Q1. PCA for Breast Cancer Diagnosis (Real Data) (4×1 = 4 marks)

🧠 Context

The Wisconsin Breast Cancer dataset contains 30 features per biopsy.

🎯 Goal

Use Principal Component Analysis (PCA) to denoise the data and visualize class separability.

📂 Data Source

Use either of the following:

- `sklearn.datasets.load_breast_cancer()`
- Kaggle copy: Breast Cancer Wisconsin Dataset

In [1]:
```
import numpy as np
import pandas as pd
from sklearn import datasets
data = datasets.load_breast_cancer()
```

In [2]:
```
data.keys()
```

Out[2]:
```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'f
ilename', 'data_module'])
```

In [3]:
```
data.data.shape, data.target.shape
```

Out[3]:
```
((569, 30), (569,))
```

In [4]:
```
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target']=data.target
print(df.shape)
print(df.head())
```

```
(569, 31)
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812
2           0.15990          0.1974              0.12790         0.2069
3           0.28390          0.2414              0.10520         0.2597
4           0.13280          0.1980              0.10430         0.1809

   mean fractal dimension  ...  worst texture  worst perimeter  worst area  \
0                 0.07871  ...          17.33           184.60      2019.0
1                 0.05667  ...          23.41           158.80      1956.0
2                 0.05999  ...          25.53           152.50      1709.0
3                 0.09744  ...          26.50            98.87       567.7
4                 0.05883  ...          16.67           152.20      1575.0

   worst smoothness  worst compactness  worst concavity  worst concave points  \
0            0.1622             0.6656           0.7119                0.2654
1            0.1238             0.1866           0.2416                0.1860
2            0.1444             0.4245           0.4504                0.2430
3            0.2098             0.8663           0.6869                0.2575
4            0.1374             0.2050           0.4000                0.1625

   worst symmetry  worst fractal dimension  target
0          0.4601                  0.11890       0
1          0.2750                  0.08902       0
2          0.3613                  0.08758       0
3          0.6638                  0.17300       0
4          0.2364                  0.07678       0

[5 rows x 31 columns]
```

🔍 Tasks

a) Standardize & Covariance

- Standardize all 30 features to have zero mean and unit variance.
- Compute the 30 × 30 sample covariance matrix (S).
- Report the Frobenius norm (|S|_F) as a numeric value.

In [5]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_std = scaler.fit_transform(data.data)
df2 = pd.DataFrame(X_std, columns=data.feature_names)
pd.concat([df2.mean(),df2.std()],axis=1, keys=['mean','std'])
```

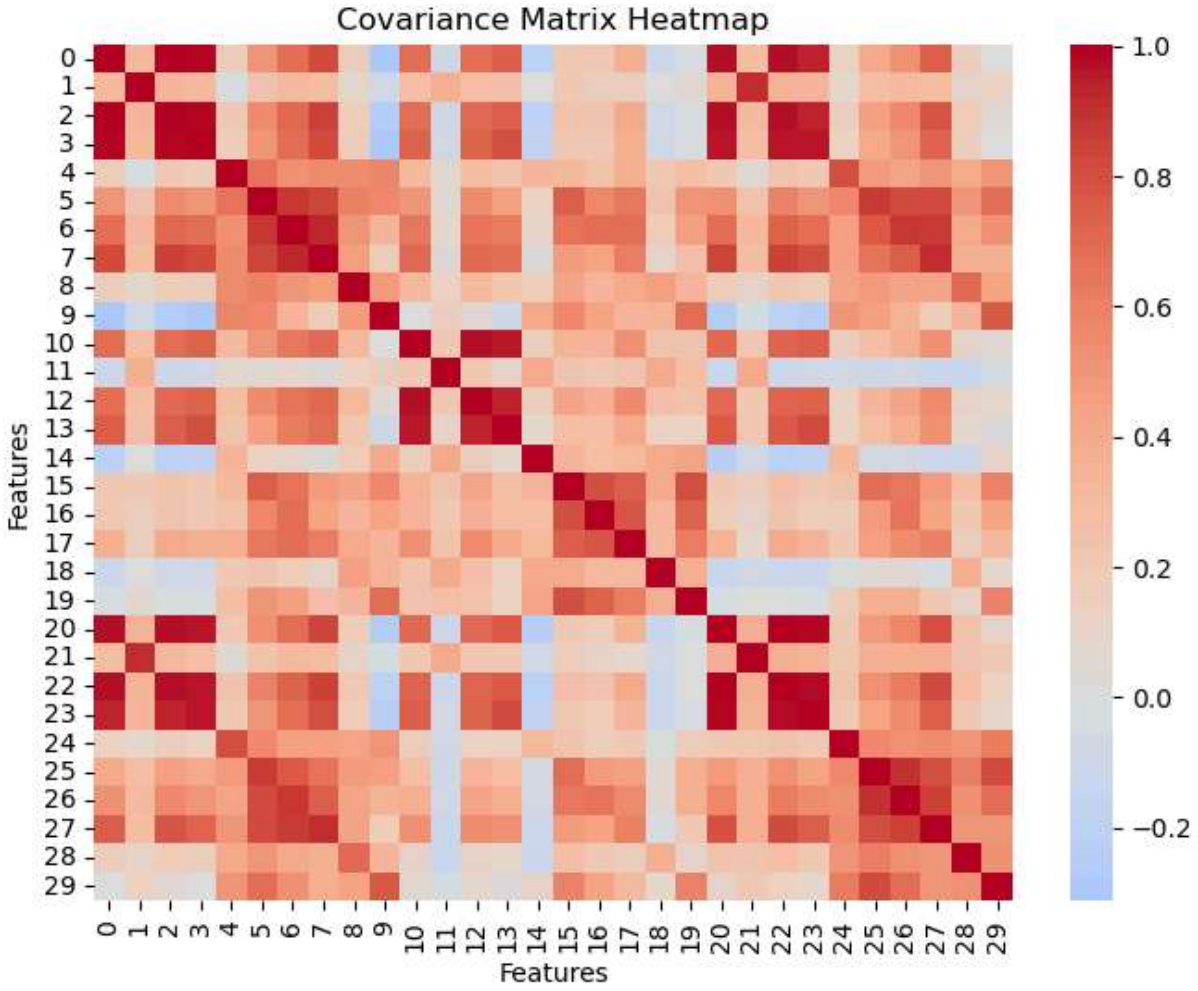
Out[5]:

	mean	std
mean radius	-3.162867e-15	1.00088
mean texture	-6.530609e-15	1.00088
mean perimeter	-7.078891e-16	1.00088
mean area	-8.799835e-16	1.00088
mean smoothness	6.132177e-15	1.00088
mean compactness	-1.120369e-15	1.00088
mean concavity	-4.421380e-16	1.00088
mean concave points	9.732500e-16	1.00088
mean symmetry	-1.971670e-15	1.00088
mean fractal dimension	-1.453631e-15	1.00088
radius error	-9.076415e-16	1.00088
texture error	-8.853492e-16	1.00088
perimeter error	1.773674e-15	1.00088
area error	-8.291551e-16	1.00088
smoothness error	-7.541809e-16	1.00088
compactness error	-3.921877e-16	1.00088
concavity error	7.917900e-16	1.00088
concave points error	-2.739461e-16	1.00088
symmetry error	-3.108234e-16	1.00088
fractal dimension error	-3.366766e-16	1.00088
worst radius	-2.333224e-15	1.00088
worst texture	1.763674e-15	1.00088
worst perimeter	-1.198026e-15	1.00088
worst area	5.049661e-16	1.00088
worst smoothness	-5.213170e-15	1.00088
worst compactness	-2.174788e-15	1.00088
worst concavity	6.856456e-16	1.00088
worst concave points	-1.412656e-16	1.00088
worst symmetry	-2.289567e-15	1.00088
worst fractal dimension	2.575171e-15	1.00088

In [6]:
```
df2.mean().mean(), df2.std().mean()
```

Out[6]:
```
(np.float64(-6.369603796435407e-16), np.float64(1.000879894582902))
```

In [7]:
```
# covariance matrix
S = np.cov(df2, rowvar=False)
import matplotlib.pyplot as plt
import seaborn as sns

# printing heat map of covariance matrix
plt.figure(figsize=(8, 6))
sns.heatmap(S, annot=False, cmap='coolwarm', center=0)
plt.title("Covariance Matrix Heatmap")
plt.xlabel("Features")
plt.ylabel("Features")
plt.show()
```



In [8]:
```
# Frobenius norm
fro_norm = np.linalg.norm(S, 'fro')
print("Frobenius norm of covariance matrix:", fro_norm)
```

```
Frobenius norm of covariance matrix: 15.062350986709806
```

b) Top-k Eigenpairs

- Set (k = 2).
- Use `sklearn.decomposition.PCA` to compute:
  - (i) Top two eigenvalues (\lambda_1, \lambda_2)
  - (ii) Explained variance ratio (%) of each
  - (iii) Cumulative explained variance (%)

In [9]:
```
# Eigvne value calculation as a reference, eigen values are printed in descending o
eigenvalues, eigenvectors = np.linalg.eig(S)
np.set_printoptions(precision=4, suppress=True)
idx = np.argsort(eigenvalues)[::-1]
print(eigenvalues[idx])
```

```
[13.305   5.7014  2.8229  1.9841  1.6516  1.2095  0.6764  0.4775  0.4176
  0.3513  0.2944  0.2616  0.2418  0.1573  0.0943  0.08    0.0595  0.0527
  0.0496  0.0312  0.03    0.0275  0.0244  0.0181  0.0155  0.0082  0.0069
  0.0016  0.0008  0.0001]
```

In [10]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
#df2 is the normalized breast cancer data (mean=0, sd = 1)
pca.fit(df2)
eigenvalues = pca.explained_variance_
print("Top 2 eigen values: ", eigenvalues)
explained_variance_ratio = pca.explained_variance_ratio_ * 100
print("Explained variance ratio: ", explained_variance_ratio)
cumulative_variance = explained_variance_ratio.cumsum()
print("Cumulative variance ratio: ", cumulative_variance)
```

```
Top 2 eigen values:  [13.305   5.7014]
Explained variance ratio:  [44.272  18.9712]
Cumulative variance ratio:  [44.272  63.2432]
```

First 2 eigen values are covering up to ~82% of variance in the data

c) 2D Projection

- Project all samples onto the top-2 principal components.
- Plot a labeled 2D scatter plot (malignant vs. benign).
- Report the class means in this 2D space.

In [11]:
```
# projecting samples onto top-2 principal components
X_pca = pca.fit_transform(df2)
df_pca = pd.DataFrame(X_pca, columns=['PC1','PC2'])
df_pca['target']= data.target
print(df_pca.head())
```

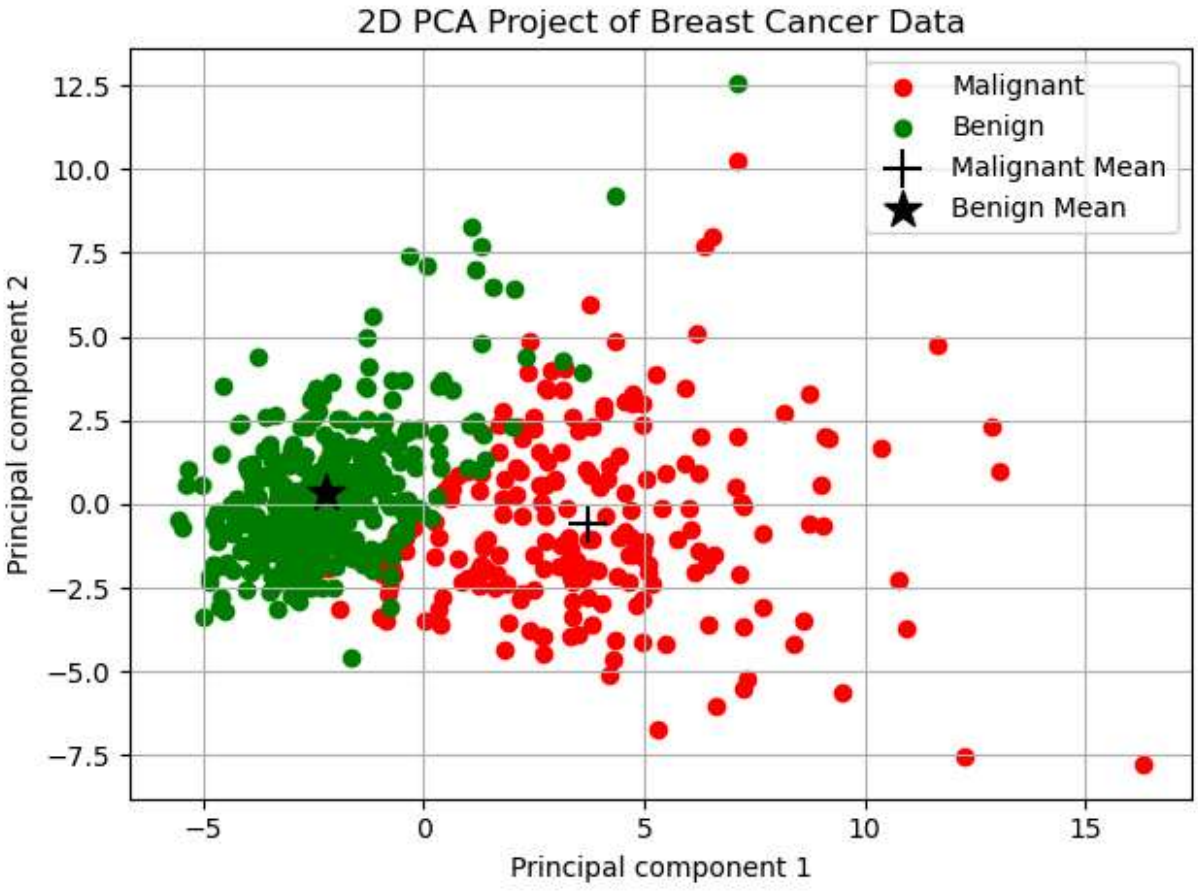
```
        PC1        PC2  target
0  9.192837   1.948583       0
1  2.387802  -3.768172       0
2  5.733896  -1.075174       0
3  7.122953  10.275589       0
4  3.935302  -1.948072       0
```

In [12]:
```
X_pca.shape
```

Out[12]: (569, 2)

In [13]:
```
malignant_features = df_pca[df_pca['target']==0][['PC1','PC2']]
benign_features = df_pca[df_pca['target']==1][['PC1','PC2']]
malignant_mean = malignant_features.mean().values
benign_mean = benign_features.mean().values
print("Malignant mean (PC1, PC2):", malignant_mean)
print("Benign mean (PC1, PC2):", benign_mean)

plt.scatter(malignant_features['PC1'], malignant_features['PC2'], c='red', label='M
plt.scatter(benign_features['PC1'], benign_features['PC2'], c='green', label='Benig

plt.scatter(*malignant_mean, c='k', marker='+', s=200, label='Malignant Mean')
plt.scatter(*benign_mean, c='k', marker='*', s=200, label='Benign Mean')
plt.xlabel('Principal component 1')
plt.ylabel('Principal component 2')
plt.title('2D PCA Project of Breast Cancer Data')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Malignant mean (PC1, PC2): [ 3.7148 -0.5831]
Benign mean (PC1, PC2): [-2.206   0.3463]
```



d) Reconstruction Error

- Inverse-transform the 2D projection back to the original 30D space.
- Report the average per-sample reconstruction Mean Squared Error (MSE) as a numeric value.

In [14]:
```
X_recon = pca.inverse_transform(X_pca)
X_recon.shape
```

Out[14]:
```
(569, 30)
```

In [15]:
```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(X_std, X_recon)
print(f"Average per-sample reconstruction MSE: {mse:.4f}")
```

```
Average per-sample reconstruction MSE: 0.3676
```

==========

Q2. PCA with Missing Values (Synthetic Spectroscopy)

🧪 Context

From an atomic spectroscopy experiment, five observables (X_1, \dots, X_5) are generated by two latent factors (t_1, t_2). Some entries in (X_5) are missing due to a spectrometer error. You are given an Excel file with 400 observations named `synthetic-pca-dataset`.

🎯 Goal

Handle missing values and compare **covariance vs. correlation PCA.**

✅ Tasks

a) Imputation

- Fill missing values in (X_5) using **mean imputation** (column mean).

In [16]:
```
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
```

In [17]:
```
spectro_df = pd.read_excel("synthetic_pca_dataset.xlsx")
spectro_df.isnull().sum()
```

Out[17]:
```
X1     0
X2     0
X3     0
X4     0
X5    34
dtype: int64
```

In [18]:
```
# mean imputation
spectro_df['X5'] = spectro_df['X5'].fillna(spectro_df['X5'].mean())
spectro_df.isnull().sum()
```

Out[18]:
```
X1    0
X2    0
X3    0
X4    0
X5    0
dtype: int64
```

b) Two PCAs

- Run:
  - **(i)** Covariance PCA on raw (unstandardized) data
  - **(ii)** Correlation PCA on standardized data

c) Spectrum

- For both (i) and (ii), report:
  - Eigenvalues
  - Explained variance ratios

d) PC Scatter

- Transform data to the first two principal components.
- Plot the **2D projection**.
- Comment briefly on **structure/cluster patterns.**

e) Reconstruction (2 PCs)

- Reconstruct the datasets using only the first two PCs.
- Report **average reconstruction MSE** for both (i) and (ii).

f) Project New Data

- Show how to project a new sample `x_new` to PC space (both cases).
- Include:
  - Code snippet
  - Resulting **2D PC coordinates**

In [19]:
```
#PCA on raw data
pca_cov = PCA(n_components=2)
# PCA on raw data (without scaling)
```

```
pca_cov.fit(spectro_df)
eigenvalues = pca_cov.explained_variance_
explained_variance_ratio = pca_cov.explained_variance_ratio_ * 100
cumulative_variance = explained_variance_ratio.cumsum()

spectro_pca_cov = pca_cov.fit_transform(spectro_df)
spectro_pca_cov.shape
```

Out[19]: (400, 2)

In [20]:
```
#PCA on normalized data
scaler = StandardScaler()
spectro_scaled = scaler.fit_transform(spectro_df)
pca_corr = PCA(n_components=2)
pca_corr.fit(spectro_scaled)
eigenvalues_corr = pca_corr.explained_variance_
explained_variance_ratio_corr = pca_corr.explained_variance_ratio_ * 100
cumulative_variance_corr = explained_variance_ratio_corr.cumsum()
```

In [21]:
```
#MSE calculation
from sklearn.metrics import mean_squared_error
spectro_cov_recon = pca_cov.inverse_transform(spectro_pca_cov)
mse_cov = mean_squared_error(spectro_df, spectro_cov_recon)

spectro_pca_corr = pca_corr.fit_transform(spectro_scaled)
spectro_corr_recon = pca_corr.inverse_transform(spectro_pca_corr)
mse_corr = mean_squared_error(spectro_scaled, spectro_corr_recon)

print(f" PCA on Raw / Unstandardized data \t\t\t\t\t\t PCA on standardized data")
print(f" Top 2 eigen values (raw data): {eigenvalues} \t\t Top eigen values (standa
print(f" Explained variance ratio: {explained_variance_ratio} \t\t\t\t Explained va
print(f" Cumulative variance ratio: {cumulative_variance} \t\t\t\t Cumulative varia
print(f" Average per-sample reconstruction MSE (cov): {mse_cov:.4f} \t \t \t \t Ave
```

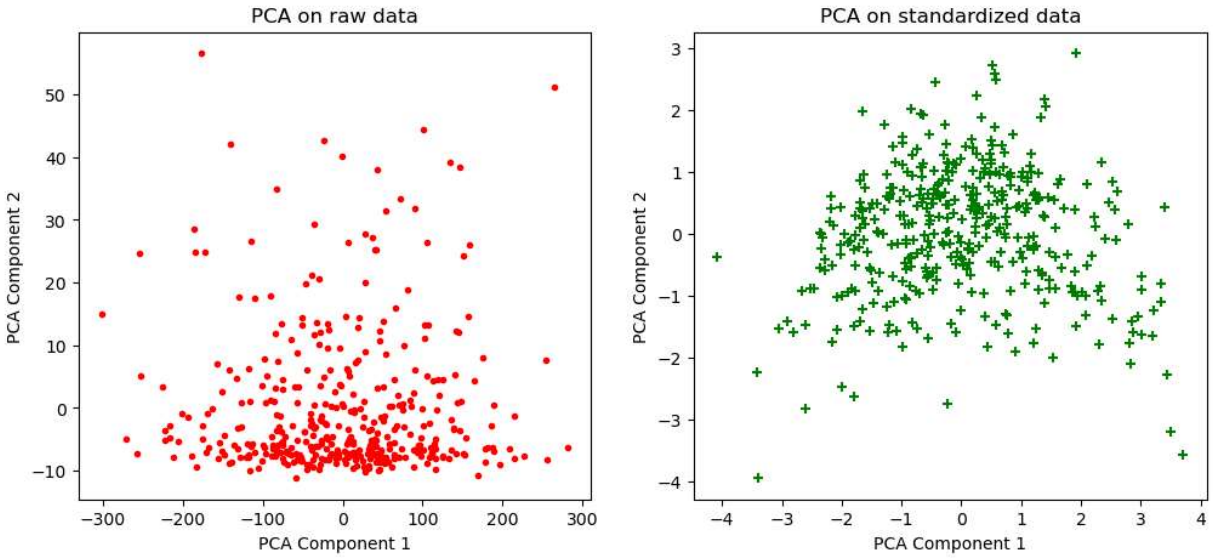
```
 PCA on Raw / Unstandardized data                                                PCA
on standardized data
 Top 2 eigen values (raw data): [10222.0768   130.4011]                 Top eigen v
alues (standardized data): [2.0071 1.0754]
 Explained variance ratio: [97.7211  1.2466]                            Explained v
ariance ratio: [40.042  21.4552]
 Cumulative variance ratio: [97.7211 98.9677]                           Cumulative
variance ratio: [40.042  61.4972]
 Average per-sample reconstruction MSE (cov): 21.5420                          Ave
rage per-sample reconstruction MSE (corr): 0.3850
```

Observation Top 2 Eigen values calculated from PCA on raw data have captured ~99% of total variance in the data resulting with a reconstruction MSE of ~21.5 Top 2 Eigen values calculated from PCA on standardized data have captured ~61.5% of total variance in the data yielding a lower scala reconstruction MSE of 0.385

In [22]:
```
#projections
fig, axes = plt.subplots(1, 2, figsize=(12, 5))  # 1 row, 2 columns
axes[0].scatter(spectro_pca_cov[:,0],spectro_pca_cov[:,1],c='red', marker='.')
axes[0].set_xlabel('PCA Component 1')
axes[0].set_ylabel('PCA Component 2')
```

```
axes[0].set_title('PCA on raw data')

axes[1].scatter(spectro_pca_corr[:,0],spectro_pca_corr[:,1],c='green', marker='+')
axes[1].set_xlabel('PCA Component 1')
axes[1].set_ylabel('PCA Component 2')
axes[1].set_title('PCA on standardized data')
```

Out[22]: Text(0.5, 1.0, 'PCA on standardized data')



Q3. Quadratic programming

python code for QP using Gradient Descent

In [23]:
```
import numpy as np
from scipy.optimize import minimize
```

In [24]:
```
# 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 [25]:
```
def objective(x):
    return 0.5 * x @ (Q @ x) + c @ x
```

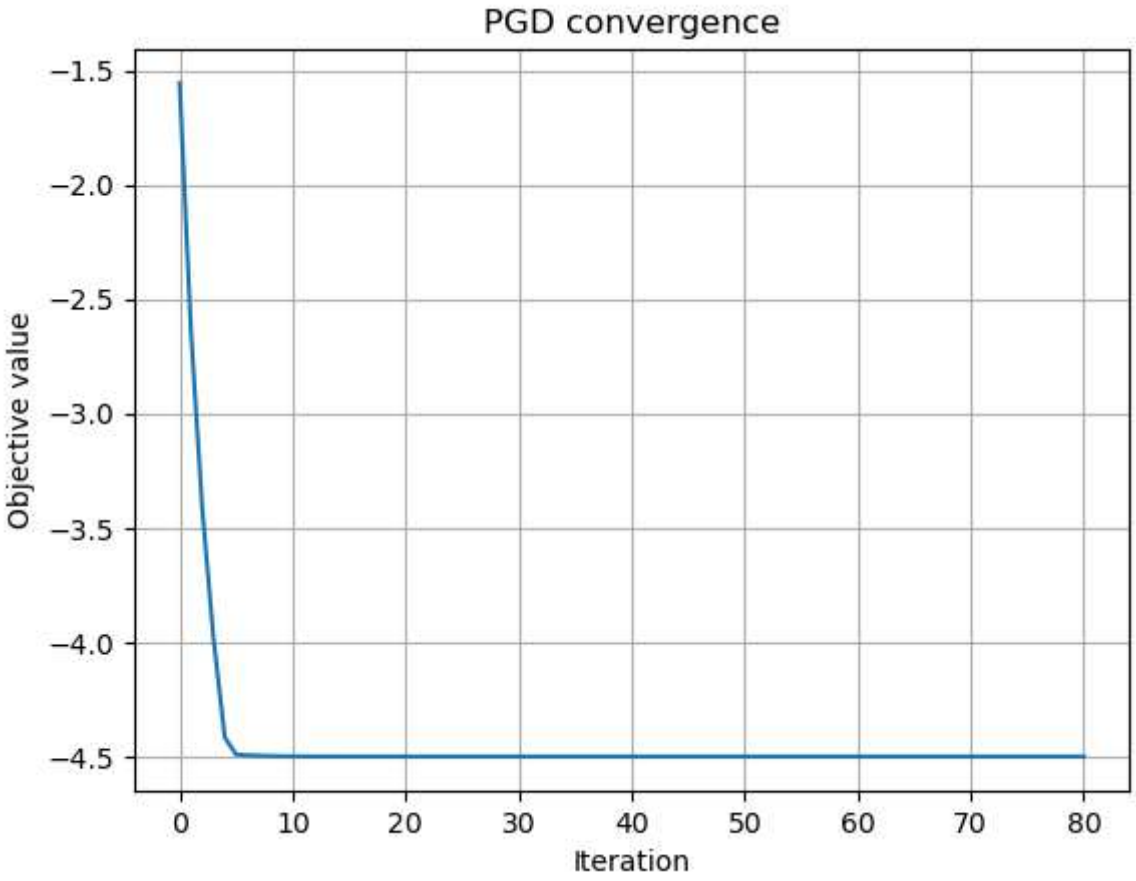
In [26]:
```
def gradient(x):
    return Q @ x + c
```

In [27]:
```python
# Box bounds: -1 <= x_i <= 1
lower = -1.0
upper =  1.0

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

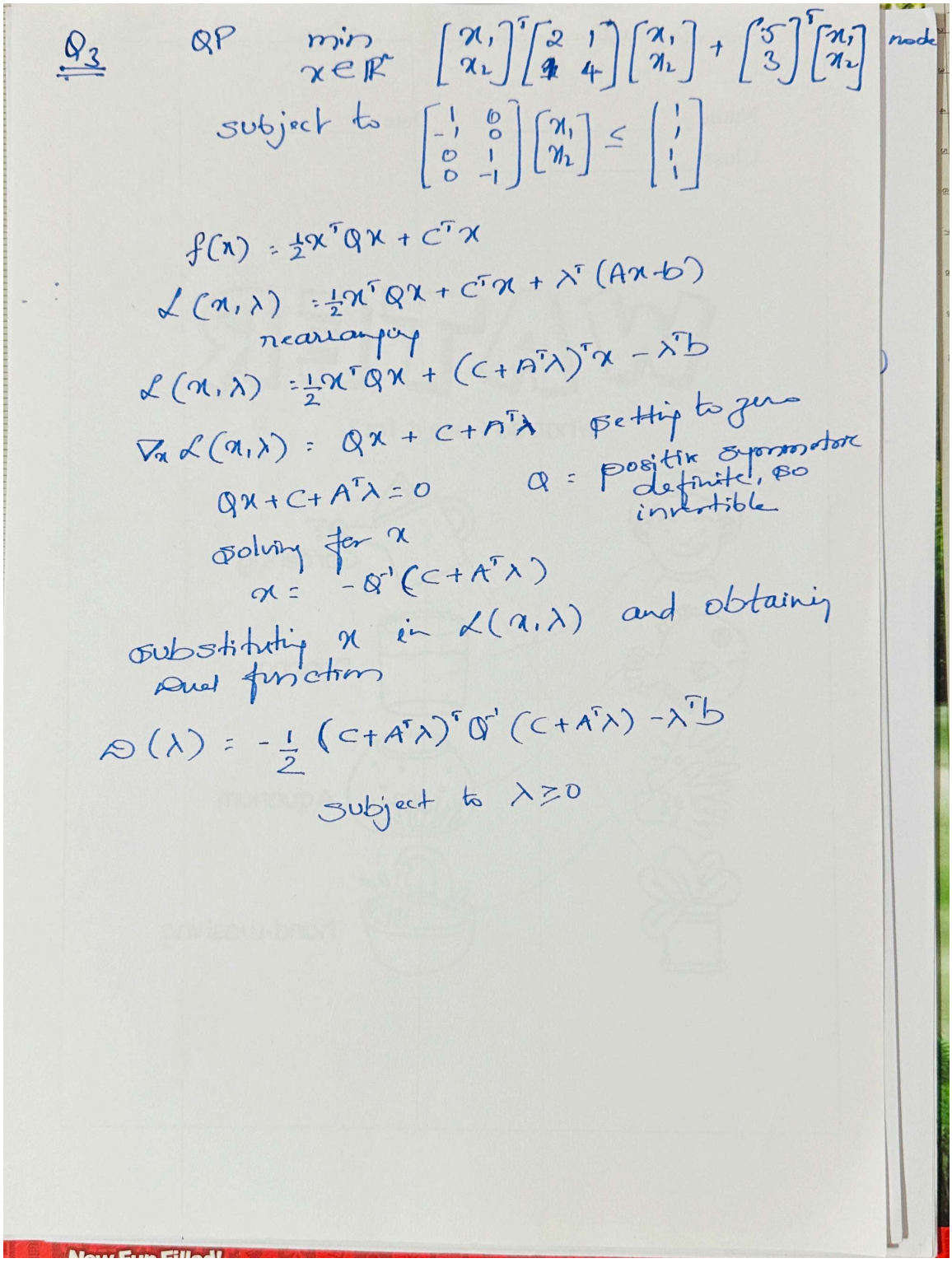
In [28]:
```python
# Projected Gradient Descent
def projected_gradient_descent(x0, alpha=0.05, max_iter=5000, tol=1e-9):
    x = project_box(np.array(x0, dtype=float))
    objs = []
    for k in range(max_iter):
        g = gradient(x)
        x_new = x - alpha * g
        x_new = project_box(x_new)
        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 [29]:
```python
# 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)
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
```



3b Langrangian dual problem



python code for dual problem

In [30]:
```
Qinv = np.linalg.inv(Q)
n = c.size
ones = np.ones(n)
```

In [31]:
```
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 [32]:
```
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 [33]:
```
# 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 [34]:
```
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                  : [2.5 0. ]
Recovered primal x      : [-1.  -0.5]
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