 0.1 and 0.9 and the diagonal elements are equal to 1.

1. Convert the problem into standard Geometric Programming form.
2. Solve the problem first with $S^{min} = 0$, and we recommand to use *cvxpy* or *scipy*. Output and plot the power levels and SINRs that you obtain.
3. What happens if you choose an S^{min} that is larger? Solve the problem again and document your results. What happens if you choose a very large S^{min}? Observe and comment.
4. Suppose that there are only 3 transmitter-receiver pairs, e.g., $N = 3$, if the SINR of the ith receiver-transmitter pair is

$$S_i = \frac{G_{ii}P_i}{0.5\sum_{j\neq i}^N (G_{ij}P_j \sum_{k\neq i, k\neq j}^N G_{ik}P_k) + \sigma_i}.$$

and other settings remain the same, please try to convert the problem into standard Geometric Programming form again.

Note:

- if you are in the minority of people who have problem installing *cvxpy* or *scipy*, then you can use other packages or even Matlab.
- the problem in (1) can be converted to either Geometric Programming or Linear Programming form, while the problem in (4) can not be converted to Linear Programming anymore.

1

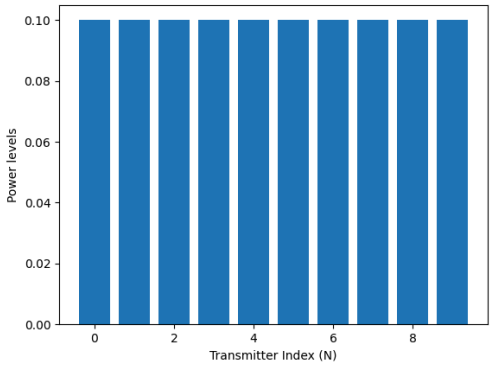
Geometric Programming:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{10} P_i \\
\text{s.t.} \quad & \frac{0.1}{P_i} \leq 1, \quad \forall i \in \{1, 2, \ldots, N\} \\
& \frac{P_i}{5} \leq 1, \quad \forall i \in \{1, 2, \ldots, N\} \\
& \frac{\sum_{k \neq i} G_{ik} P_k + 0.2}{G_{ii} P_i / S^{\min}} \leq 1, \quad \forall i \in \{1, 2, \ldots, N\}
\end{aligned}
$$

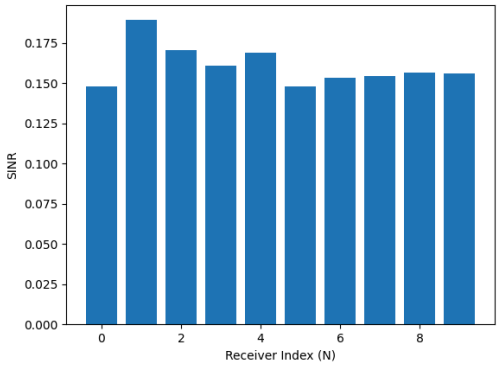
2

Using cvxpy:

Poer levels:



SINRS:



Results of power levels and SINRS:

```
(ELEN90088) (base) mjh@maojunhengdeMBP exercise1 % python3 question5.py
Power levels:  [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
SINR:  [0.15, 0.19, 0.17, 0.16, 0.17, 0.15, 0.15, 0.15, 0.16, 0.16]
```

3

If $S_{\min}$ becomes larger, P_i and the total power will become larger. If $S_{\min}$ is very large, the solution might be infeasible.

```
(ELEN90088) (base) mjh@maojunhengdeMBP exercise1 % python3 question5.py
S_min = 0.1: Status = optimal, Power levels = [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
S_min = 0.2: Status = optimal, Power levels = [0.29045521 0.21913068 0.25394392 0.2701135  0.25368757 0.29068672
 0.28459942 0.28525934 0.27726345 0.27294005]
S_min = 0.5: Status = infeasible, Power levels = None
S_min = 1.0: Status = infeasible, Power levels = None
S_min = 2.0: Status = infeasible, Power levels = None
S_min = 10.0: Status = infeasible, Power levels = None
```

From the graph above, we can observe that P_i becomes larger when $S_{\min}$ becomes larger. However, when $S_{\min} > 0.5$, the solutions become infeasible.

4

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{3} P_i \\
\text{s.t.} \quad & \frac{0.1}{P_i} \leq 1, \quad \forall i \in \{1, 2, \ldots, N\} \\
& \frac{P_i}{5} \leq 1, \quad \forall i \in \{1, 2, \ldots, N\} \\
& \frac{0.5(\sum_{j \neq i}^{3} G_{ij} P_j \sum_{k \neq i, k \neq j}^{3} G_{ik} P_k) + 0.2}{G_{ii} P_i / S^{\min}} \leq 1, \quad \forall i \in \{1, 2, \ldots, N\}
\end{aligned}
$$

In [ ]:
```python
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt

def obj_fun():
    # matrix G
    np.random.seed(42)
    G = np.random.uniform(0.1, 0.9, (N, N))
    np.fill_diagonal(G, 1.0)

    P = cp.Variable(N, pos=True)
    objective = cp.Minimize(cp.sum(P))

    # SINR is not a constraint when S_min = 0
    constraints = [P >= P_min, P <= P_max]

    prob = cp.Problem(objective, constraints)
    prob.solve(gp=True)

    SINR = [G[i,i] * P.value[i] / (sigma + sum(G[i,k] * P.value[k] for k in rang

    print("Power levels: ", P.value)
    print("SINR: ", SINR)

    return P.value, SINR
```

```
def obj_fun_large_sm(N, P_min, P_max, sigma, S_min):
    np.random.seed(42)

    G = np.random.uniform(0.1, 0.9, (N, N))
    np.fill_diagonal(G, 1.0)

    P = cp.Variable(N, pos=True)
    objective = cp.Minimize(cp.sum(P))
    constraints = [P_min / P <= 1.0, P / P_max <= 1.0]

    for i in range(N):
        tmp = cp.sum([G[i, k] * P[k] for k in range(N) if k != i])
        sinr_inv_expr = (sigma + tmp) / (G[i, i] * P[i])
        constraints.append(sinr_inv_expr <= 1.0 / S_min)

    prob = cp.Problem(objective, constraints)
    prob.solve(gp = True)

    print(f"S_min = {S_min}: Status = {prob.status}, Power levels = {P.value}")

def plot_graph(P_opt, SINR):
    plt.figure()
    plt.bar(range(N), P_opt)
    plt.xlabel('Transmitter Index (N)')
    plt.ylabel('Power levels')
    plt.show()

    plt.figure()
    plt.bar(range(N), SINR)
    plt.xlabel('Receiver Index (N)')
    plt.ylabel('SINR')
    plt.show()

if __name__ == "__main__":
    N = 10
    P_min = 0.1
    P_max = 5
    sigma = 0.2
    # S_min = 0
    S_min = [0.1, 0.2, 0.5, 1.0, 2.0, 10.0]

    # P_opt, SINR = obj_fun()
    # plot_graph(P_opt, SINR)
    for sm in S_min:
        obj_fun_large_sm(N, P_min, P_max, sigma, sm)
```

Question 6 (Gradient descent) (Mark: 2 points)

Consider the Gradient descent (GD) algorithm:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \alpha_t \nabla f(\boldsymbol{x}_t),$$

where $\boldsymbol{x}_t \in \mathbb{R}^n$ is the variable vector at the t-th iteration and α_t is the step size. In this question, we investigate the impact of the step size α_t adopted in the GD

algorithm. Consider the following two objective functions:

1. Quadratic function: $f(x_1, x_2) = x_1^2 + x_1 + 2x_2^2$.
2. Non-convex function: $f(x_1, x_2) = \sin(x_1) + 0.5x_2^2$.

Minimise them by implementing the GD algorithm, with various choices of step size as listed below:

1. Constant step size: The step size is fixed throughout the optimization.
2. Vanishing step size: The step size decreases over time, i.e., $\alpha_t = \frac{\alpha_0}{1+\lambda t}$, where α_0 is the initial step size, t is the iteration index and λ is the decay rate.
3. Backtracking line search: The step size is updated by

   $$\alpha = \beta * \alpha$$

   as long as the following Armijo condition is satisfied

   $$f(\boldsymbol{x}_t - \alpha \nabla f(\boldsymbol{x}_t)) > f(\boldsymbol{x}_t) - \frac{\alpha}{2}\|\nabla f(\boldsymbol{x}_t)\|^2,$$

   where $0 < \beta < 1$ is a shrinkage factor.
4. Exact line search (only for the quadratic function in (1)): This finds the optimal step size at each iteration, i.e.,

   $$\min_{\alpha \geq 0} f(\boldsymbol{x}_t - \alpha \nabla f(\boldsymbol{x}_t))$$

Implement GD with different choices of step size. You can experiment with different choices of parameters, and think about how to best illusrate and compare your results. For example, consider how many iterations you should run, and what stopping criterion you should take.

The minimum value of quadratic function is approaching -0.25, when $x_1 = -0.5$ and $x_2 = 0.0$. The minimum value of non-convex function is approaching -1.0, when $x_1 = -1.56$ and $x_2 = 0.0$.

Results of the four methods:

```
(ELEN90088) C:\Users\CGN-NME\Desktop\MJH\Unimelb\25s1\ELEN90088\exercise\exercise1>D:/anaconda/envs/ELEN90088/python.exe c:/Users/CGN-NME/Desktop/MJH/Unimelb/25s1/ELEN90088/exercise/exercise1/question6.py
Constant step size (func_1): (-0.4951051208185936), 0.0, -0.2499768402828565)
Constant step size (func_2): (-1.5689434191476688, 0.0, -0.9999514604981332)
Vanishing step size (func_1): (-0.4875615384615386, 0.0, -0.2498452846745562)
Vanishing step size (func_2): (-1.3907293935995757, 0.0, -0.9838317875628883)
Backtracking line search (func_1): (-0.4951051208185936), 0.0, -0.2499768402828565)
Backtracking line search (func_2): (-1.5689434191476688, 0.0, -0.9999514604981332)
Exact line search (func_1): (-0.4999999956746621 3, 0.0,  -0.2499999999999997)
```

1 Constant step size

There are four parameters: step size, iterations, initial values, and stopping criterion. We change the value of α (step size). If α is too large, it may oscillate or even diverge; if α is too small, the convergence rate is slow. For example, when the iteration is 1000 and the initial values are $x_1 = x_2 = 0.0$, if α is 1.0, the minimum value is -0.99; if α is 0.0001, the minimum value is -0.08.

2 Vanishing step size

There are three additional parameters: initial step size, decay rate, and iteration index. We change the value of α (step size). In the early stage, the step size is large and the minimum value decreases rapidly. In the late stage, the step size is small and value has stable convergence.

3 Backtracking line search

There is an additional parameter: shrinkage factor β. This method ensures sufficient decrease in each iteration by shrinking α until the Armijo condition is met. Larger β values reduce shrinkage per step but may require more Armijo checks; smaller β aggressively shrink the step size, potentially slowing convergence.

4 Exact line search

Exact line search computes the optimal step size at each step by minimizing $f(x_t - \alpha \nabla f(x_t))$. For the quadratic function, the optimal step size has a closed-form solution derived from the Hessian, which can guarantee the steepest descent per iteration, leading to rapid convergence. However, this method can not be used for non-convex functions due to the lack of analytical solutions.

In [ ]:
```python
import numpy as np
from scipy.optimize import minimize

def obj_fun_1(x1, x2):
    return x1**2 + x1 + 2 * x2**2

def obj_fun_2(x1, x2):
    return np.sin(x1) + 0.5 * x2**2

def deriv_obj_fun_1(x1, x2):
    return (2 * x1 + 1, 4 * x2)

def deriv_obj_fun_2(x1, x2):
    return (np.cos(x1), x2)

# question1
def gradient_decs_1(n): # n is the iteration
    alpha = 0.01 # step size
    x1, x2 = 0, 0 # initial value
    f1 = obj_fun_1(x1, x2)
    for i in range(n):
        deriv1, deriv2 = deriv_obj_fun_1(x1, x2)
        x1 = x1 - alpha * deriv1
        x2 = x2 - alpha * deriv2
        f2 = obj_fun_1(x1, x2)
        if f1 - f2 < 1e-6: # stopping criterion
            return float(x1), float(x2), float(f2)
        if f1 > f2:
            f1 = f2
    return float(x1), float(x2), float(f2)

def gradient_decs_2(n):
    alpha = 0.01
```

```
    x1, x2 = 0, 0
    f1 = obj_fun_2(x1, x2)
    for i in range(n):
        deriv1, deriv2 = deriv_obj_fun_2(x1, x2)
        x1 = x1 - alpha * deriv1
        x2 = x2 - alpha * deriv2
        f2 = obj_fun_2(x1, x2)
        if f1 - f2 < 1e-6:
            return float(x1), float(x2), float(f2)
        if f1 > f2:
            f1 = f2
    return float(x1), float(x2), float(f2)

# question2
def gradient_decs_3(n):
    alpha_0 = 0.01 # initial step size
    lamda = 0.01 # decay rate
    x1, x2 = 0, 0
    f1 = obj_fun_1(x1, x2)
    for i in range(n):
        deriv1, deriv2 = deriv_obj_fun_1(x1, x2)
        alpha = alpha_0 / (1 + lamda * i)
        x1 = x1 - alpha * deriv1
        x2 = x2 - alpha * deriv2
        f2 = obj_fun_1(x1, x2)
        if f1 - f2 < 1e-6:
            return float(x1), float(x2), float(f2)
        if f1 > f2:
            f1 = f2
    return float(x1), float(x2), float(f2)

def gradient_decs_4(n):
    alpha_0 = 0.01
    lamda = 0.01
    x1, x2 = 0, 0
    f1 = obj_fun_2(x1, x2)
    for i in range(n):
        deriv1, deriv2 = deriv_obj_fun_2(x1, x2)
        alpha = alpha_0 / (1 + lamda * i)
        x1 = x1 - alpha * deriv1
        x2 = x2 - alpha * deriv2
        f2 = obj_fun_2(x1, x2)
        if f1 - f2 < 1e-6:
            return float(x1), float(x2), float(f2)
        if f1 > f2:
            f1 = f2
    return float(x1), float(x2), float(f2)

# question3
def gradient_decs_5(n):
    alpha = 0.01
    beta = 0.7 # shrinkage factor
    x1, x2 = 0, 0
    f1 = obj_fun_1(x1, x2)
    for i in range(n):
        deriv1, deriv2 = deriv_obj_fun_1(x1, x2)
        while True:
            x1 = x1 - alpha * deriv1
            x2 = x2 - alpha * deriv2
            f2 = obj_fun_1(x1, x2)
```

```
            # Armijo condition
            if f2 <= f1 - (alpha / 2) * (deriv1**2 + deriv2**2):
                break
            alpha *= beta
        if f1 - f2 < 1e-6:
            return float(x1), float(x2), float(f2)
        if f1 > f2:
            f1 = f2
    return float(x1), float(x2), float(f2)

def gradient_decs_6(n):
    alpha = 0.01
    beta = 0.7
    x1, x2 = 0, 0
    f1 = obj_fun_2(x1, x2)
    for i in range(n):
        deriv1, deriv2 = deriv_obj_fun_2(x1, x2)
        while True:
            x1 = x1 - alpha * deriv1
            x2 = x2 - alpha * deriv2
            f2 = obj_fun_2(x1, x2)
            if f2 <= f1 - (alpha / 2) * (deriv1**2 + deriv2**2):
                break
            alpha *= beta
        if f1 - f2 < 1e-6:
            return float(x1), float(x2), float(f2)
        if f1 > f2:
            f1 = f2
    return float(x1), float(x2), float(f2)

# question4
def gradient_decs_7(n):
    x1, x2 = 0, 0
    f1 = obj_fun_1(x1, x2)
    for i in range(n):
        deriv1, deriv2 = deriv_obj_fun_1(x1, x2)

        # line search
        def objective(alpha):
            x1_new = x1 - alpha * deriv1
            x2_new = x2 - alpha * deriv2
            return obj_fun_1(x1_new, x2_new)

        res = minimize(objective, x0 = 0, bounds = [(0, None)])
        alpha = res.x[0] # optimal step size

        x1 = x1 - alpha * deriv1
        x2 = x2 - alpha * deriv2
        f2 = obj_fun_1(x1, x2)
        if f1 - f2 < 1e-6:
            return float(x1), float(x2), float(f2)
        if f1 > f2:
            f1 = f2
    return float(x1), float(x2), float(f2)

if __name__ == "__main__":
    print(f"Constant step size (func_1): {gradient_decs_1(1000)}")
    print(f"Constant step size (func_2): {gradient_decs_2(1000)}")
    print(f"Vanishing step size (func_1): {gradient_decs_3(1000)}")
    print(f"Vanishing step size (func_2): {gradient_decs_4(1000)}")
```

```
print(f"Backtracking line search (func_1): {gradient_decs_5(1000)}")
print(f"Backtracking line search (func_2): {gradient_decs_6(1000)}")
print(f"Exact line search (func_1): {gradient_decs_7(1000)}")
```

Question 7 (Design problem) (Bonus mark: 2 points)

The aim of this exercise is to improve your ability to identify, model, and solve optimization problems by applying mathematical techniques to real-world scenarios.

Step 1. Choose a (possibly engineering-related) problem from your coursework, personal interests, or daily experiences that can be addressed through optimization.

Step 2. Translate the chosen scenario into a mathematical optimization problem by:

1. Defining variables: Identify the variables that can be controlled or adjusted.
2. Establishing the objective function: determine the function that needs to be maximized or minimized (e.g., cost, time, efficiency).
3. setting Constraints: List the limitations or requirements that must be satisfied (e.g., resource limits, safety standards).

Step 3. Solve the optimization. Use a solver (e.g. via scipy) if you need to do it numerically.

Your answer to this question should provide sufficient deetails to all three steps above.

1

We want to build a rectangular place against a wall using 60 meters of fencing material. Because one side (the long side) is along the wall, we only need to build three sides:

One length side l

Two width sides w

We want to maximize the area of the rectangle: $A = l \cdot w$.

2

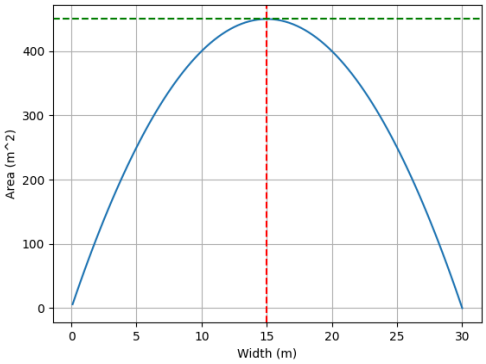
Variables: l, w

Objective function: $A(l, w) = l \cdot w$

Constraints: $l + 2w \leq 60$

3

Results of w and A:



The optimal values of w and l, the maximum value of A:

```
(ELEN90088) mjh@maojunhengdeMacBook-Pro exercise1 % python3 question7.py
Length: 30.000012900212838
Width: 14.99999320483684
Area: 449.99998964821015
```

From the graphs above, we can observe that when $l = 30m$ and $w = 15m$, $A_{\max} = 450m^2$.

In [ ]:
```python
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt

l = cp.Variable(pos=True)
w = cp.Variable(pos=True)

constraints = [l + 2*w <= 60]

objective = cp.Maximize(l * w)
problem = cp.Problem(objective, constraints)
problem.solve(gp=True)

print("Length:", l.value)
print("Width:", w.value)
print("Area:", l.value * w.value)

w = np.linspace(0.1, 30, 100)
l = 60 - 2 * w
A = l * w
plt.plot(w, A, label='Area = w * (60 - 2w)')
plt.axvline(15, color='red', linestyle='--', label='Optimal width = 15m')
plt.axhline(450, color='green', linestyle='--', label='Max area = 450 m²')
plt.xlabel('Width (m)')
plt.ylabel('Area (m^2)')
plt.grid(True)
plt.show()
```