
PCA Interpretation and NumPy Computation

ECE 57000 — September 18, 2026

David I. Inouye

| Wednesday proved **which PCA subspace is optimal**

- For centered data, minimizing reconstruction error is equivalent to maximizing $\|XW\|_F^2$.
- Every orthonormal W retains at most $\sum_{j=1}^k \sigma_j^2$; choosing $W = V_k$ attains that bound.
- The discarded squared singular values give reconstruction error: $\|X - X_k\|_F^2 = \sum_{j>k} \sigma_j^2$.
- Each PCA coordinate has sample variance $\sigma_j^2 / (n - 1)$; choosing k still depends on the intended use.

Next: revisit the variance interpretation, finish PCA's geometric interpretation, then implement it in NumPy.

| Squared singular values give **the variance of each PCA coordinate**

The j th coordinate across observations is $Z_{:j} = X\mathbf{v}_j = \sigma_j\mathbf{u}_j$. Since X is centered, this coordinate also has mean zero.

$$\begin{aligned}\text{Var}_{\text{sample}}(Z_{:j}) &= \frac{1}{n-1} \sum_{i=1}^n Z_{ij}^2 \\ &= \frac{\sigma_j^2}{n-1} \|\mathbf{u}_j\|_2^2 \\ &= \frac{\sigma_j^2}{n-1}\end{aligned}$$

(Variance of a centered coordinate; $n > 1$)

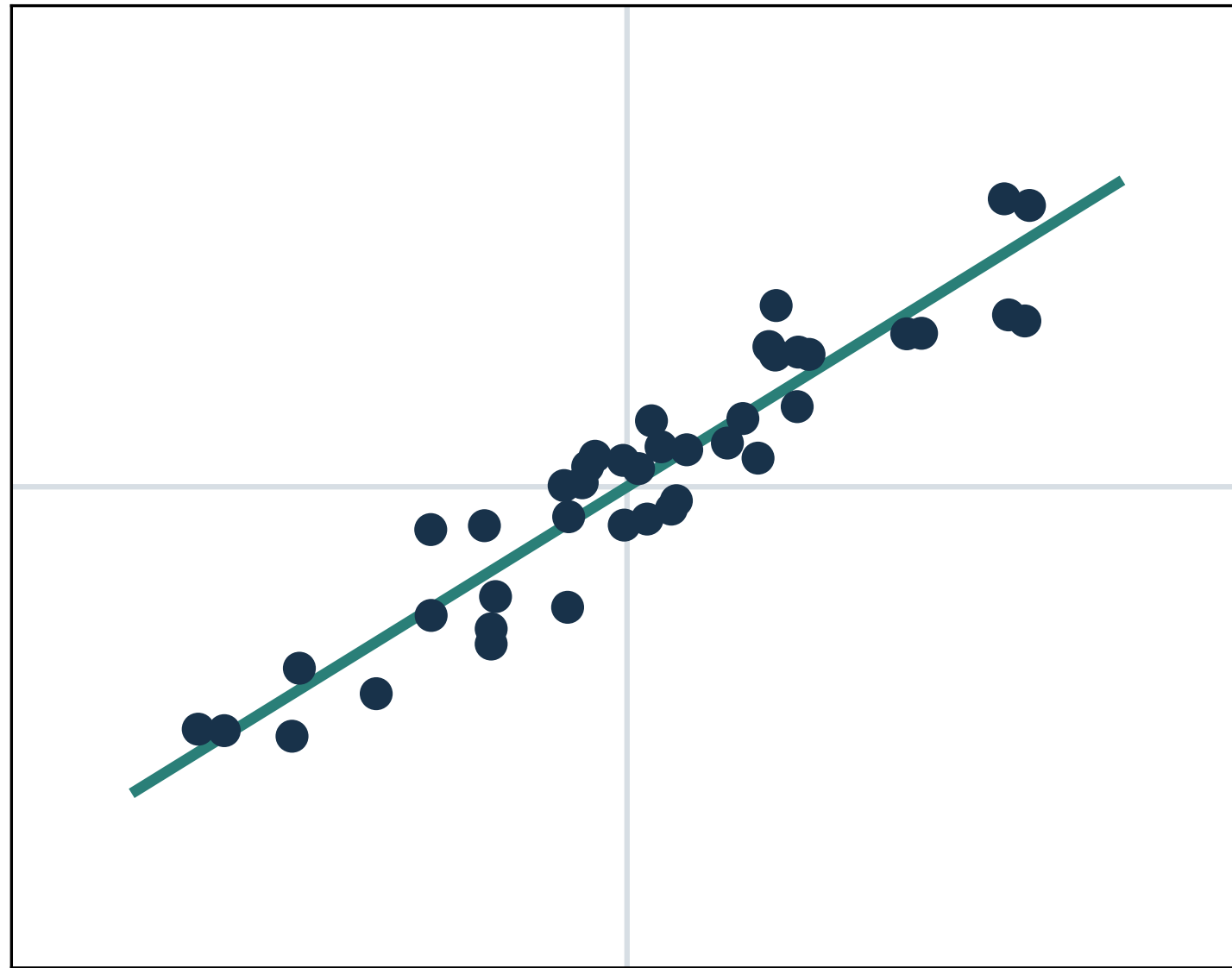
(Substitute $Z_{:j} = \sigma_j\mathbf{u}_j$)

($\mathbf{u}_j$ is a unit vector)

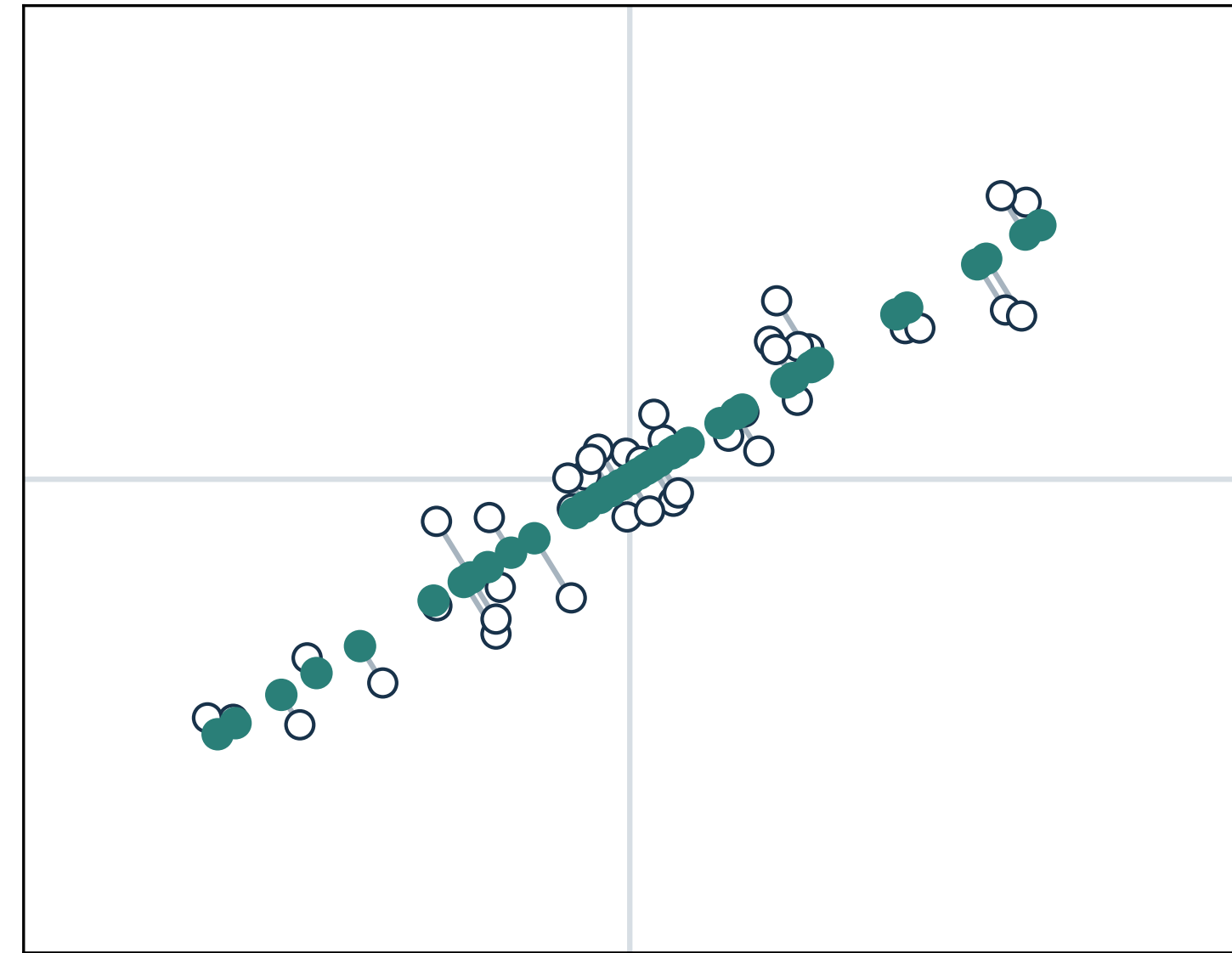
The leading PCA coordinates therefore have the **largest variances**. This interprets the solution we already proved.

| One retained direction can reconstruct a narrow data cloud

Centered Observations



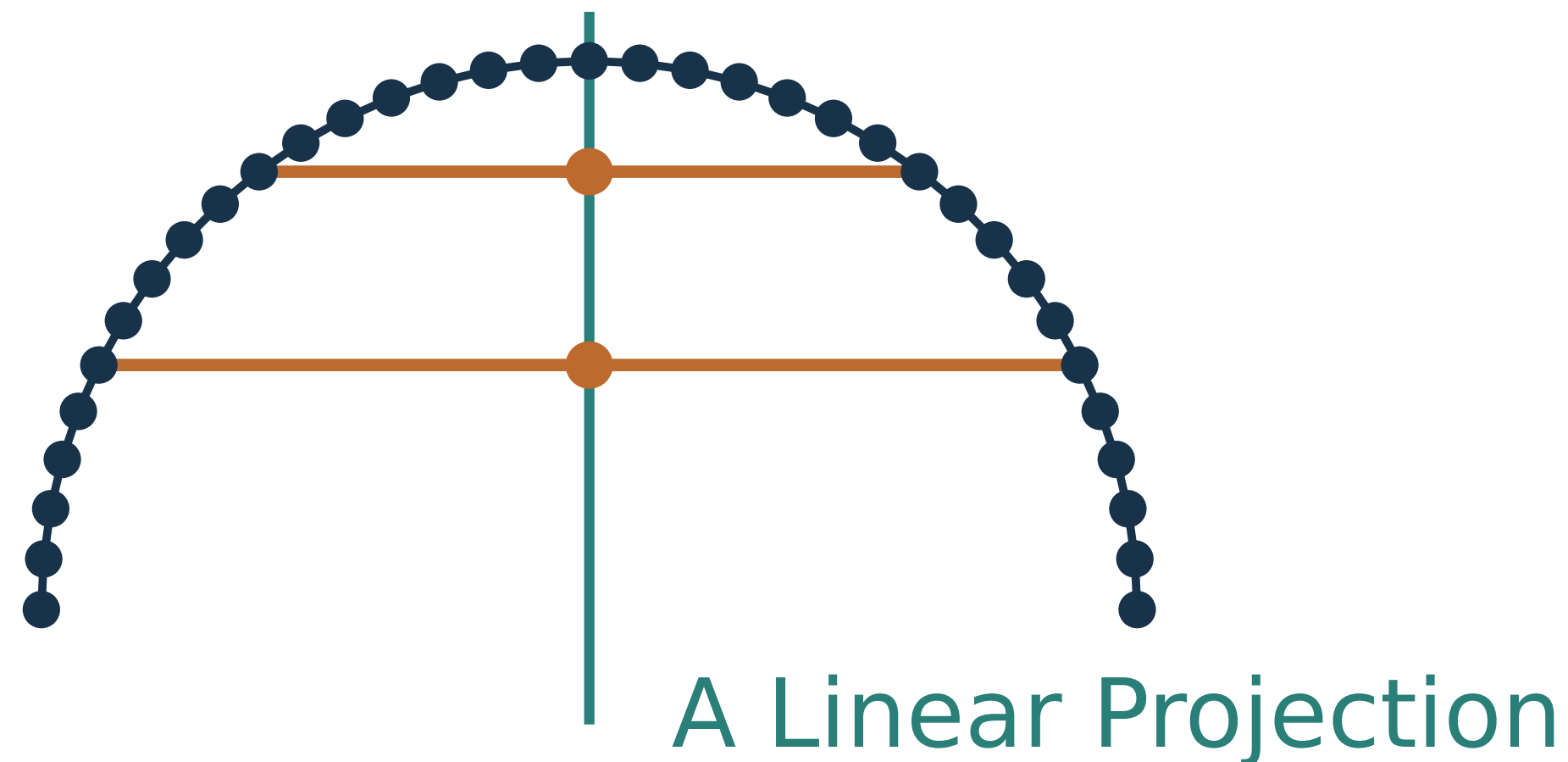
Rank-One Reconstruction



Original-scale reconstruction: $\hat{\tilde{x}}_i = \mu + Wz_i$.

| A curved structure can be **low-dimensional** without being linear

One Curved Coordinate



Would one linear feature reconstruct every point exactly? Could a nonlinear coordinate do so? What does “one-dimensional” mean in each claim?

| PCA solves the stated problem; usefulness still needs evidence

- **Chosen objective:** squared error in the selected measurements and units.
- **Mathematical solution:** leading right singular vectors of centered data.
- **Remaining questions:** which features and preprocessing, how many components, and whether the retained variation matters for the intended use.

Next: translate this mathematical contract into arrays and reliable numerical computation—then ask what the observed data justify claiming about new cases.

NumPy and Numerical Computation for PCA

David I. Inouye

Dynamic Content · Last Updated: September 17, 2026

| The PCA equations give us a computation to implement

Let $\widetilde{X} \in \mathbb{R}^{n \times d}$ contain the original observations as rows.

Operation	Mathematics	Shape
Center	$X = \widetilde{X} - \mathbf{1}\mu^T$	$n \times d$
Choose directions	$X = U\Sigma V^T, W = V_k$	$d \times k$
Encode	$Z = XW$	$n \times k$
Reconstruct	$\widehat{\widetilde{X}} = ZW^T + \mathbf{1}\mu^T$	$n \times d$

Route: represent the objects, implement the operations, check the numerical result.

| Lists and arrays give different meanings to *

```
1 import numpy as np
2
3 values = [2, 5, 8]
4 x = np.array(values, dtype=np.float64)
5
6 print("type(values):", type(values))
7 print("type(x):", type(x))
8 print(f"values * 2: {values * 2!r}")
9 print(f"x * 2: {x * 2!r}")
10 print("x.shape:", x.shape)
11 print("x.dtype:", x.dtype)
```

```
type(values): <class 'list'>
type(x): <class 'numpy.ndarray'>
values * 2: [2, 5, 8, 2, 5, 8]
x * 2: array([ 4., 10., 16.])
x.shape: (3,)
x.dtype: float64
```

- A list is a Python sequence. Multiplication by an integer repeats it.
- An array stores values with a common data type. Here multiplication scales each entry.
- `shape` describes the axes. `dtype` describes how entries are stored.

| A one-dimensional array has **no row or column axis to swap**

Shapes: $(d,)$ = one observation; (n, d) = observations $\times$ features; (b, n, d) adds a batch axis.

```
1 x = np.array([2., 5., 8.])
2 print("x.shape:", x.shape)
3 print("x.T.shape:", x.T.shape)
4 print("x[:, None].shape:", x[:, None].shape)
5 print("x[None, :].shape:", x[None, :].shape)
```

```
x.shape: (3,)
x.T.shape: (3,)
x[:, None].shape: (3, 1)
x[None, :].shape: (1, 3)
```

- `None` (equivalently `np.newaxis`) **inserts a length-one axis** at that position.
- `x[:, None]` makes a column; `x[None, :]` makes a row.
- For 2D arrays, `.T` swaps the row and column axes.

Indexing: `A[i, :]` selects observation i ; `A[:, j]` selects feature j .

| * multiplies entries; @ implements matrix multiplication

```
1 P = np.array([[1., 2.],
2               [3., 4.]])
3 Q = np.array([[2., 0.],
4               [1., 2.]])
5 print("P * Q (entrywise):\n", P * Q)
6 print("P @ Q (matrix product):\n", P @ Q)
```

P * Q (entrywise):

[[2. 0.]

[3. 8.]]

P @ Q (matrix product):

[[4. 4.]

[10. 8.]]

For PCA, $Z = X @ W$ maps $(n, d) @ (d, k)$ to (n, k) .

The upper-left entries are $1 \cdot 2 = 2$ and $1 \cdot 2 + 2 \cdot 1 = 4$. Both results have shape $(2, 2)$. Shape alone cannot tell us which operation we intended.

| Centering averages across observations for each feature

For each feature, $\mu_j = \frac{1}{n} \sum_{i=1}^n \widetilde{X}_{ij}$.

```
1 A = np.array([[2., 10.],
2               [4., 14.], [6., 18.]])
3 mu = A.mean(axis=0)
4 X = A - mu
5 print("mu (feature means):", mu)
6 print("X (centered data):\n", X)
```

```
mu (feature means): [ 4. 14.]
X (centered data):
[[-2. -4.]
 [ 0.  0.]
 [ 2.  4.]]
```

Axis rule: count positions in the shape tuple from zero. Aggregate over the chosen dimension and remove it by default. The resulting shapes are:

```
1 T = np.zeros((2, 3, 4))
2 print("T.shape:", T.shape)
3 print("axis=0:", T.mean(axis=0).shape)
4 print("axis=1:", T.mean(axis=1).shape)
5 print("axis=2:", T.mean(axis=2).shape)
```

```
T.shape: (2, 3, 4)
axis=0: (3, 4)
axis=1: (2, 4)
axis=2: (2, 3)
```

Why does $X = A - \mu$ work? Ordinary matrix subtraction requires equal shapes. A matrix minus a vector is undefined. NumPy supplies **broadcasting**.

| Broadcasting compares dimensions **from right to left**

At each position, **either** of two rules must hold:

1. The two dimension sizes are **equal**: keep that size.
2. One size is **1**: reuse its value along the other dimension's size.

Missing dimensions count as 1s (marked * below). If neither rule holds at any position, broadcasting fails.

Vector (4,)

```
(3, 4)
(*, 4)
-----
(3, 4)
```

Matrix (3, 1)

```
(2, 3, 4)
(*, 3, 1)
-----
(2, 3, 4)
```

Both inputs expand

```
(3, 1)
(1, 4)
-----
(3, 4)
```

Scalar ()

```
(2, 3, 4)
(*, *, *)
-----
(2, 3, 4)
```

Vector (3,): incompatible

```
(3, 4)
(*, 3)
-----
Error
```

Rightmost sizes **4 and 3** are neither equal nor 1.

| Mathematics makes the repeated mean **an explicit matrix**

$$\underbrace{X}_{n \times d} = \underbrace{\widetilde{X}}_{n \times d} - \underbrace{\mathbf{1}_n \boldsymbol{\mu}^T}_{(n \times 1)(1 \times d) = n \times d}.$$

In NumPy, `A - mu` produces the same entries: $X_{ij} = \widetilde{X}_{ij} - \mu_j$.

```
1 ones = np.ones((A.shape[0], 1))
2 mu_matrix = ones @ mu[None, :]
3 print("mu_matrix:\n", mu_matrix)
4 print("A - mu:\n", A - mu)
5 print("Same result:",
6       np.allclose(A - mu_matrix, A - mu))
```

```
mu_matrix:
[[ 4. 14.]
 [ 4. 14.]
 [ 4. 14.]]
A - mu:
[[-2. -4.]
 [ 0.  0.]
 [ 2.  4.]]
Same result: True
```

`A: (3, 2); mu: (2, )`. **Broadcasting avoids copying the repeated means**, saving allocation and memory traffic. The output still needs storage.

| `keepdims=True` preserves reduced axes **with length one**

For `T.shape == (2, 3, 4)`, reducing `axis=1` gives `(2, 4)` by default. With `keepdims=True`, it gives `(2, 1, 4)`: the reduced position stays explicit.

```
1 reduced = T.mean(axis=1)
2 kept = T.mean(axis=1, keepdims=True)
3 restored = reduced[:, None, :]
4 print("T.shape:", T.shape)
5 print("reduced.shape:", reduced.shape)
6 print("kept.shape:", kept.shape)
7 print("restored.shape:", restored.shape)
8 print("(T - kept).shape:", (T - kept).shape)
```

```
T.shape: (2, 3, 4)
reduced.shape: (2, 4)
kept.shape: (2, 1, 4)
restored.shape: (2, 1, 4)
(T - kept).shape: (2, 3, 4)
```

- `keepdims` preserves a reduced axis; `None` inserts a new axis where specified.
- `(2, 1, 4)` broadcasts back against `(2, 3, 4)`.
- `(2, 4)` would align as `(1, 2, 4)` and **fail** at the middle position.

| Division broadcasts **one standard deviation per feature**

For positive feature standard deviations s_j , $H_{ij} = (\widetilde{X}_{ij} - \mu_j) / s_j$. In matrix notation, $H = X \text{diag}(1/s_1, \dots, 1/s_d)$.

```
1 std = A.std(axis=0)
2 H = (A - mu) / std
3 H_mul = (A - mu) * (1 / std)
4 print("std:", std.round(3))
5 print("H.round(3):\n", H.round(3))
6 print("H.std(axis=0):", H.std(axis=0))
7 print("Division equals multiplication:",
8       np.allclose(H, H_mul))
```

```
std: [1.633 3.266]
```

```
H.round(3):
```

```
[[ -1.225 -1.225]
```

```
 [ 0.      0.   ]
```

```
 [ 1.225  1.225]]
```

```
H.std(axis=0): [1. 1.]
```

```
Division equals multiplication: True
```

$(3, 2) / (2,)$ reuses each feature's scale across observations. The same shape rules apply to elementwise $+$, $-$, $*$, and $/$. **Scaling changes the relative weight of features in PCA's reconstruction error.**

Code can run successfully while centering the wrong objects

```
1 A = np.array([[2., 10.],  
2               [4., 14.], [6., 18.]])  
3 B = A - A.mean(axis=1, keepdims=True)  
4 print("B:\n", B)  
5 print("B.mean(axis=1):", B.mean(axis=1))  
6 print("B.mean(axis=0):", B.mean(axis=0))
```

```
B:  
[[-4.  4.]  
 [-5.  5.]  
 [-6.  6.]  
 B.mean(axis=1): [0. 0. 0.]  
 B.mean(axis=0): [-5.  5.]
```

What does this code subtract? Which averages become zero? Would this prepare the data for the PCA problem we defined? Repair the expression and explain why.

| NumPy returns the right singular vectors **as rows of Vt**

```
1 U, s, Vt = np.linalg.svd(  
2     X, full_matrices=False)  
3 W = Vt[:k, :].T
```

`full_matrices=False` omits extra directions multiplied by zero rows or columns of rectangular Σ . This **reduced SVD** keeps $m = \min(n, d)$ singular values; selecting the first k is a separate truncation.

Returned array	Shape	Mathematical role
U	(n, m)	Left singular vectors
s	(m,)	Singular values, largest first
Vt	(m, d)	Rows are right singular vectors transposed
W	(d, k)	First k right singular vectors as columns

| Encoding and reconstruction follow the same equations as before

`Xraw` stores $\widetilde{X}$, the original observations.

```
1 Xraw = np.array([[2., 10.], [4., 15.],  
2                 [6., 17.], [8., 22.]])  
3 mu = Xraw.mean(axis=0)  
4 X = Xraw - mu  
5 U, s, Vt = np.linalg.svd(  
6     X, full_matrices=False)  
7 k = 1  
8 W = Vt[:k, :].T  
9 Z = X @ W  
10 Xraw_hat = Z @ W.T + mu  
11 print("Z.round(3):\n", Z.round(3))
```

```
Z.round(3):  
[[-6.708]  
 [-1.347]  
 [ 1.347]  
 [ 6.708]]
```

For $X \in \mathbb{R}^{n \times d}$ and $W \in \mathbb{R}^{d \times k}$, $\widehat{X} = (XW)W^T = X(WW^T)$.

Centered reconstruction	Arithmetic cost	Intermediate
<code>Xhat = (X @ W) @ W.T</code>	$O(ndk)$	$n \times k$
<code>Xhat = X @ (W @ W.T)</code>	$O(d^2k + nd^2)$	$d \times d$

| The computed arrays show PCA's shapes and reconstruction

```
1 print("X.shape:", X.shape)
2 print("U.shape:", U.shape)
3 print("s.shape:", s.shape)
4 print("Vt.shape:", Vt.shape)
5 print("W.shape:", W.shape)
6 print("Z.shape:", Z.shape)
7 print("Xraw_hat.shape:", Xraw_hat.shape)
8 print("s.round(4):", s.round(4))
9 print("Xraw_hat.round(3):\n",
10       Xraw_hat.round(3))
11 error = np.sum((Xraw - Xraw_hat)**2)
12 print(f"error (squared): {error:.6f}")
```

```
X.shape: (4, 2)
U.shape: (4, 2)
s.shape: (2,)
Vt.shape: (2, 2)
W.shape: (2, 1)
Z.shape: (4, 1)
Xraw_hat.shape: (4, 2)
s.round(4): [9.6755 0.6201]
Xraw_hat.round(3):
[[ 1.923 10.04 ]
 [ 4.382 14.803]
 [ 5.618 17.197]
 [ 8.077 21.96 ]]
error (squared): 0.384552
```

We retained one direction, but `full_matrices=False` still computed **both** singular values.

| New observations use **the fitted mean and directions**

```
1 # Apply the representation fitted on Xraw
2 Xraw_new = np.array([[5., 16.], [7., 20.]])
3 Z_new = (Xraw_new - mu) @ W
4 Xraw_new_hat = Z_new @ W.T + mu
5 print("Z_new.round(3):\n", Z_new.round(3))
6 print("Xraw_new_hat.round(3):\n",
7       Xraw_new_hat.round(3))
```

```
Z_new.round(3):
[[0.   ]
 [4.472]]
Xraw_new_hat.round(3):
[[ 5.    16.   ]
 [ 7.051 19.974]]
```

- **Fitting** estimates μ and W from a chosen dataset.
- **Applying** uses those fixed values for other observations with the same feature definitions.
- Re-estimating the mean or directions creates a different fitted representation.

PCA relies on centering and **then** measuring variance. **What would happen if we computed variance directly, without centering first?**