Mathematics for Machine Learning – Assignment Solutions

Q1) (1) Generate dataset $X \in R^{(50x6)}$

```
import numpy as np
import pandas as pd

# f1, f2, f3, f4 are generated from standard normal distribution
f1 = np.random.randn(50)
f2 = np.random.randn(50)
f3 = np.random.randn(50)
f4 = np.random.randn(50)

# since f5 and f6 are dependent features on f1, f2, f3, f4.
f5 = 2 * f1 + f2
f6 = f3 - f4

# Create a 2D Numpy array with size (50x6)
X = np.column_stack((f1, f2, f3, f4, f5, f6))

# Round all elements to 8 decimals
X = np.round(X, 8, out = X)

# Using DataFrame to view the constructed matrix (50x6)
dataFrame = pd.DataFrame(X, columns=["f1", "f2", "f3", "f4", "f5",
"f6"])

# Save the matrix to a csv file
dataFrame.to_csv("dataset_X.csv", index=False, float_format="%.8f")
```

Comment on round off errors:
Floating-point numbers are stored with limited precision. Some decimal values cannot be written exactly in binary, so the machine stores a very close value, not the exact one. So, the machine introduces round-off errors because it stores numbers in floating-point format.
When machine calculates the features f5 and f6 using features f1, f2, f3, f4, it may not hold the exact result, only something very close. After that, when we round everything to 8 decimals, numbers look neat, but it introduces another small difference. Because if we calculate f5 and f6 after this, then
f5 will not be equal to 2 * f1 + f2; From data shown let's take first row:

f1	**f2**	**f3**	**f4**	**f5**	**f6**
1.51473295	1.03897799	-0.19350891	-0.77230558	4.06844388	0.57879667

f5 = 2 * (1.51473295) + 1.03897799
f5 = 4.06844458 **not equal to** 4.06844388(actual value of f5)

Therefore, we can conclude that after rounding-off the features f5 and f6 are not anymore linear-combination of f1, f2, f3, f4. Hence, it affects the rank of matrix that was supposed to be 4 before round-off. Now it can compute the rank to be more than 4.

Q1) (2) Rank of X and Display the dataset generated above

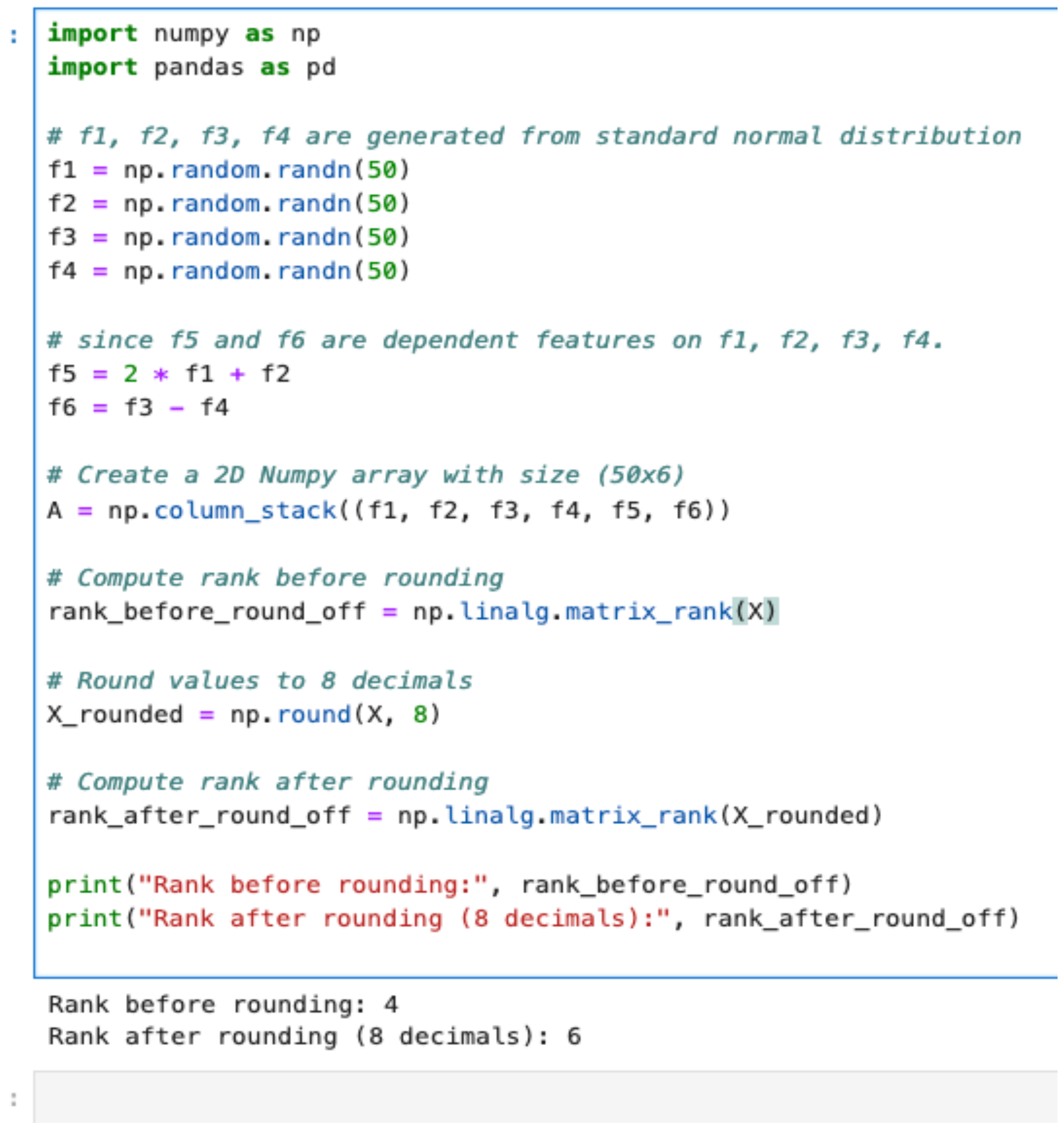
Matrix generated: -

f1	f2	f3	f4	f5	f6
1.51473295	1.03897799	-0.19350891	-0.77230558	4.06844388	0.57879667
0.55612370	-0.37658555	0.15895096	-0.07771266	0.73566186	0.23666361
1.87367610	0.71358571	-1.69390379	-0.44800883	4.46093791	-1.24589496
-0.61246131	-0.76743760	-0.87233761	-0.35944159	-1.99236022	-0.51289602
-0.31830765	-0.28419433	0.69143401	0.25075850	-0.92080964	0.44067550
1.82634663	-0.69490505	0.05882057	-0.45360821	2.95778822	0.51242878
-0.66267779	-1.39802916	0.28214850	-0.48506501	-2.72338473	0.76721351
-0.46912689	-0.62598233	1.40916763	-0.80978351	-1.56423611	2.21895114
0.08875751	0.03152032	0.31918064	-0.90281282	0.20903535	1.22199346
1.14049398	0.61550934	-0.20151043	-0.59783172	2.89649731	0.39632129
-0.46636538	1.81951833	0.57159963	-0.74916070	0.88678757	1.32076033
-1.30061475	0.32566171	-0.04373780	0.27531079	-2.27556779	-0.31904860
0.06855454	-0.50293399	0.28980232	1.20223722	-0.36582492	-0.91243490
0.83232728	0.64950437	0.20502086	-2.08673046	2.31415894	2.29175132
-0.34208647	1.17493397	0.96801232	1.87844527	0.49076103	-0.91043295
0.36490518	0.12085086	-1.29668145	-0.15239870	0.85066122	-1.14428275
-0.85765662	-1.43304473	0.26134458	0.05747247	-3.14835797	0.20387211
0.74765205	1.15993017	-0.36315585	-1.63510429	2.65523427	1.27194844
1.55081486	-0.23096670	-0.08319415	0.02804994	2.87066303	-0.11124409
1.59995425	-0.30119700	-0.51412009	1.01764198	2.89871150	-1.53176208
-0.06676010	1.08277473	0.23831879	-0.64908602	0.94925454	0.88740481
-0.73534138	-0.71293811	0.69977258	0.56372050	-2.18362087	0.13605209
-2.56553843	-1.62230694	-2.60748119	-0.63562123	-6.75338379	-1.97185995
2.13979775	-0.49468318	2.08398988	-0.87763914	3.78491232	2.96162901
-0.53594272	-0.09054304	-1.06609250	0.00075514	-1.16242847	-1.06684763
-1.17730364	-0.66704127	0.70785622	1.30025035	-3.02164855	-0.59239412
0.19746067	-1.88487149	0.37480753	0.60331900	-1.48995016	-0.22851147
0.12590063	0.36316333	0.27993583	-0.88844411	0.61496458	1.16837994
-0.61156124	0.18918903	-0.30602791	0.91646286	-1.03393346	-1.22249076

-0.28502094	-1.34524405	1.38284296	-1.34935152	-1.91528593	2.73219448
-0.69197237	0.49332920	-0.75813499	2.80104847	-0.89061555	-3.55918347
0.80950651	0.83759505	1.05477204	0.93786236	2.45660806	0.11690968
0.31048408	-0.85727109	-0.89759696	0.76705848	-0.23630294	-1.66465543
1.36349834	-0.74031840	-0.53434020	0.67638316	1.98667828	-1.21072335
-0.84179065	1.08321728	-0.73412102	0.82242479	-0.60036402	-1.55654581
2.37224062	0.26614713	0.71193089	0.48055228	5.01062838	0.23137861
0.20606076	-1.42987180	-0.63420665	-0.19584426	-1.01775028	-0.43836238
-0.44896032	-0.17780886	0.00133727	-0.85623735	-1.07572951	0.85757462
0.42976965	0.62025314	0.37802316	-1.39641669	1.47979243	1.77443985
0.80868043	1.66210537	-1.07227056	0.35509518	3.27946623	-1.42736573
0.86367769	0.20683421	1.40906276	0.81087469	1.93418959	0.59818807
0.60923140	-0.80715580	-0.48847807	0.06015808	0.41130701	-0.54863615
-0.07353674	-1.01763150	-1.20800380	1.15317991	-1.16470498	-2.36118371
-2.33144406	-1.31357305	0.60866272	-0.07454821	-5.97646117	0.68321093
-1.46499818	0.08979187	-0.53591484	-0.29088163	-2.84020448	-0.24503320
0.60851949	0.95971539	-0.31749436	0.69551343	2.17675437	-1.01300779
1.48998103	0.37825637	-0.06316210	1.81160110	3.35821843	-1.87476320
-0.34160011	-0.09281886	-0.85877341	-0.72718507	-0.77601909	-0.13158834
0.36528215	0.73837777	2.08475150	0.44407302	1.46894207	1.64067849
0.00143750	0.51746561	0.20346029	0.05043079	0.52034060	0.15302950

Python Code to compute Rank of X:

```python
import numpy as np
import pandas as pd

# f1, f2, f3, f4 are generated from standard normal distribution
f1 = np.random.randn(50)
f2 = np.random.randn(50)
f3 = np.random.randn(50)
f4 = np.random.randn(50)

# since f5 and f6 are dependent features on f1, f2, f3, f4.
f5 = 2 * f1 + f2
f6 = f3 - f4

# Create a 2D Numpy array with size (50x6)
A = np.column_stack((f1, f2, f3, f4, f5, f6))

# Compute rank before rounding
rank_before_round_off = np.linalg.matrix_rank(X)

# Round values to 8 decimals
X_rounded = np.round(X, 8)

# Compute rank after rounding
rank_after_round_off = np.linalg.matrix_rank(X_rounded)

print("Rank before rounding:", rank_before_round_off)
print("Rank after rounding (8 decimals):", rank_after_round_off)
```



```
import numpy as np
import pandas as pd

# f1, f2, f3, f4 are generated from standard normal distribution
f1 = np.random.randn(50)
f2 = np.random.randn(50)
f3 = np.random.randn(50)
f4 = np.random.randn(50)

# since f5 and f6 are dependent features on f1, f2, f3, f4.
f5 = 2 * f1 + f2
f6 = f3 - f4

# Create a 2D Numpy array with size (50x6)
A = np.column_stack((f1, f2, f3, f4, f5, f6))

# Compute rank before rounding
rank_before_round_off = np.linalg.matrix_rank(X)

# Round values to 8 decimals
X_rounded = np.round(X, 8)

# Compute rank after rounding
rank_after_round_off = np.linalg.matrix_rank(X_rounded)

print("Rank before rounding:", rank_before_round_off)
print("Rank after rounding (8 decimals):", rank_after_round_off)
```

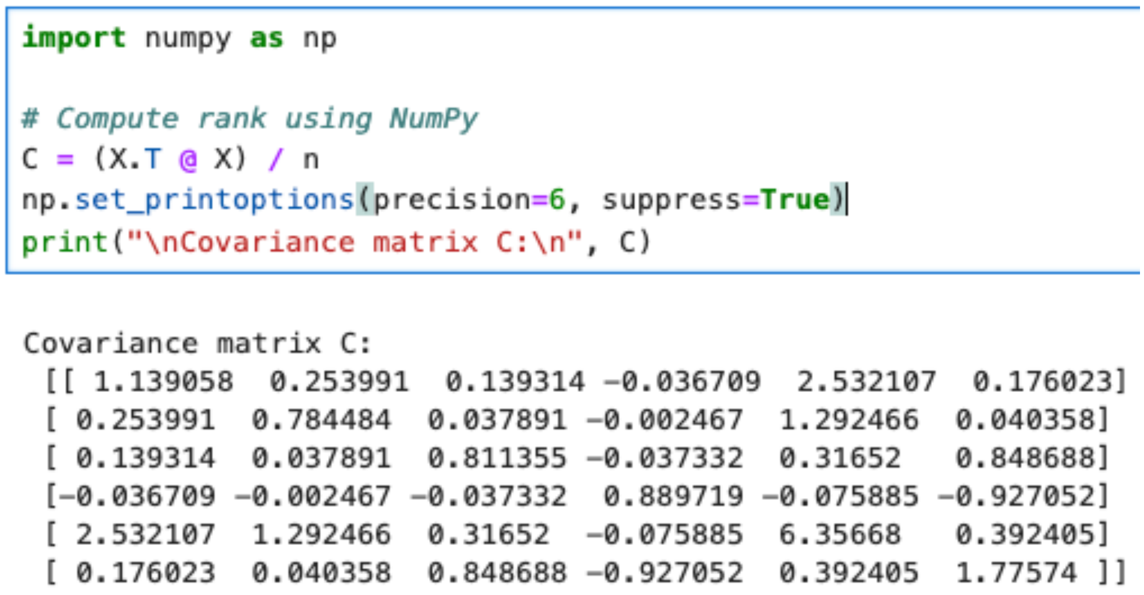
```
Rank before rounding: 4
Rank after rounding (8 decimals): 6
```

Because of round-off errors, the machine computed the rank as 6 instead of 4. This happens because after rounding, f5 and f6 are not exact linear combination of f1, f2, f3 and f4 anymore.

Q1) (3) Using Power Method to find dominant eigenvalue and it's corresponding eigenvector

a) Compute covariance matrix:

```python
import numpy as np

# Compute rank using NumPy
C = (X.T @ X) / n
np.set_printoptions(precision=6, suppress=True)
print("\nCovariance matrix C:\n", C)
```

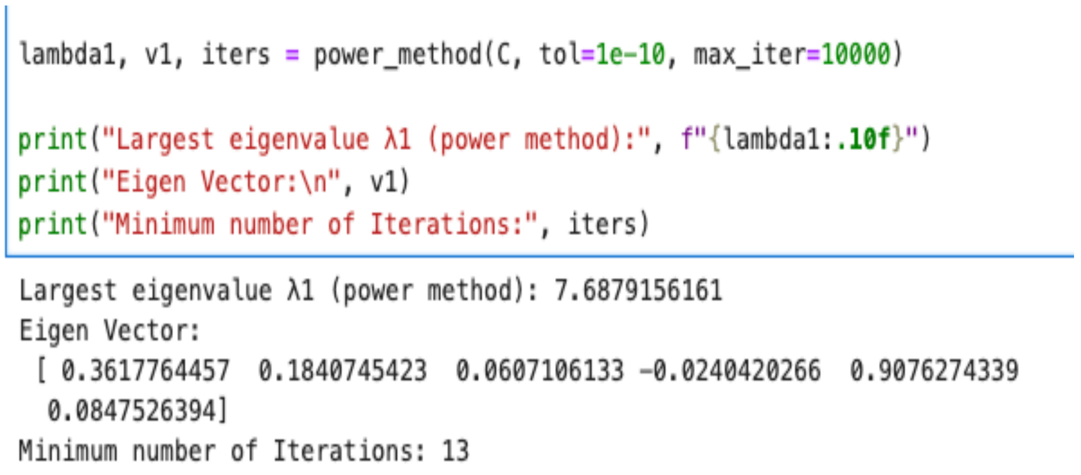


```python
import numpy as np

# Compute rank using NumPy
C = (X.T @ X) / n
np.set_printoptions(precision=6, suppress=True)
print("\nCovariance matrix C:\n", C)
```

```
Covariance matrix C:
 [[ 1.139058  0.253991  0.139314 -0.036709  2.532107  0.176023]
 [ 0.253991  0.784484  0.037891 -0.002467  1.292466  0.040358]
 [ 0.139314  0.037891  0.811355 -0.037332  0.31652   0.848688]
 [-0.036709 -0.002467 -0.037332  0.889719 -0.075885 -0.927052]
 [ 2.532107  1.292466  0.31652  -0.075885  6.35668   0.392405]
 [ 0.176023  0.040358  0.848688 -0.927052  0.392405  1.77574 ]]
```

b) Implement Power Method to approximate largest eigenvalue λ1 and its corresponding eigenvector v1 of C.

```
import numpy as np
from numpy.linalg import norm

# Function to estimate eigenvalue for a given vector
def rayleigh_quotient(X, v):
    v = v / (norm(v) + 1e-18) # add 1e-18 to avoid dividing by zero
    return float(v.T @ X @ v) # Return estimate of eigenvalue using formula
v^T X v

# Power Method to find largest eigenvalue and eigenvector
def power_method(X, tol=1e-10, max_iter=10000, x0=None):
    n = X.shape[0]
    x = np.random.randn(n) if x0 is None else np.array(x0, dtype=float)
    x = x / (norm(x) + 1e-18)

    lambda_old = 0.0
    for k in range(1, max_iter + 1):
        y = X @ x
        ny = norm(y)
        if ny == 0:
            # rare fallback: if X sends x to (near) zero, restart with a new random
x
            x = np.random.randn(n)
            x = x / (norm(x) + 1e-18)
            continue
        x = y / ny
        # New eigenvalue estimate using Rayleigh quotient
        lambda_new = rayleigh_quotient(X, x)
        # Stop if the new and old eigenvalues are very close (relative change ≤
tol)
        if abs(lambda_new - lambda_old) <= tol * (abs(lambda_new) + 1e-18):
            # Flip sign of vector so results are consistent
            if x[0] < 0:
                x = -x
            return lambda_new, x, k
        lambda_old = lambda_new
    # If we reach max iterations without meeting tolerance, return last result
    if x[0] < 0:
        x = -x
    return lambda_new, x, max_iter

lambda1, v1, iters = power_method(C, tol=1e-10, max_iter=10000)

print("Largest eigenvalue λ1 (power method):", f"{lambda1:.10f}")
print("Eigen Vector:\n", v1)
print("Minimum number of Iterations:", iters)
```



```
lambda1, v1, iters = power_method(C, tol=1e-10, max_iter=10000)

print("Largest eigenvalue λ1 (power method):", f"{lambda1:.10f}")
print("Eigen Vector:\n", v1)
print("Minimum number of Iterations:", iters)
```

```
Largest eigenvalue λ1 (power method): 7.6879156161
Eigen Vector:
 [ 0.3617764457  0.1840745423  0.0607106133 -0.0240420266  0.9076274339
  0.0847526394]
Minimum number of Iterations: 13
```

c) **Next largest eigenvalue λ_2 and its corresponding eigenvector V_2 by applying power method on $C - V_1 V_1^T C$.**

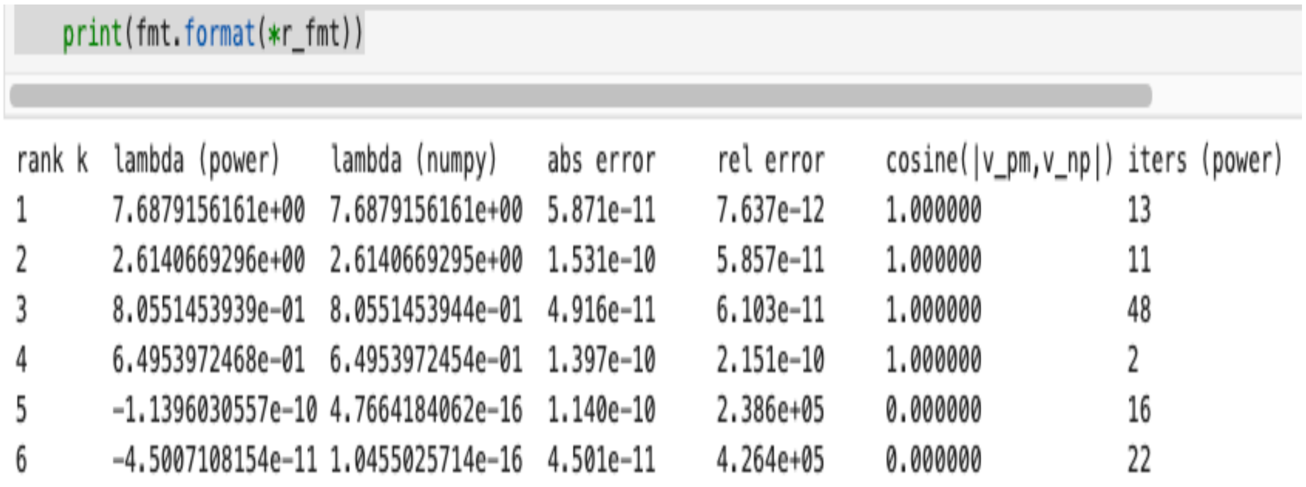
```
def top_k_power_deflation(X, k, tol=1e-10, max_iter=10000):
    """
    Returns top-k eigenpairs using:
      X <- X - λ v v^T
    after each dominant eigenpair is found.
    """
    X_work = X.copy()
    Eigvals, eigvecs, iters_list = [], [], []
    for j in range(k):
        lam, v, iters = power_method(X_work, tol=tol, max_iter=max_iter)
        eigvals.append(lam)
        eigvecs.append(v)
        iters_list.append(iters)
        # Remove the found mode so the next run finds the next eigenpair
        X_work = X_work - lam * np.outer(v, v)
    return np.array(eigvals), np.column_stack(eigvecs), np.array(iters_list)

# -----------------------------
# 5) Get λ1, v1; then λ2, v2 using prompt’s deflation C - v1 v1^T C
# -----------------------------
lambda1, v1, iters1 = power_method(C)

# Your prompt’s specific deflation: X2 = C - v1 v1^T C
X2_prompt = C - np.outer(v1, v1.T @ C)
lambda2_prompt, v2_prompt, iters2_prompt = power_method(X2_prompt)

# (Optional) Also get top-k using standard deflation
k = C.shape[0]
eigvals_defl, eigvecs_defl, iters_defl = top_k_power_deflation(C, k=k)

print("=== Matrix sizes ===")
print("X:", X.shape, "  C:", C.shape)

print("\n=== λ1, v1 from C (power method) ===")
print("λ1:", f"{lambda1:.10f}")
print("v1:", v1)
print("iterations:", iters1)

print("\n=== Next eigenpair using prompt's deflation X2 = C - v1 v1^T C ===")
print("λ2 (power on X2):", f"{lambda2_prompt:.10f}")
print("v2 (power on X2):", v2_prompt)
print("iterations:", iters2_prompt)

print("\n=== Top-k eigenpairs using standard deflation (X <- X - λ v v^T) ===")
for j in range(k):
    print(f"k={j+1}: λ={eigvals_defl[j]:.10f}, iters={iters_defl[j]}")
    print("  v:", eigvecs_defl[:, j])
print("\n=== Top-k eigenpairs using standard deflation (X <- X - λ v v^T) ===")
for j in range(k):
    print(f"k={j+1}: λ={eigvals_defl[j]:.10f}, iters={iters_defl[j]}")
    print("  v:", eigvecs_defl[:, j])
```

```
=== Matrix sizes ===
X: (50, 6)   C: (6, 6)

=== λ1, v1 from C (power method) ===
λ1: 7.6879156161
v1: [ 0.3617763455  0.1840743691  0.0607120992 -0.0240437831  0.9076270602
  0.0847558819]
iterations: 13

=== Next eigenpair using prompt's deflation X2 = C - v1 v1^T C ===
λ2 (power on X2): 2.6140669295
v2 (power on X2): [ 0.0250668387  0.0433152257 -0.371627917   0.4393267397  0.0934489026
 -0.8109546563]
iterations: 10

=== Top-k eigenpairs using standard deflation (X <- X - λ v v^T) ===
k=1: λ=7.6879156161, iters=14
  v: [ 0.3617763227  0.1840743297  0.0607124375 -0.024044183   0.9076269752
  0.0847566201]
k=2: λ=2.6140669296, iters=11
  v: [ 0.0250687973  0.0433175973 -0.3716250169  0.4393288912  0.0934551913
 -0.8109539077]
k=3: λ=0.8055145394, iters=53
  v: [ 0.0268996091 -0.0813170247  0.7209530532  0.686242367  -0.0275178085
  0.0347106853]
k=4: λ=0.6495397247, iters=2
  v: [ 0.4484408302 -0.8893594729 -0.0714576224 -0.0463278376  0.0075221887
 -0.0251297848]
k=5: λ=-0.0000000001, iters=12
  v: [ 0.3617800482  0.1840758414  0.0607158691 -0.0240483583  0.9076240619
  0.0847649886]
k=6: λ=-0.0000000000, iters=24
  v: [ 0.0250715738  0.0433204614 -0.3716215266  0.4393246848  0.0934645049
 -0.8109564738]
```

d) All eigenvalues and eigenvector using

```
def cosine_similarity(u, v):
    """|cos(angle)|; 1.0 means same direction up to sign."""
    u = u / (norm(u) + 1e-18)
    v = v / (norm(v) + 1e-18)
    return float(abs(u @ v))

# ---------- (1) Built-in eigenpairs with NumPy ----------
# eigh is for symmetric matrices like C; it returns ascending order.
eigvals_all, eigvecs_all = np.linalg.eigh(C)
idx = np.argsort(eigvals_all)[::-1]          # sort descending
eigvals_np = eigvals_all[idx]                # shape: (n,)
eigvecs_np = eigvecs_all[:, idx]             # shape: (n,n)

# ---------- (2) Power method + deflation (recomputed for comparison) -------
---
k = C.shape[0]
eigvals_pm, eigvecs_pm, iters_pm = top_k_power_deflation(C, k=k, tol=1e-
10, max_iter=10000)

# ---------- (3) Compare eigenvalues and eigenvectors ----------
rows = []
for j in range(k):
    lam_pm = eigvals_pm[j]
    lam_np = eigvals_np[j]
    vec_pm = eigvecs_pm[:, j]
    vec_np = eigvecs_np[:, j]

    abs_err = float(abs(lam_pm - lam_np))
    rel_err = float(abs_err / (abs(lam_np) + 1e-18))
    cos_sim = cosine_similarity(vec_pm, vec_np)

    rows.append([
        j+1, lam_pm, lam_np, abs_err, rel_err, cos_sim, int(iters_pm[j])
    ])

# Pretty print comparison table
```

```
header = ["rank k", "lambda (power)", "lambda (numpy)", "abs error", "rel
error", "cosine(|v_pm,v_np|)", "iters (power)"]
col_widths = [8, 18, 18, 14, 14, 20, 14]
fmt = "".join([f"{{:<{w}}}" for w in col_widths])
print(fmt.format(*header))
for r in rows:
    # format floats compactly
    r_fmt = [
        r[0],
        f"{r[1]:.10e}",
        f"{r[2]:.10e}",
        f"{r[3]:.3e}",
        f"{r[4]:.3e}",
        f"{r[5]:.6f}",
        r[6]
    ]
    print(fmt.format(*r_fmt))
```



```
    print(fmt.format(*r_fmt))
```

```
rank k  lambda (power)    lambda (numpy)    abs error     rel error     cosine(|v_pm,v_np|) iters (power)
1       7.6879156161e+00  7.6879156161e+00  5.871e-11     7.637e-12     1.000000            13
2       2.6140669296e+00  2.6140669295e+00  1.531e-10     5.857e-11     1.000000            11
3       8.0551453939e-01  8.0551453944e-01  4.916e-11     6.103e-11     1.000000            48
4       6.4953972468e-01  6.4953972454e-01  1.397e-10     2.151e-10     1.000000            2
5       -1.1396030557e-10 4.7664184062e-16  1.140e-10     2.386e+05     0.000000            16
6       -4.5007108154e-11 1.0455025714e-16  4.501e-11     4.264e+05     0.000000            22
```

Comparison:

- absolute error / relative error: this tells us how close the power-method eigenvalues are to NumPy's.
- cosine(|v_pm, v_np|): this tells us how well the eigenvectors line up.
- Iters(power): this tells us how many power-method steps each rank took.

e) All eigenvalues and eigenvector using

Rank 1 ($\lambda_1 \approx$ 7.6879156161, 13 iters):

- Next eigenvalue $\lambda_2 \approx 2.6140666930$
- Ratio $\lambda_2/\lambda_1 \approx 2.6141 / 7.6879 \approx 0.34$
- This means that each step the leftover error shrinks by around 34% and hence converged quickly in 13 iterations

Rank 2 ($\lambda_2 \approx$ 2.6140666930, 11 iters)

- After removing rank-1, the new competitor is $\lambda_3 \approx 0.8055145394$
- Ratio $\lambda_3/\lambda_2 \approx 0.8055 / 2.6141 \approx 0.31$
- This means that error shrinks a bit faster than rank-1 and hence converged in 11 iterations only

Rnak 3 ($\lambda_3 \approx$ 0.8055145394, 48 iters)

- Now the next competitor is $\lambda_4 \approx 0.6495397245$
- Ratio $\lambda_4/\lambda_3 \approx 0.6495 / 0.8055 \approx 0.81$
- This means the ratio is much closer to 1 than before, so the error decays slowly; a lot more steps are needed. Hence, it took 48 iterations.

Rank 4 ($\lambda_4 \approx$ 0.6495397245, 2 iters)

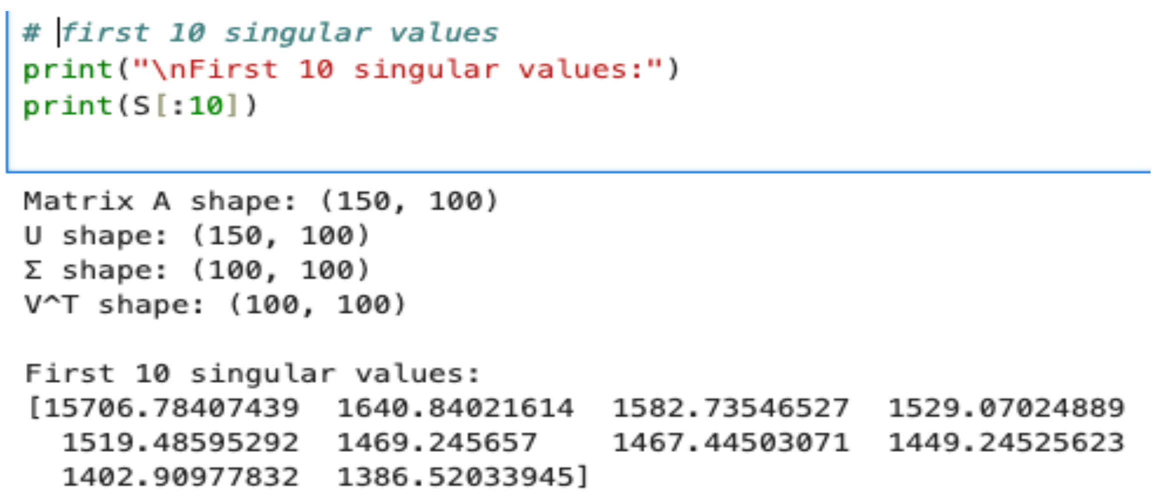
- Next competitor is $\lambda_5 \approx$ 4.77e-16 (essentially zero).
- Ratio $\lambda_5/\lambda_4 \approx \sim 0 / 0.6495 \approx 0$
- With almost nothing competing, the method locks in very quickly. Hence 2 iterations.

Rank 5 and 6 (near-zero eigenvalues; 16 and 22 iters)

- $\lambda_5 \approx$ 4.77e-16, $\lambda_6 \approx$ 1.05e-16 (basically zero).
- Here, floating-point noise and small rounding effects dominate.
- The table shows power estimates around (-1.14e-10 and -4.50e-11), huge relative errors ($\approx$ 2.386e+05 and 4.264e+05), and cosine = 0.0 (i.e., vectors don't line up and hence there is no stable direction to lock onto).
- Hence the 16 and 22 iterations don't carry any useful meaning and they are in numerical noise area.
-

Q2) (a) Compute SVD of A ∈ $R^{150 \times 100}$

```
import numpy as np

# Size: 150x100, values between 0-255 like pixel intensities
A = np.random.randint(0, 256, size=(150, 100)).astype(float)

print("Matrix A shape:", A.shape)

# Perform Singular Value Decomposition (SVD)
# full_matrices=False gives compact form (U: 150x100, Σ: 100x100, Vt:
100x100)
U, S, Vt = np.linalg.svd(A, full_matrices=False)

# Convert S (1D array of singular values) to diagonal matrix Σ
Sigma = np.diag(S)


print("U shape:", U.shape)       # (150, 100)
print("Σ shape:", Sigma.shape)   # (100, 100)
print("V^T shape:", Vt.shape)    # (100, 100)

# first 10 singular values
print("\nFirst 10 singular values:")
print(S[:10])
```



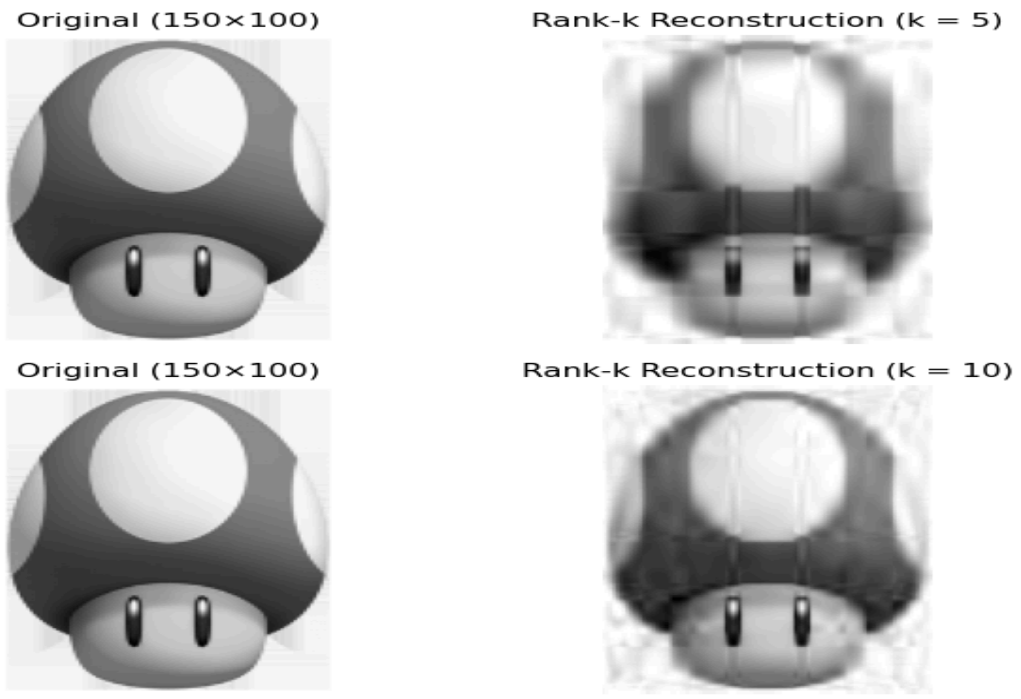
```
# first 10 singular values
print("\nFirst 10 singular values:")
print(S[:10])
```

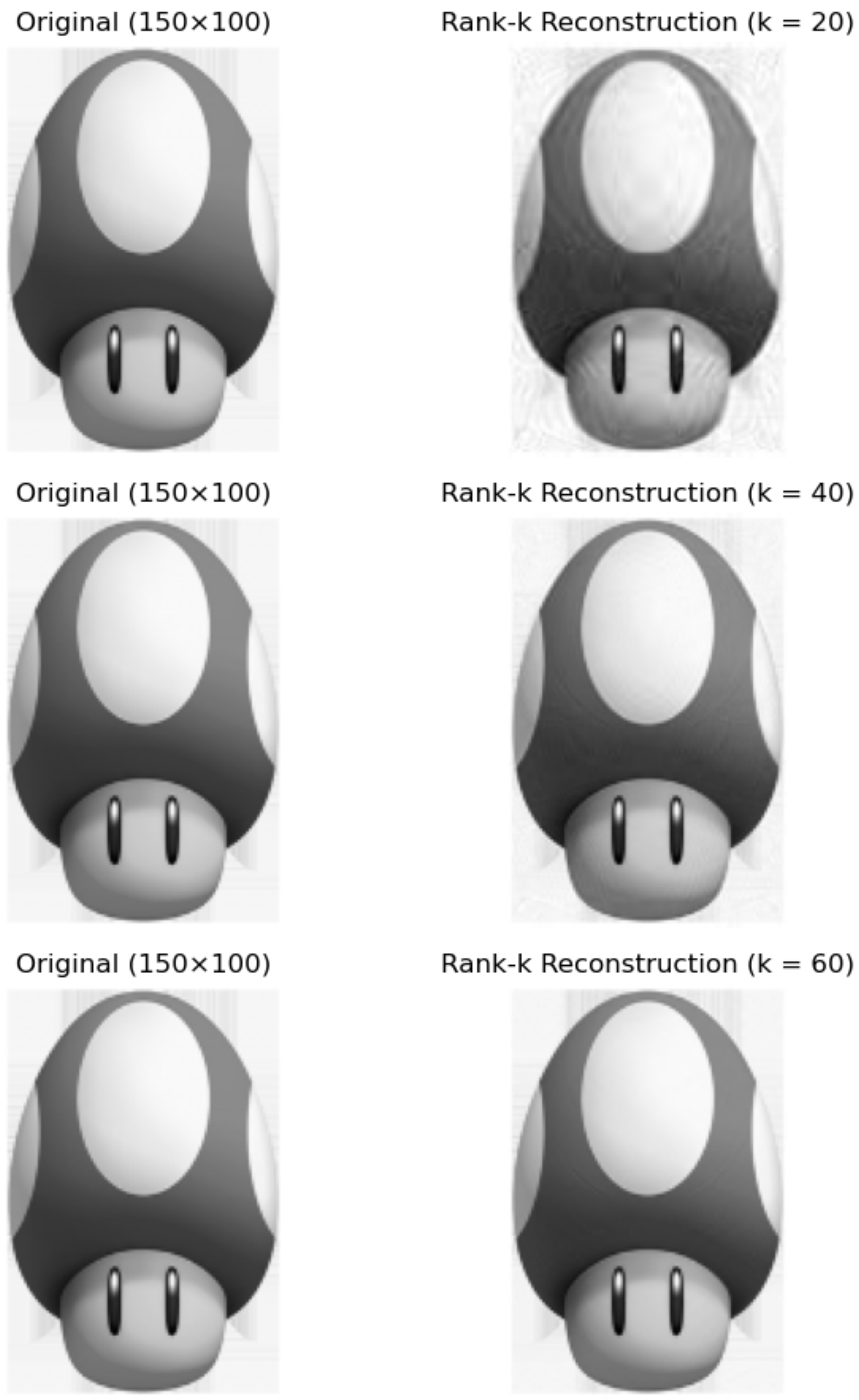
```
Matrix A shape: (150, 100)
U shape: (150, 100)
Σ shape: (100, 100)
V^T shape: (100, 100)

First 10 singular values:
[15706.78407439  1640.84021614  1582.73546527  1529.07024889
  1519.48595292  1469.245657    1467.44503071  1449.24525623
  1402.90977832  1386.52033945]
```

Q2) (b) Reconstruction with Multiple Ranks

```python
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, color
from skimage.transform import resize

# safe grayscale loader that handles 1/2/3/4 channels
def load_as_gray(img_path):
    """
    Loads an image as grayscale float64 in [0,1].
    Handles:
      - HxW (gray)
      - HxWx2 (L + Alpha)  -> use L
      - HxWx3 (RGB)        -> rgb2gray
      - HxWx4 (RGBA)       -> rgba2rgb -> rgb2gray
    """
    img = io.imread(img_path)

    if img.ndim == 2:
        # already grayscale
        A = img.astype(np.float64)
        if A.max() > 1.0:  # convert 0..255 to 0..1 if needed
            A = A / 255.0
        return A

    if img.ndim == 3:
        c = img.shape[2]
        if c == 3:
            # RGB -> gray
            A = color.rgb2gray(img)
            return A
        elif c == 4:
            # RGBA -> RGB -> gray
            rgb = color.rgba2rgb(img)
            A = color.rgb2gray(rgb)
            return A
        elif c == 2:
            # L + Alpha (or similar) -> use L channel
            A = img[..., 0].astype(np.float64)
            if A.max() > 1.0:
                A = A / 255.0
            return A
        else:
            raise ValueError(f"Unsupported number of channels: {c}")

    raise ValueError(f"Unsupported image shape: {img.shape}")

# Load image
img_path = "mush.png"
A = load_as_gray(img_path)  # float64 in [0,1]

# Resize to A ∈ R^{150×100}
A = resize(A, (150, 100), anti_aliasing=True)  # stays in [0,1]

# SVD (compact)
# A = U @ diag(S) @ Vt, with S sorted descending
U, S, Vt = np.linalg.svd(A, full_matrices=False)  # U: (150,100), S: (100,), Vt: (100,100)

# rank-k reconstruction
def reconstruct_rank_k(U, S, Vt, k):
    """
    Build A_k = U[:, :k] @ diag(S[:k]) @ Vt[:k, :]
    Returns float image in [0,1].
    """
    Uk = U[:, :k]
    Sk = np.diag(S[:k])
    Vtk = Vt[:k, :]
    Ak = Uk @ Sk @ Vtk
    return np.clip(Ak, 0.0, 1.0)

# Build rank-k images for requested ks
ks = [5, 10, 20, 40, 60]
reconstructions = [(k, reconstruct_rank_k(U, S, Vt, k)) for k in ks]

# Display: Original (left) + each rank-k (right)
n_rows = len(ks)
```

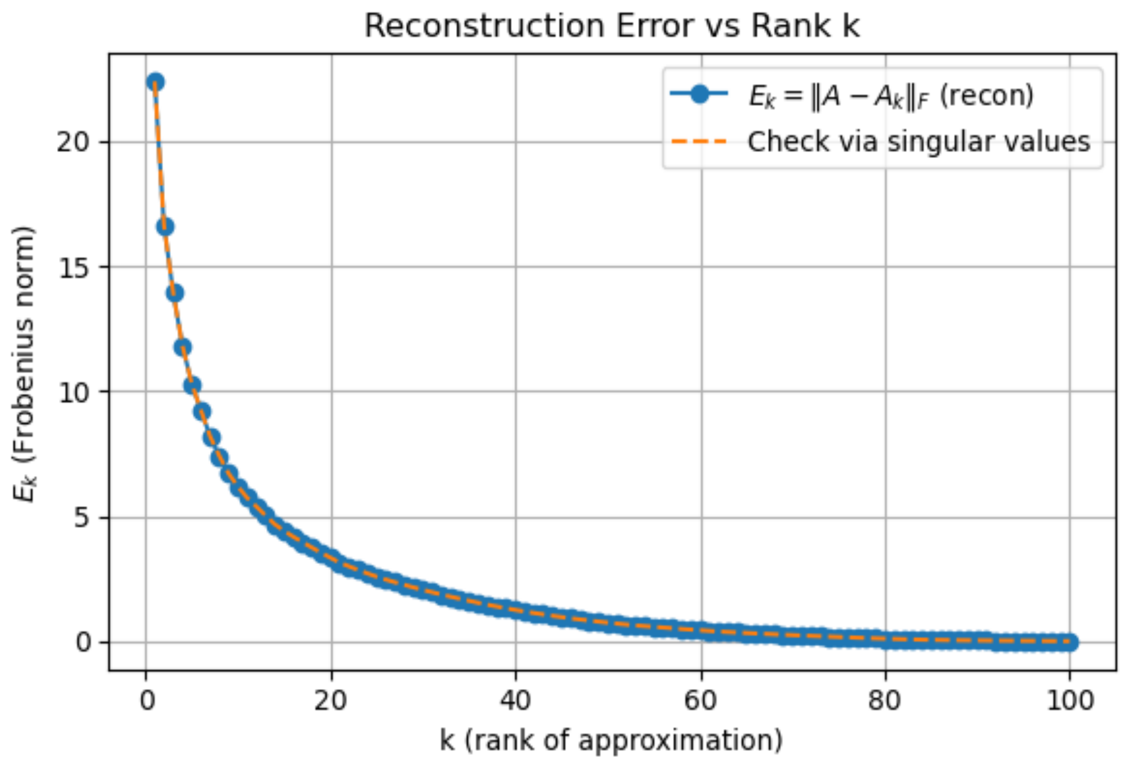
```python
fig, axes = plt.subplots(n_rows, 2, figsize=(8, 3*n_rows))

if n_rows == 1:
    axes = np.array([axes])  # ensure 2D indexing even for one row

for i, (k, Ak) in enumerate(reconstructions):
    # Left: original
    axes[i, 0].imshow(A, cmap='gray', vmin=0, vmax=1)
    axes[i, 0].set_title("Original (150×100)")
    axes[i, 0].axis('off')

    # Right: rank-k
    axes[i, 1].imshow(Ak, cmap='gray', vmin=0, vmax=1)
    axes[i, 1].set_title(f"Rank-k Reconstruction (k = {k})")
    axes[i, 1].axis('off')

plt.tight_layout()
plt.show()
```




Original (150×100)
Rank-k Reconstruction (k = 20)
Original (150×100)
Rank-k Reconstruction (k = 40)
Original (150×100)
Rank-k Reconstruction (k = 60)

Q2) (c) Error Analysis:

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

def reconstruct_rank_k(U, S, Vt, k):
    Uk = U[:, :k]
    Sk = np.diag(S[:k])
    Vtk = Vt[:k, :]
    Ak = Uk @ Sk @ Vtk
    return Ak

# Compute E_k = ||A - A_k||_F for all k = 1..min(m,n)
m, n = A.shape
kmax = min(m, n)
ks = np.arange(1, kmax + 1)

# Method A (direct reconstruction)
E_recon = np.empty_like(ks, dtype=float)
for i, k in enumerate(ks):
    Ak = reconstruct_rank_k(U, S, Vt, k)
    E_recon[i] = np.linalg.norm(A - Ak, ord='fro')

# Method B (theory check via singular values):
# For SVD, ||A - A_k||_F^2 = sum_{j>k} S_j^2
# This is a fast consistency check; should match E_recon (up to tiny round-
off).
E_svals = np.sqrt(np.array([np.sum(S[k:]**2) for k in ks]))

# normalized errors (relative to ||A||_F)
A_fro = np.linalg.norm(A, ord='fro')
E_rel = E_recon / A_fro

# Plot E_k vs k (and the singular-value-based check)
plt.figure(figsize=(6,4))
plt.plot(ks, E_recon, marker='o', label=r'$E_k = \|A-A_k\|_F$ (recon)')
plt.plot(ks, E_svals, linestyle='--', label='Check via singular values')
plt.xlabel('k (rank of approximation)')
plt.ylabel(r'$E_k$ (Frobenius norm)')
plt.title('Reconstruction Error vs Rank k')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

# Small table for requested k values
ks_report = [5, 10, 20, 40, 60]
print("k   E_k (Fro)    E_k/||A||_F")
for k in ks_report:
    Ek = E_recon[k-1]      # because ks starts at 1
    Erk = Ek / A_fro
    print(f"{k:<3} {Ek:>10.6f}   {Erk:>10.6f}")
```

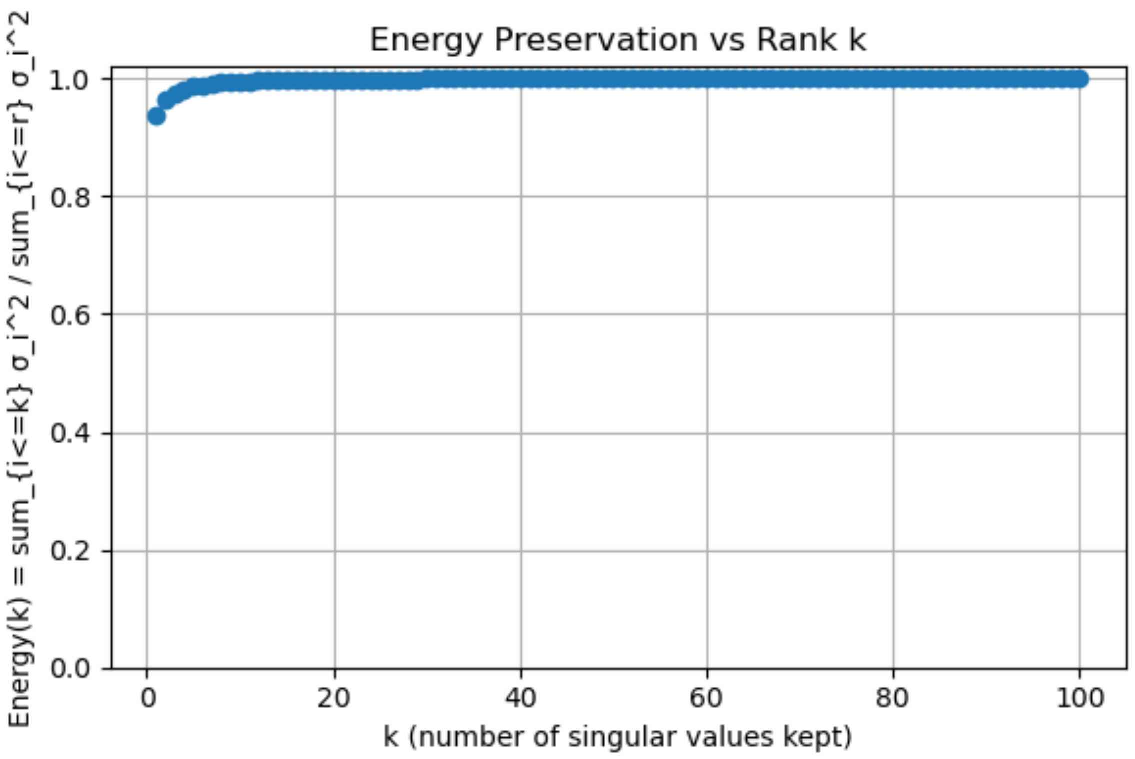


```
k   E_k (Fro)    E_k/||A||_F
5    10.304588     0.114925
10    6.182480     0.068952
20    3.340287     0.037254
40    1.249868     0.013940
60    0.436072     0.004863
```

Q2) (d) Energy Preservation:

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

# Assumes you already have S (singular values from np.linalg.svd(A,
full_matrices=False))
# If not, uncomment the next line:
# U, S, Vt = np.linalg.svd(A, full_matrices=False)

# Energy(k): cumulative proportion of sigma^2
sigma2 = S**2
total_energy = np.sum(sigma2)
cum_energy = np.cumsum(sigma2)                # length r
energy_k = cum_energy / (total_energy + 1e-18) # proportion in [0,1]

r = len(S)
ks = np.arange(1, r+1)

# Plot Energy(k) vs k
plt.figure(figsize=(6,4))
plt.plot(ks, energy_k, marker='o', linewidth=1)
plt.xlabel('k (number of singular values kept)')
plt.ylabel('Energy(k) = sum_{i<=k} σ_i^2 / sum_{i<=r} σ_i^2')
plt.title('Energy Preservation vs Rank k')
plt.ylim(0, 1.02)
plt.grid(True)
plt.tight_layout()
plt.show()

# Table for specific k values
ks_report = [5, 10, 20, 40, 60]
print("k    Energy(k)")
for k in ks_report:
    if k <= r:
        print(f"{k:<4} {energy_k[k-1]:.6f}")
    else:
        print(f"{k:<4} (k exceeds rank r={r})")
```

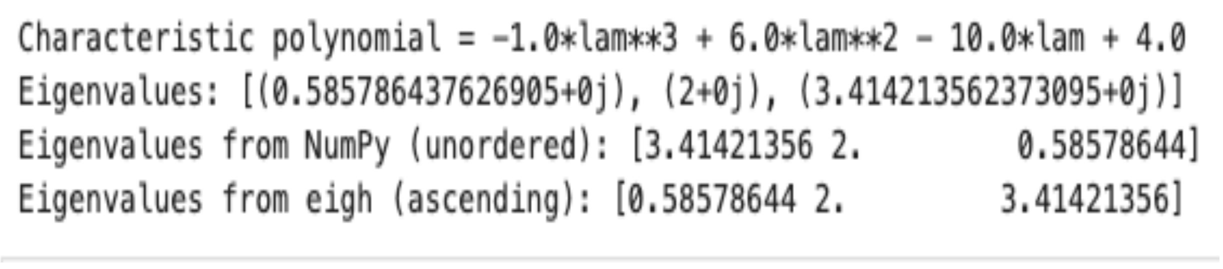
```python
# Find smallest k to reach common thresholds (e.g., 90%, 95%)
def min_k_for_threshold(energy_curve, thresh):
    idx = np.searchsorted(energy_curve, thresh, side='left')
    return None if idx >= len(energy_curve) else (idx+1)

for thresh in [0.90, 0.95]:
    k_needed = min_k_for_threshold(energy_k, thresh)
    if k_needed is None:
        print(f"No k achieves {int(thresh*100)}% energy.")
    else:
        print(f"Smallest k for >= {int(thresh*100)}% energy: k = {k_needed}")
```



```
k    Energy(k)
5    0.986792
10   0.995246
20   0.998612
40   0.999806
60   0.999976
Smallest k for >= 90% energy: k = 1
Smallest k for >= 95% energy: k = 2
```

Q3) (a) Eigenvalue Computation:

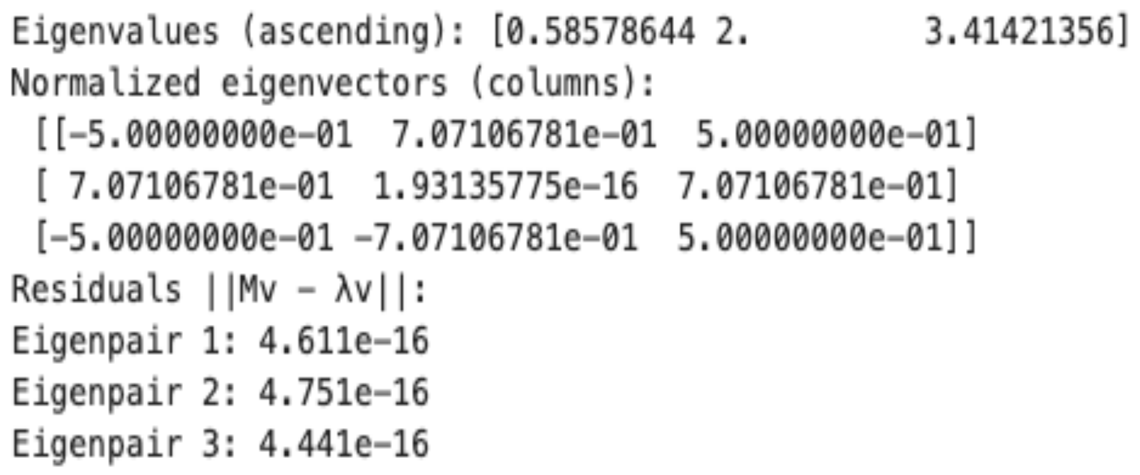
```python
import numpy as np
import sympy as sp

# Matrix
M = np.array([[2, 1, 0],
              [1, 2, 1],
              [0, 1, 2]], dtype=float)

# Characteristic polynomial using (|M - λI| = 0)
lam = sp.symbols('lam')
Ms = sp.Matrix(M)
char_poly = sp.expand((Ms - lam*sp.eye(3)).det())
print("Characteristic polynomial =", char_poly)
roots = [complex(r) for r in sp.nroots(char_poly)]
print("Eigenvalues:", roots)

# numeric eigenvalues (verification)
w, _ = np.linalg.eig(M)
print("Eigenvalues from NumPy (unordered):", w)

# For a symmetric matrix, eigh gives sorted real eigenvalues
w_eigh, _ = np.linalg.eigh(M)
print("Eigenvalues from eigh (ascending):", w_eigh)
```



```
Characteristic polynomial = -1.0*lam**3 + 6.0*lam**2 - 10.0*lam + 4.0
Eigenvalues: [(0.585786437626905+0j), (2+0j), (3.414213562373095+0j)]
Eigenvalues from NumPy (unordered): [3.41421356 2.         0.58578644]
Eigenvalues from eigh (ascending): [0.58578644 2.         3.41421356]
```

Q3) (b) Eigenvectors:

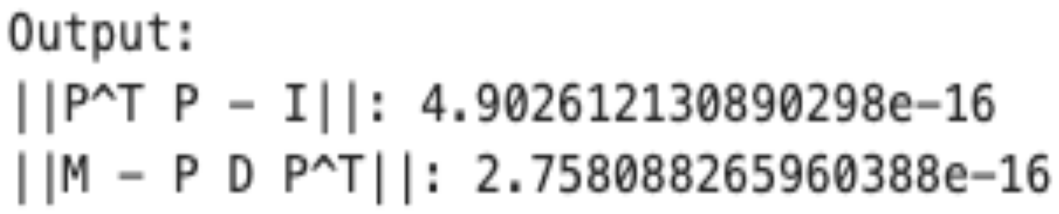
```python
# Use eigh (best for symmetric matrices) to get orthonormal eigenvectors
vals, vecs = np.linalg.eigh(M)   # vals ascending; columns of vecs are
eigenvectors

# Normalize (they already are, but we'll do it explicitly) and check residuals
residuals = []
V_norm = np.zeros_like(vecs)
for i in range(3):
    v = vecs[:, i]
    v = v / np.linalg.norm(v)
    V_norm[:, i] = v
    lhs = M @ v
    rhs = vals[i] * v
    res = np.linalg.norm(lhs - rhs)
    residuals.append(res)

print("Eigenvalues (ascending):", vals)
print("Normalized eigenvectors (columns):\n", V_norm)
print("Residuals ||Mv - λv||:")
for i, res in enumerate(residuals, start=1):
    print(f"Eigenpair {i}: {res:.3e}")
```



```
Eigenvalues (ascending): [0.58578644 2.         3.41421356]
Normalized eigenvectors (columns):
 [[-5.00000000e-01  7.07106781e-01  5.00000000e-01]
 [ 7.07106781e-01  1.93135775e-16  7.07106781e-01]
 [-5.00000000e-01 -7.07106781e-01  5.00000000e-01]]
Residuals ||Mv - λv||:
Eigenpair 1: 4.611e-16
Eigenpair 2: 4.751e-16
Eigenpair 3: 4.441e-16
```

Q3) (c) Diagonalizability:

```
# Since M is real and symmetric, so it is diagonalizable by an orthogonal matrix
# From eigh above: vecs is orthonormal (P), vals are eigenvalues (diagonal D)
# Therefore, M = PDP^T
P = V_norm              # columns are orthonormal eigenvectors
D = np.diag(vals)        # diagonal of eigenvalues

# Reconstruct M and measure error
M_recon = P @ D @ P.T
orth_err = np.linalg.norm(P.T @ P - np.eye(3))
recon_err = np.linalg.norm(M - M_recon)

print("Output:")
print("||P^T P - I||:", orth_err)
print("||M - P D P^T||:", recon_err)
```



```
Output:
||P^T P - I||: 4.902612130890298e-16
||M - P D P^T||: 2.758088265960388e-16
```

Q3) (d) Generalization:

PAGE

Solution :-

Given $M = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}$

General case of M being an $n \times n$ tridiagonal matrix by replacing 2 with a and 1 with b.

$$T = \begin{bmatrix} a & b & 0 & \cdots & 0 & 0 & 0 \\ b & a & b & \cdots & 0 & 0 & 0 \\ 0 & b & a & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a & b & 0 \\ 0 & 0 & 0 & \cdots & b & a & b \\ 0 & 0 & 0 & \cdots & 0 & b & a \end{bmatrix}_{n*n}$$

Here, every diagonal entry in principal diagonal is 'a' and the entries just below and above the principal diagonal elements are 'b', and every other position in matrix has 0.
Hence, this is a symmetric tridiagonal matrix.

For Eigenvalues (λ) such that there exists a vector x with
$Tx = \lambda x$, if $x = (x_1, x_2, \dots, x_n)^T$ is a vector

$$\begin{bmatrix} a & b & 0 & \cdots & 0 & 0 & 0 \\ b & a & b & \cdots & 0 & 0 & 0 \\ 0 & b & a & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a & b & 0 \\ 0 & 0 & 0 & \cdots & b & a & b \\ 0 & 0 & 0 & \cdots & 0 & b & a \end{bmatrix}_{n\times n} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix}_{n\times 1} = \begin{bmatrix} \lambda x_1 \\ \lambda x_2 \\ \lambda x_3 \\ \vdots \\ \vdots \\ \vdots \\ \lambda x_n \end{bmatrix}$$

$\therefore$ from above eqn we get

$$a x_1 + b x_2 = \lambda x_1 \quad - \textcircled{i} \quad \longrightarrow$$ from 1st row multiplied by vector x

Similarly $b x_1 + a x_2 + b x_3 = \lambda x_2$

$b x_2 + a x_3 = \lambda x_3$

Last row can be represented as:

$$b x_{n-1} + a x_n = \lambda x_n \quad - \text{(ii)}$$

Any middle row can be represented as

$$b x_{j-1} + a x_j + b x_{j+1} = \lambda x_j \quad \text{where } j = 2, 3, \ldots n-1$$

Hence,

each entry of x is linked by its neighbours by this recurrence..

$$b x_{j-1} + a x_j + b x_{j+1} - \lambda x_j = 0$$

$$\therefore \quad b x_{j-1} + (a-\lambda) x_j + b x_{j+1} = 0 \quad - \text{(iii)}$$

So, matrix condition is now a recurrence relation, where every entry is tied to the previous & the next.

At ends, we can pretend there are 2 extra entries:

$x_0 = 0$, before first row & $x_{n+1} = 0$ after last row

This way, the recurrence applies uniformly from $j = 1$ to n.

So matrix problem is reduced to,

$$b x_{j-1} + (a-\lambda) x_j + b x_{j+1} = 0 \quad - \text{(iv)} \quad \text{where } j = 1, 2, \ldots n \ \& \ x_0 = 0 \ \& \ x_{n+1} = 0$$

For such recurrences, a standard trick is to guess that the entries look like powers of some number 'r':

Let $x_j = r^j$

Substituting this to above equation (iv)

$$b r^{j-1} + (a-\lambda) r^j + b r^{j+1} = 0 \quad - \text{(v)}$$

Divide by r^{j-1}

$$\Rightarrow \frac{b r^{j-1}}{r^{j-1}} + \frac{(a-\lambda) r^j}{r^{j-1}} + \frac{b r^{j+1}}{r^{j-1}} = 0$$

$$\Rightarrow b + (a-\lambda) r + b r^2 = 0 \quad - \text{(vi)}$$

This is a quadratic eqn in 'r'.

Solving the equation $br^2 + (a-\lambda)r + b = 0$

product of roots $= \frac{b}{b} = \underline{\underline{1}}$

$\left[\begin{array}{l} \because ax^2+bx+c=0 \\ \text{Product of roots} = \left(\frac{c}{a}\right) \\ \text{Sum of roots} = \left(-\frac{b}{a}\right) \end{array}\right]$

& Sum of roots $= \left(-\frac{(a-\lambda)}{b}\right) = r + \frac{1}{r}$ — (vii)

$\because$ If one of the solution of above equation is r, then other will be $1/r$.

$\because$ Roots are complex conjugate pair that are also reciprocals, they must be located on the unit circle in complex plane.

$\therefore$ Any complex number with a modulus of 1 can be written in the form $r = e^{i\theta}$ using Euler's formula.

$\therefore \frac{1}{r} = e^{-i\theta}$

$\because$ Sum of roots $= r + \frac{1}{r} = e^{i\theta} + e^{-i\theta}$

Now,

$$r + \frac{1}{r} = (\cos\theta + i\sin\theta) + (\cos\theta - i\sin\theta) \quad \text{[Using Euler's formula]}$$

$$r + \frac{1}{r} = 2\cos\theta$$

$$-\frac{(a-\lambda)}{b} = 2\cos\theta \quad \left(\because r + \frac{1}{r} = \left(-\frac{(a-\lambda)}{b}\right)\right)$$

$$\therefore \boxed{\lambda = a + 2b\cos\theta} \quad \text{— (viii)} \quad \boxed{\text{This is the formula for Eigen Values}}$$

Now, we need to find which values of θ are allowed in above eq^n.

We set $x_0 = 0$ & $x_{n+1} = 0$. Therefore General Solution is

$$x_j = Ae^{ij\theta} + Be^{-ij\theta} \quad \text{— (ix)}$$

Substitute $e^{ij\theta} = \cos(j\theta) + i\sin(j\theta)$ & $e^{-ij\theta} = \cos(j\theta) - i\sin(j\theta)$ in above eq^n (ix)

we get $x_j = (A+B)\cos(j\theta) + i(A-B)\sin(j\theta)$

Let $C = i(A-B)$ & $D = (A+B)$

$$\therefore x_j = D\cos(j\theta) + C\sin(j\theta) \quad \text{— (x)}$$

Applying Boundary Conditions :

(i) At $x_0 = 0$ in eq (x)

$$x_0 = D\cos(0) + C\sin(0) \qquad [\because \sin(0) = 0 \ \& \ \cos(0) = 1]$$

$$x_0 = D(1) + C(0) = D$$

$\because x_0 = 0 \ \therefore \ D = 0$

Plugging $D=0$ in eq (x)

$$\boxed{x_j = C\sin(j\theta)} \quad - (xi)$$

(ii) At $x_{n+1} = 0$ in eq (x)

$$x_{n+1} = D\cos((n+1)\theta) + C\sin((n+1)\theta)$$

$$[\because x_{n+1} = 0 \ \& \ D = 0]$$

$$C\sin((n+1)\theta) = 0$$

$$\sin((n+1)\theta) = 0$$

$$(n+1)\theta = \sin^{-1}(0)$$

[$\because$ C is scaling factor & we don't want C=0, as it will make whole vector as zero, which then will not be eigen vector.]

$$\theta_m = \frac{m\pi}{(n+1)} \quad , \text{where } [m = 1, 2, \dots n] \quad - (xii)$$

Substituting value of θ_m to eigenvalue formula, we get

$$\boxed{\lambda_m = a + 2b\cos\left(\frac{m\pi}{n+1}\right)} \quad (xiii)$$

where $[m = 1, 2, \dots n]$