**Mathematics for Machine Learning – Assignment Solutions**  
  
  
**Q1) (1)** Generate dataset X ∈ 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:**

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

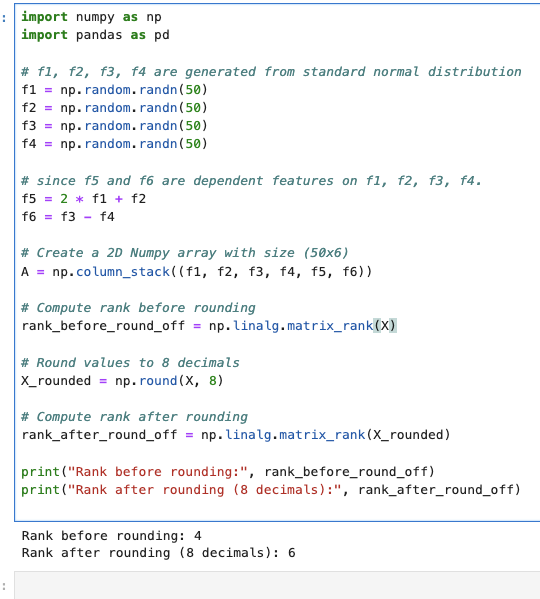
X\_rounded = np.round(X, 8)

# 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)

****

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

1. **Compute covariance matrix:**

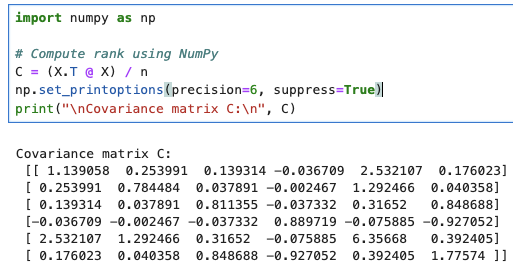
import numpy as np

# Compute rank using NumPy

C = (X.T @ X) / n

np.set\_printoptions(precision=6, suppress=True)

print("\nCovariance matrix C:\n", C)



1. **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

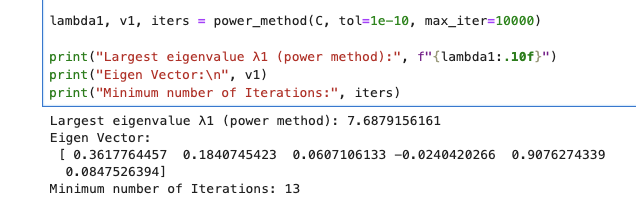
return lambda\_new, x, max\_iter

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)

****

1. **Next largest eigenvalue λ2 and its corresponding eigenvector V2 by applying power method on C−V1V1TC.**

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)

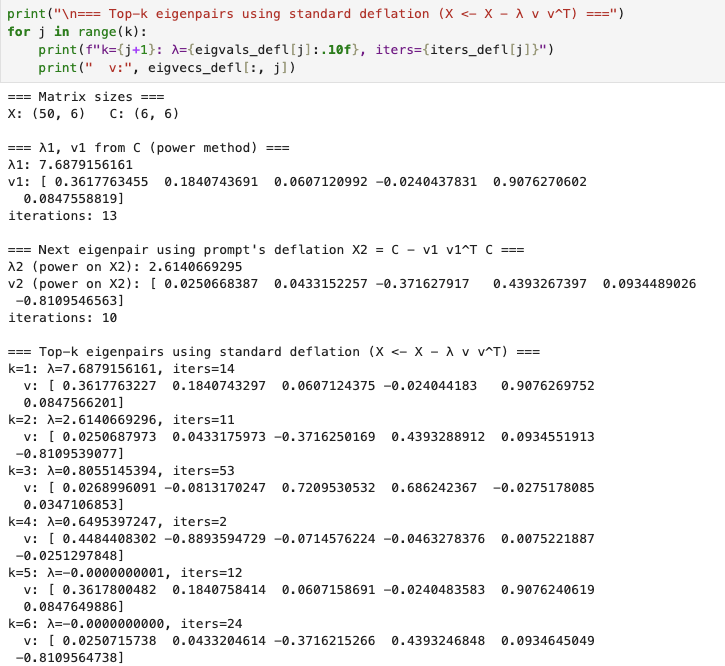
print("iterations:", iters2\_prompt)

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])



1. **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}",

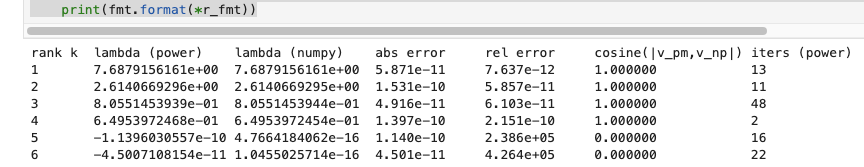
f"{r[4]:.3e}",

f"{r[5]:.6f}",

r[6]

]

print(fmt.format(\*r\_fmt))



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.

1. **All eigenvalues and eigenvector using**

Rank 1 (λ₁≈ 7.6879156161, 13 iters):

* Next eigenvalue λ₂ ≈ 2.6140666930
* Ratio λ₂/λ₁ ≈ 2.6141 / 7.6879 ≈ 0.34
* This means that each step the leftover error shrinks by around 34% and hence converged quickly in 13 iterations

Rank 2 (λ₂ ≈ 2.6140666930, 11 iters)

* After removing rank-1, the new competitor is λ₃ ≈ 0.8055145394
* Ratio λ₃/λ₂ ≈ 0.8055 / 2.6141 ≈ 0.31
* This means that error shrinks a bit faster than rank-1 and hence converged in 11 iterations only

Rnak 3 (λ₃ ≈ 0.8055145394, 48 iters)

* Now the next competitor is λ₄ ≈ 0.6495397245
* Ratio λ₄/λ₃ ≈ 0.6495 / 0.8055 ≈ 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 (λ₄ ≈ 0.6495397245, 2 iters)

* Next competitor is λ₅ ≈ 4.77e-16 (essentially zero).
* Ratio λ₅/λ₄ ≈ ~0 / 0.6495 ≈ 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)

* λ₅ ≈ 4.77e-16, λ₆ ≈ 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 (≈ 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 ∈ R150 x 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)

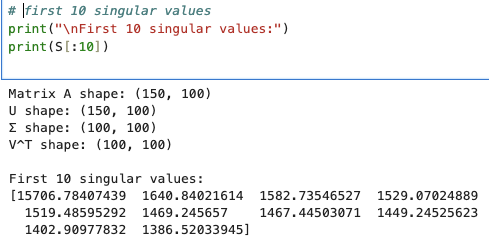
print("Σ shape:", Sigma.shape) # (100, 100)

print("V^T shape:", Vt.shape) # (100, 100)

# first 10 singular values

print("\nFirst 10 singular values:")

print(S[:10])



**Q2) (b)** Reconstruction with Multiple Ranks

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)

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

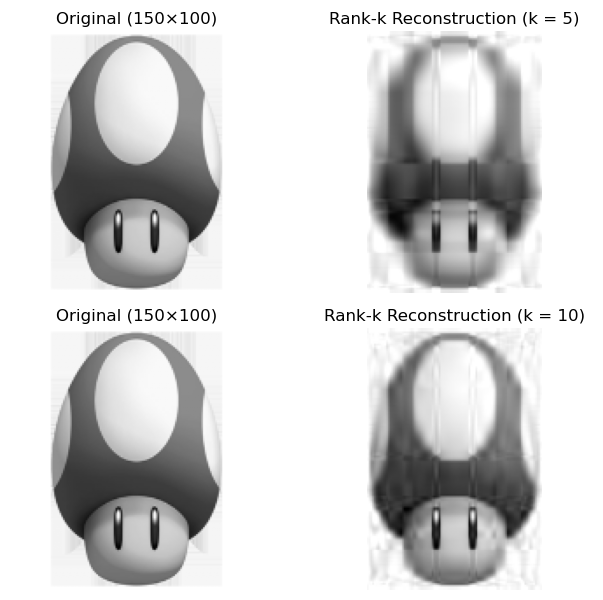
axes[i, 1].imshow(Ak, cmap='gray', vmin=0, vmax=1)

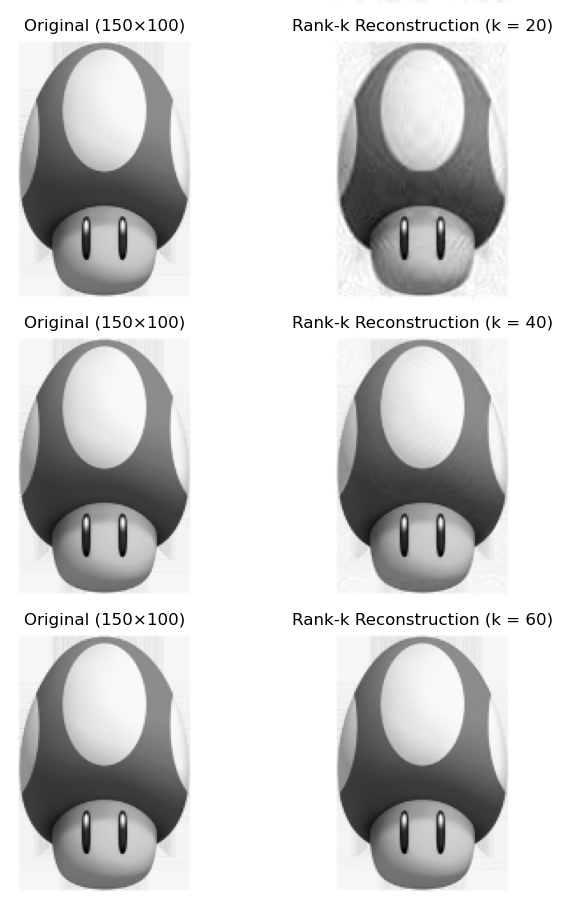
axes[i, 1].set\_title(f"Rank-k Reconstruction (k = {k})")

axes[i, 1].axis('off')

plt.tight\_layout()

plt.show()





**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]

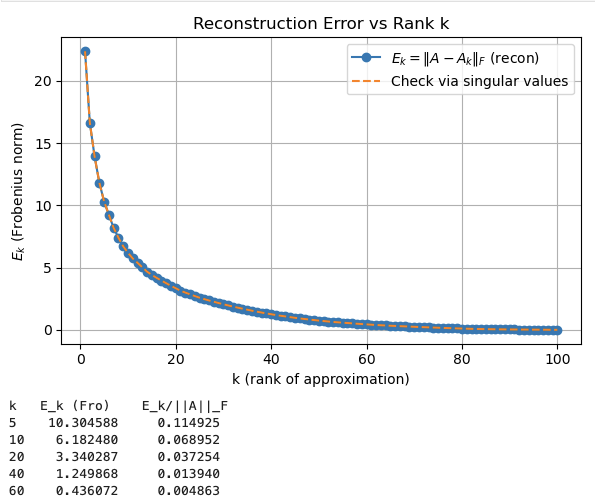
print("k E\_k (Fro) E\_k/||A||\_F")

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}")



**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})")

# 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]:

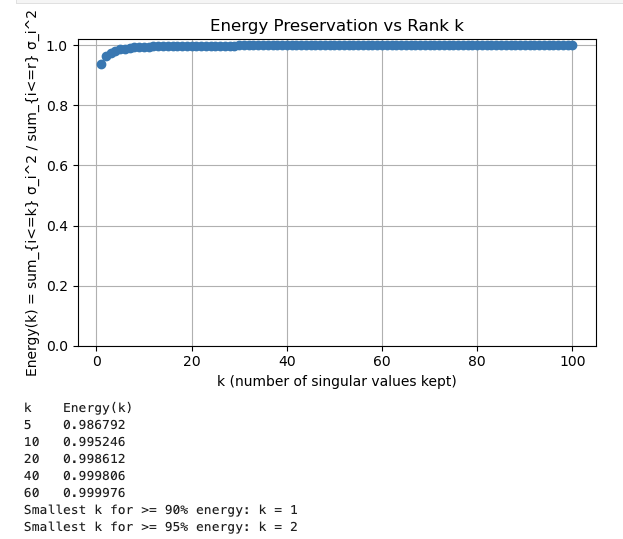
k\_needed = min\_k\_for\_threshold(energy\_k, thresh)

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}")



**Q3) (a)** Eigenvalue Computation:

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)

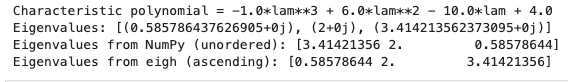
w, \_ = np.linalg.eig(M)

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)



**Q3) (b)** Eigenvectors:

# 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)

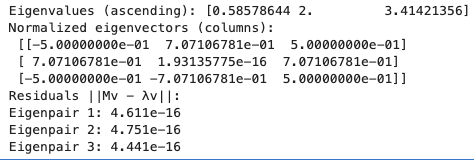
print("Eigenvalues (ascending):", vals)

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}")



**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)



**Q3) (d)** Generalization:

