Q3. Quadratic programming

python code for QP using Gradient Descent

In [52]:
```
import numpy as np
from scipy.optimize import minimize
```

In [59]:
```
# QP and constraints
Q = np.array([[2, 1],
              [1,4]])
c = np.array([5, 3])
A = np.array([[1, 0],
              [-1, 0],
              [0, 1],
              [0, 1]])
b = np.array([[1],
              [1],
              [1],
              [1]])
```

In [54]:
```
def objective(x):
    return 0.5 * x @ (Q @ x) + c @ x
```

In [55]:
```
def gradient(x):
    return Q @ x + c
```

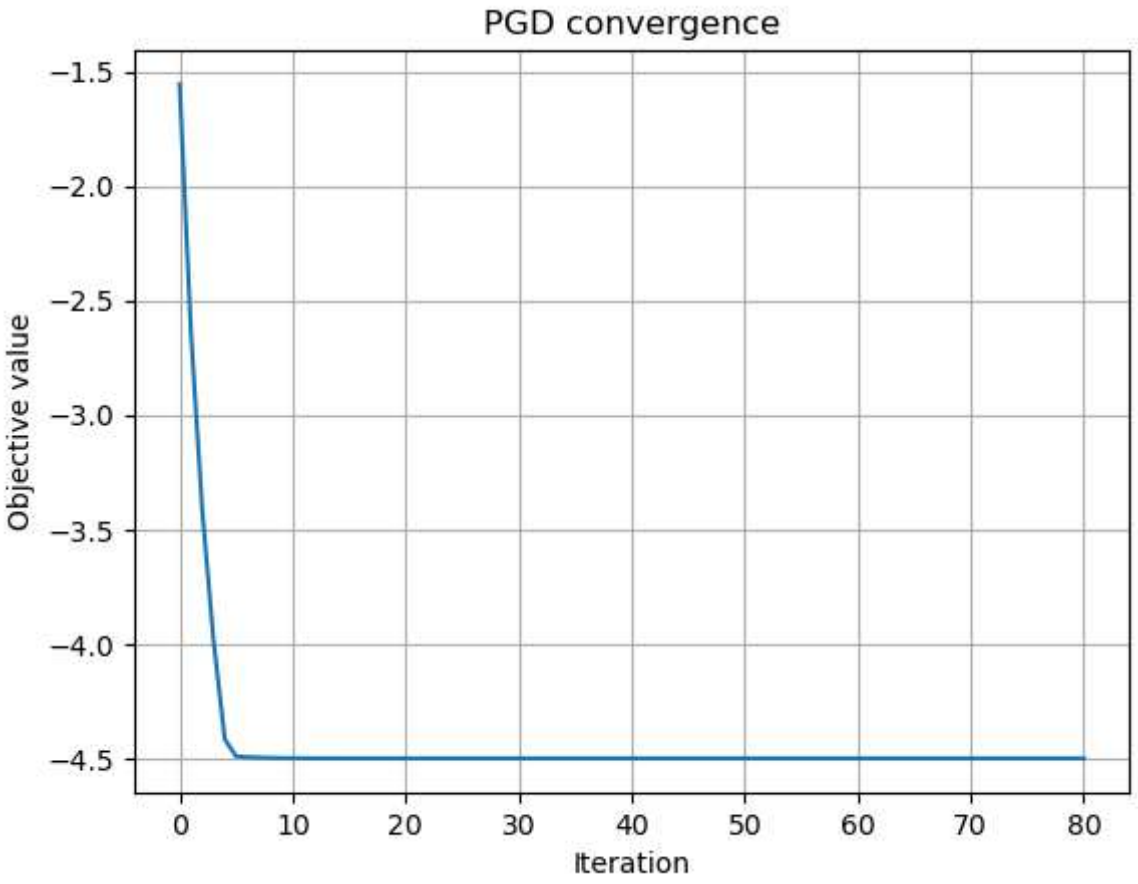
In [56]:
```
# Box bounds: -1 <= x_i <= 1
lower = -1.0
upper =  1.0

def project_box(x):
    return np.clip(x, lower, upper)
```

In [57]:
```
# Projected Gradient Descent
def projected_gradient_descent(x0, alpha=0.05, max_iter=5000, tol=1e-9):
    x = project_box(np.array(x0, dtype=float))
    objs = []
    for k in range(max_iter):
        g = gradient(x)
        x_new = x - alpha * g
        x_new = project_box(x_new)
        objs.append(objective(x_new))
        if np.linalg.norm(x_new - x) < tol:
            x = x_new
            break
        x = x_new
    return x, objective(x), objs
```

In [58]:
```
# Run projected gradient descent
x0 = np.array([0.0, 0.0])
x_pgd, f_pgd, objs = projected_gradient_descent(x0, alpha=0.05, max_iter=5000)
```

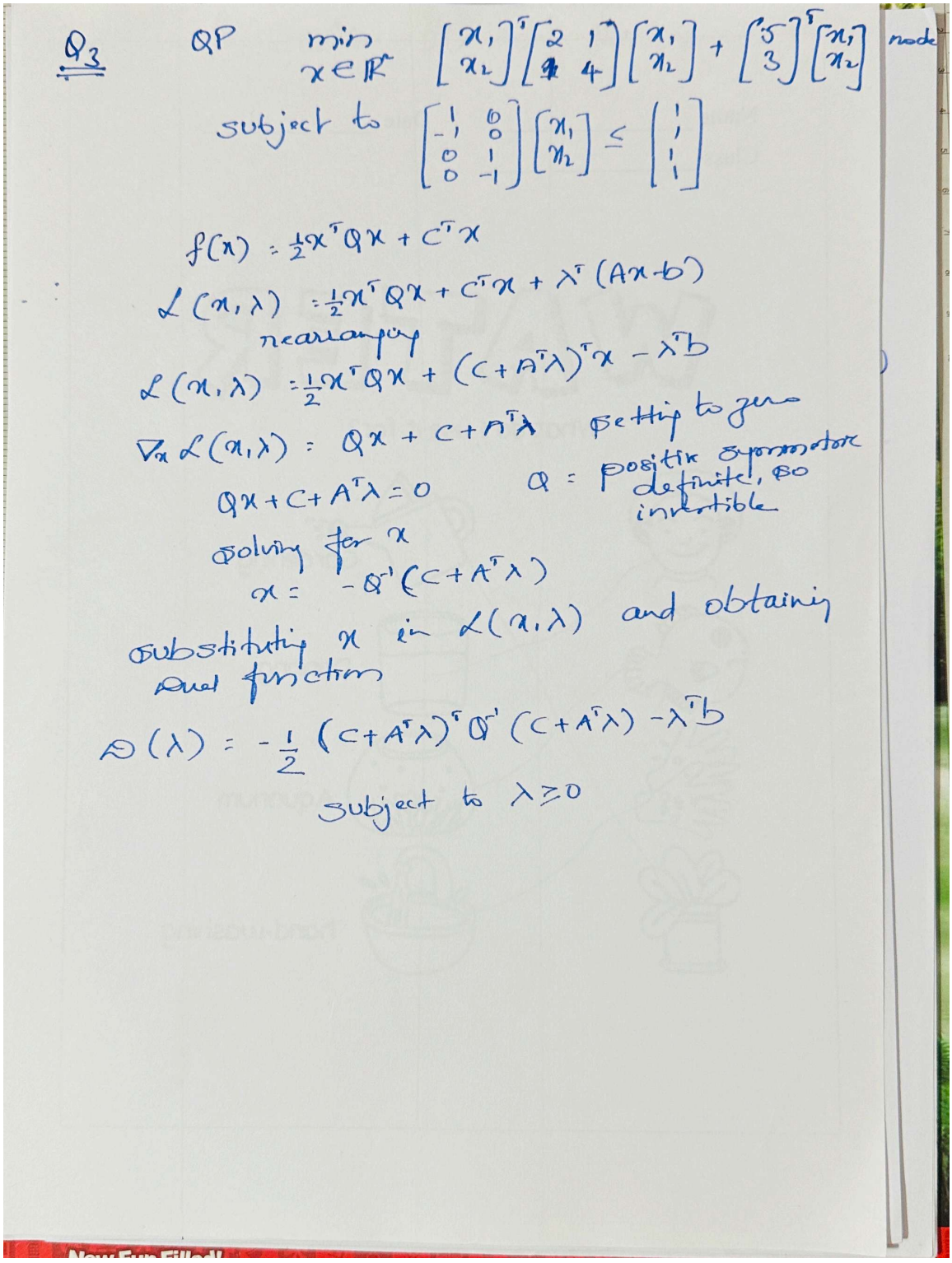
```python
print("projected gradient descent result:")
print(" x_pgd =", x_pgd)
print(" f(x_pgd) =", f_pgd)
print(" PGD iterations =", len(objs))

# Optional: quick plot of PGD convergence if matplotlib is available
try:
    import matplotlib.pyplot as plt
    plt.plot(objs)
    plt.xlabel("Iteration")
    plt.ylabel("Objective value")
    plt.title("PGD convergence")
    plt.grid(True)
    plt.show()
except Exception:
    pass
```

```
projected gradient descent result:
 x_pgd = [-1.  -0.5]
 f(x_pgd) = -4.5
 PGD iterations = 81
```



Langrangian dual problem

python code for dual problem



In [63]:
```
Qinv = np.linalg.inv(Q)
n = c.size
ones = np.ones(n)
```

In [64]:
```
def dual_grad(z):
    # z is length 2n: [lam (n), mu (n)]
    lam = z[:n]
    mu  = z[n:]
    v = c + lam - mu             # shape (n,)
```

```
    Qinv_v = Qinv @ v
    # gradient of g wrt lam and mu
    grad_lam = - Qinv_v - ones
    grad_mu  =   Qinv_v - ones
    return np.hstack([grad_lam, grad_mu])  # shape (2n,)
```

In [65]:
```
def dual_obj(z):
    lam = z[:n]
    mu  = z[n:]
    v = c + lam - mu
    term1 = -0.5 * v @ (Qinv @ v)
    term2 = - ones @ (lam + mu)
    return term1 + term2
```

In [66]:
```
# Projected gradient ascent
def solve_dual_pga(lr=0.1, max_iter=5000, tol=1e-8, verbose=False):
    z = np.zeros(2*n)     # initial lambda=mu=0
    obj_hist = []
    for k in range(max_iter):
        g = dual_grad(z)
        z_new = z + lr * g          # ascent step
        z_new = np.maximum(z_new, 0)  # projection onto nonnegative orthant
        obj = dual_obj(z_new)
        obj_hist.append(obj)
        if np.linalg.norm(z_new - z) < tol:
            z = z_new
            break
        z = z_new
    return z, obj_hist
```

In [67]:
```
z_opt, objs = solve_dual_pga(lr=0.05, max_iter=20000)
lam_opt = z_opt[:n]
mu_opt  = z_opt[n:]
dual_value = dual_obj(z_opt)

# Recover primal x from stationarity
x_from_dual = - Qinv @ (c + lam_opt - mu_opt)
primal_obj_from_dual = 0.5 * x_from_dual @ (Q @ x_from_dual) + c @ x_from_dual

print("Dual value (PGA)        :", dual_value)
print("lambda_opt              :", lam_opt)
print("mu_opt                  :", mu_opt)
print("Recovered primal x      :", x_from_dual)
print("Primal objective (from x):", primal_obj_from_dual)
```

```
Dual value (PGA)        : -4.500000000000032
lambda_opt              : [0. 0.]
mu_opt                  : [2.49999967 0.        ]
Recovered primal x      : [-1.00000019 -0.49999995]
Primal objective (from x): -4.500000477725507
```

The results of primal and dual problem are almost similar and have a smaller ignorable tolerance. Hence the duality is strong

In [ ]: