Notes for Linear Algebra with Python and R

Justin Burruss

2026-02-20

Background

The code and notes below are for the kinds of linear algebra that come up in probability, statistics, and statistical learning. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Conventions

A common convention is to write matrices as capital Latin or Greek letters, e.g., M, Σ, then leave vectors and matrix elements in lowercase, e.g., m, σ. Often the letters will match, for instance if the vectors were named $a_1, a_2, ...$ then the matrix of those vectors would be A. There's less consistency when it comes to boldface conventions or whether to decorate vectors with arrows. Some authors clarify scalar vs. vector vs. matrix through a combination of case and font style: regular lowercase for scalars, bold lowercase for vectors, bold uppercase for matrices, e.g., scalar a, vector $\mathbf{a}$, and matrix $\mathbf{A}$.

The convention for matrix dimensions is to give them in row, column order: a matrix $M \in \mathbb{R}^{m\times n}$ has m rows and n columns. It's common for A_{ij} to refer to the element in the ith row and jth column of A; some sources will separate the row and column with a comma, e.g., $A_{i,j}$. Authors may use $A_{i,:}$ to refer to the elements in row i and $A_{:,j}$ to refer to the elements in column j. Often A_i will refer to either row vector i or column vector i (based on context).

Sources typically use either brackets or parentheses around a matrix, so we usually see something along the following lines.

$$A \in \mathbb{R}^{m\times n} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \qquad (m \text{ rows by } n \text{ columns})$$

Shape and Structure

A matrix with the same number of rows as columns ($m = n$) is *square*. Having more rows than columns ($m > n$) makes the matrix *tall* while having fewer rows than columns ($m < n$) makes the matrix *wide*.

A square matrix where the only non-zero elements are on the diagonal (i.e., where $i = j$) is a *diagonal matrix*.

$$\begin{bmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{bmatrix} \qquad (\text{a diagonal matrix } A)$$

The matrix is *tridiagonal* if there are non-zero elements only on the main diagonal and the diagonals immediate above and below.

I

Naming the identity matrix I is almost universal. $I \in \mathbb{R}^{n\times n}$ has ones along the diagonal, zeroes elsewhere.

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \qquad \text{(identity matrix } I\text{)}$$

Sometimes a subscript may be used to indicate the size, e.g., I_2 for the 2×2 identity matrix.

With NumPy, `np.eye()` gets you an identity matrix, just like `eye()` in MATLAB or Octave, whereas in R you would use `diag()`.

M^{-1}

It's common to use $^{-1}$ to indicate a matrix inverse with the use calling to mind reciprocals in real numbers. The inverse of a square matrix $M \in \mathbb{R}^{n\times n}$ is written as M^{-1} and is defined by the property that $MM^{-1} = M^{-1}M = I$. Not all square matrices are invertible, but if they are, then the inverse is unique.

In NumPy this is done with `np.linalg.inv`, in MATLAB or Octave with `inv()`.

Square matrices A and B are said to be *similar* if there exists another matrix S such that $B = S^{-1}AS$.

Rectangular or square matrices A and B are said to be *equivalent* if there exist matrices U and V such that $B = U^{-1}AV$.

M^P

Powers of a matrix are written using superscripts, e.g., $M^3 = MMM$. This implies the matrix is square, otherwise the dimensions would not line up.

Machine learning sources often use superscripts to indicate the layer number, in which case parentheses or brackets will also appear; for example the weight matrix for layer 2 might be written as $W^{(2)}$ or $W^{[2]}$.

M^T

Most sources use either ^{T} or $'$ (single quote) to indicate transpose. The transpose operation ^{T} switches rows with columns, so if $M \in \mathbb{R}^{m\times n}$ then $M^T \in \mathbb{R}^{n\times m}$. A second transpose puts M back, i.e., $(M^T)^T = M$. For matrix products, $(AB)^T = B^TA^T, (ABC)^T = C^TB^TA^T$, and so forth for products of more than three matrices. Transpose applies to vectors as well, so if x is a column vector then x^T is a row vector and vice versa.

In NumPy, `.T` returns a view of the array with the axes transposed. In R, `t()` returns the transpose. In Octave or MATLAB a single quote is used to transpose, e.g., `M'` is M^T.

If $M^T = M$, as can be the case with a square matrix, the matrix is said to be *symmetric*.

If $M \in \mathbb{R}^{n\times n}$ is such that $M^TM = I$ then the matrix is said to be *orthogonal*. A common convention is to write orthogonal matrices as Q. Thus, if $Q^TQ = I$, then we also have $Q^T = Q^{-1}$. The column vectors of an orthogonal matrix form an orthonormal set, meaning that for any column vectors q_i and q_j:

$$q_i^T q_j = \begin{cases} 0 \text{ if } i \neq j \\ 1 \text{ if } i = j \end{cases}$$

Likewise, the row vectors of an orthogonal matrix form an orthonormal set.

For $M \in \mathbb{R}^{m \times n}$ we're guaranteed that $M^T M$ is *positive semi-definite*, i.e., $x^T M x \geq 0$, $\forall x \in \mathbb{R}^n$.

det(M)

The determinant of matrix M is variously indicated as $\det M$, $\det(M)$ or $|M|$, but $|M|$ could also be indicating the absolute value of each entry of M. $|M_{ij}|$ on the other hand is less ambiguous and is likely indicating the element-wise absolute values.

R, Octave, MATLAB all have `det()` for finding the determinant. With NumPy it's `np.linalg.det()`.

μ_X, R_X, and C_X

Just as lowercase Greek Mu (μ) is often used to evoke the "m" in "mean", μ_X is often chosen to name the column vector having expected values of an $m \times n$ matrix X of random vectors $X_1 \dots X_n$:

$$E[X] = \mu_X = \begin{bmatrix} E[X_1] \\ E[X_2] \\ \vdots \\ E[X_n] \end{bmatrix}$$

The correlation of X—called the autocorrelation matrix, the correlation matrix, or the second moment matrix—is the symmetric matrix where each i, j element is the expected value of $X_i X_j$.

$$R_X = E[XX^T] = \begin{bmatrix} E[X_1X_1] & E[X_1X_2] & \cdots & E[X_1X_n] \\ E[X_2X_1] & E[X_2X_2] & \cdots & E[X_2X_n] \\ \vdots & \vdots & \ddots & \vdots \\ E[X_nX_1] & E[X_nX_2] & \cdots & E[X_nX_n] \end{bmatrix}$$

Here the choice of capital R can bring to mind the use of r or ρ (lowercase Greek Rho) for the Pearson correlation coefficient.

From μ_X and R_X we construct the covariance matrix C_X (sometimes named K_X).

$$C_X = E[(X - \mu_X)(X - \mu_X)^T] = R_X - \mu_X \mu_X^T$$

The choice of "C" or "K" can remind us of the hard C sound in "Covariance matrix" (in German, the literal K of *Kovarianzmatrix*). The matrix looks like this:

$$C_X = \begin{bmatrix} \text{Cov}[X_1X_1] & \text{Cov}[X_1X_2] & \cdots & \text{Cov}[X_1X_n] \\ \text{Cov}[X_2X_1] & \text{Cov}[X_2X_2] & \cdots & \text{Cov}[X_2X_n] \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}[X_nX_1] & \text{Cov}[X_nX_2] & \cdots & \text{Cov}[X_nX_n] \end{bmatrix} = \begin{bmatrix} \text{Var}[X_1] & \text{Cov}[X_1X_2] & \cdots & \text{Cov}[X_1X_n] \\ \text{Cov}[X_2X_1] & \text{Var}[X_2] & \cdots & \text{Cov}[X_2X_n] \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}[X_nX_1] & \text{Cov}[X_nX_2] & \cdots & \text{Var}[X_n] \end{bmatrix}$$

The covariance matrix is symmetric and positive semi-definite.

From using R_X for autocorrelation it's a natural extension to use R_{XY} for the cross-correlation between X and Y and C_{XY} for the cross-covariance between X and Y.

H_f

The *Hessian matrix* H_f (or sometimes just H) is the square matrix of the second partial derivatives of function f. If we're using subscript notation so that for example f_x is the partial derivative of f with respect to x and f_{xx} is the partial derivative of f_x with respect to x (the second partial derivative), then the Hessian looks like this:

$$H_f = \begin{bmatrix} f_{x_1x_1} & f_{x_1x_2} & \cdots & f_{x_1x_n} \\ f_{x_2x_1} & f_{x_2x_2} & \cdots & f_{x_2x_n} \\ \vdots & \vdots & \ddots & \vdots \\ f_{x_nx_1} & f_{x_nx_2} & \cdots & f_{x_nx_n} \end{bmatrix} \qquad (n \times n \text{ Hessian matrix})$$

Hessians are normally symmetric since—if they exist—we expect $f_{xy} = f_{yx}$ for the kinds of functions we come across in probability, statistics, and statistical learning.

J_f

The *Jacobian matrix* J_f (or sometimes just J) is the matrix of the first order partial derivatives of function f. It need not be square: f may be from $\mathbb{R}^n$ to $\mathbb{R}^m$, yielding an $m \times n$ matrix. It's tidier if we use "D notation" here so that for example $\partial_{x_1} f_1$ is the partial derivative of f_1 with respect to x_1, in which case our Jacobian looks like this:

$$J_f = \begin{bmatrix} \partial_{x_1} f_1 & \partial_{x_2} f_1 & \cdots & \partial_{x_n} f_1 \\ \partial_{x_1} f_2 & \partial_{x_2} f_2 & \cdots & \partial_{x_n} f_2 \\ \vdots & \vdots & \ddots & \vdots \\ \partial_{x_1} f_m & \partial_{x_2} f_m & \cdots & \partial_{x_n} f_m \end{bmatrix} \qquad (m \times n \text{ Jacobian matrix})$$

A small concrete example: let's say we're changing variables to swap between rectangular and polar coordinates ($f : \mathbb{R}^2 \to \mathbb{R}^2$), so we'll have a square Jacobian that we build by finding the partials for $x = r\cos\theta$ and $y = r\sin\theta$.

$$J = \begin{bmatrix} x_r & x_\theta \\ y_r & y_\theta \end{bmatrix} = \begin{bmatrix} \cos\theta & -r\sin\theta \\ \sin\theta & r\cos\theta \end{bmatrix}$$

Now we can quickly find the scaling factor for integrating over a region when swapping from rectangular to polar by taking the absolute value of the determinant, just like we would use the absolute value of the determinant to find the scaling factor of any linear map $A \in \mathbb{R}^{2\times 2}$. We expect a factor of r to get us from $\iint_R f(x,y)dA$ to $\iint_R f(r\cos\theta, r\sin\theta) r dr d\theta$, and that's exactly what we see:

$$\begin{aligned} \det(J) &= r\cos^2\theta + r\sin^2\theta && \text{determinant of a } 2\times 2 \text{ is } ad - bc \\ &= r(\cos^2\theta + \sin^2\theta) && \text{factor out the } r \\ &= r(1) && \cos^2\theta + \sin^2\theta = 1 \\ &= r \end{aligned}$$

Radius r is always nonnegative, so no need to use $|r|$ in this case, but in general we would need to remember to use absolute value to scale the areas.

Quadratic Form

A polynomial having all terms of degree two is in quadratic form. To illustrate: $2x_1^2 - 3x_1x_2 + 5x_2^2$ is in quadratic form because every term is of degree two. These can be expressed with column vector $x \in \mathbb{R}^n$ and a symmetric matrix $A \in \mathbb{R}^{n\times n}$ as:

$$x^T A x$$

For example, $2x_1^2 - 3x_1x_2 + 5x_2^2$ just needs vector $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and matrix $A = \begin{bmatrix} 2 & -\frac{3}{2} \\ -\frac{3}{2} & 5 \end{bmatrix}$:

$$\begin{aligned} x^T A x &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 2 & -\frac{3}{2} \\ -\frac{3}{2} & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= x_1(2x_1 - \frac{3}{2}x_2) + x_2(-\frac{3}{2}x_1 + 5x_2) \\ &= 2x_1^2 - \frac{3}{2}x_1x_2 - \frac{3}{2}x_1x_2 + 5x_2^2 \\ &= 2x_1^2 - 3x_1x_2 + 5x_2^2 \end{aligned}$$

If A is a diagonal matrix that we get no cross-product terms. For instance, if $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ then $x^T A x = x_1^2 + x_2^2$.

Decompositions

LU Decomposition

The factorization of a square matrix $A \in \mathbb{R}^{n \times n}$ into lower triangular matrix L and upper triangular matrix U is called LU Decomposition. This is the one we get by doing row reduction. If we needed to swap any rows during row reduction, then the factorization requires a permutation matrix P, and we have:

$$A = PLU$$

For linear algebra tasks, programming languages typically call LAPACK behind the scenes. Although NumPy uses LAPACK, and LAPACK certainly has LU Decomposition, NumPy does *not* have a function for LU Decomposition. However, SciPy (which also uses LAPACK) does include a function for LU decomposition. In R we could use the `Matrix` library for an LU Decomposition.

Here's an example using R. The factorization is just a one-liner, but it's extra work to print the results in a nicely-formatted matrix with fractions instead of decimals.

```r
library(MASS)
library(Matrix)

# create a 3x3 matrix
A<-matrix(c(1, 2, 3, 5, 5, 5, 0, 0, -7), ncol=3)
m<-nrow(A)
n<-ncol(A)

# perform LU decomposition
A.lu<-expand(lu(A))

# fractions from the MASS library formats as fractions so we see 1/3, 2/3, etc.
P<-fractions(matrix(as.numeric(A.lu$P), ncol=n))
L<-fractions(matrix(A.lu$L, ncol=n))
U<-fractions(matrix(A.lu$U, ncol=n))

# helper function to build LaTeX matrix rows
mat_to_latex<-function(M) {
  M_str<-apply(M, 1, function(x) {
    paste(
      paste(x, collapse = "&"),
      "\\\\"
      )
  })
  return(M_str)
}

# print our A = PLU; collect the LaTeX code into a block, otherwise between
# writeLines and RMarkdown the codes can be escaped inconsistently.
begin<-"\\begin{bmatrix}"
end<-"\\end{bmatrix}"
block<-c("$$",
         begin, mat_to_latex(A), end,
         " = ",
         begin, mat_to_latex(P), end,
         begin, mat_to_latex(as.character(L)), end,
         begin, mat_to_latex(as.character(U)), end,
         "$$")
writeLines(block)
```

$$\begin{bmatrix}1&5&0\\2&5&0\\3&5&-7\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix}\begin{bmatrix}1&0&0\\1/3&1&0\\2/3&1/2&1\end{bmatrix}\begin{bmatrix}3&5&-7\\0&10/3&7/3\\0&0&7/2\end{bmatrix}$$

Now let's see if PLU really gets us back to A. In R the matrix multiplication operator is `%*%`.

```
A_recov<-P %*% L %*% U
block<-c("$$", "A_{recov} = ", begin, mat_to_latex(A_recov), end, "$$")
writeLines(block)
```

$$A_{recov} = \begin{bmatrix}1&5&0\\2&5&0\\3&5&-7\end{bmatrix}$$

It works.

Let's also try it in Python with SciPy. Now our matrix multiplication operator is `@`. We will check that PLU really gets us back to A using `np.allclose`.

```
import numpy as np
from scipy.linalg import lu
import matplotlib.pyplot as plt

def check(statement_true, lhs, rhs):
    """Print TeX with = or != depending on whether statement is true"""
    relation = "=" if statement_true else r"\ne"
    print(rf"\[ {lhs} {relation} {rhs} \]")

A = np.array([[1, 2, 3],
              [5, 5, 5],
              [0, 0, -7]])
```

```
# perform LU decomposition
P, L, U = lu(A)

# Check: does P @ L @ U get us back to A?
check(np.allclose(A, P @ L @ U), "A", "PLU")
```

$$A = PLU$$

We recovered A from PLU just fine.

QR Decomposition

The factorization of matrix $A \in \mathbb{R}^{n\times n}$ into orthogonal matrix Q and upper triangular matrix R is called QR Decomposition.

A direct way to build QR from A is with what's called the Gram-Schmidt process. We start with the first vector as-is, then for the second vector we subtract off the projection of the first vector onto the second vector, then for the third vector we subtract off the projections of the first and second vectors onto that vector, and so forth. We also normalize the lengths of the vectors so our final result is not just orthogonal but orthonormal.

When we use Gram-Schmidt to construct $A = QR$, we always get an upper-triangular R because of how—starting with the second vector—we subtract from the prior vector the redundant components (the projections of our new vector onto the prior vectors). Let's say we start with matrix A having linearly independent column vectors, then what we're creating looks like this (for simplicity square in this illustration, but it doesn't have to be):

$$A = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \dots & a_n \\ | & | & & | \end{bmatrix} = QR = \begin{bmatrix} | & | & & | \\ q_1 & q_2 & \dots & q_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} a_1^T q_1 & a_2^T q_1 & \dots & a_n^T q_1 \\ a_1^T q_2 & a_2^T q_2 & \dots & a_n^T q_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_1^T q_n & a_2^T q_n & \dots & a_n^T q_n \end{bmatrix}$$

When carrying out Gram-Schmidt we first chose q_1 to be a unit vector in the direction of a_1, then when we got to q_2 we took care to ensure it was perpendicular to a_1 by subtracting out the redundant portion (the projection of our new candidate onto the prior). If it's perpendicular, then $a_1^T q_2$ must be zero by the property of perpendicular vectors. And it's not just $a_1^T q_2$. For any q_i having $i > 1$ we made sure when following the procedure to chose a unit vector perpendicular to a_1. It's perpendicular turtles all the way down: we always chose q_n to be perpendicular to all the previous $a_{n-1}, a_{n-2}, \dots, a_1$ by subtracting out those projections when executing Gram-Schmidt. Thus, when we fill in the R matrix we get zeroes for every element below the diagonal, i.e., for R_{ij} where $i > j$.

$$\begin{bmatrix} a_1^T q_1 & a_2^T q_1 & \dots & a_n^T q_1 \\ a_1^T q_2 & a_2^T q_2 & \dots & a_n^T q_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_1^T q_n & a_2^T q_n & \dots & a_n^T q_n \end{bmatrix} = \begin{bmatrix} a_1^T q_1 & a_2^T q_1 & \dots & a_n^T q_1 \\ 0 & a_2^T q_2 & \dots & a_n^T q_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_n^T q_n \end{bmatrix}$$

With Python + NumPy we can accomplish this using `np.linalg.qr`. The "classical" Gram-Schmidt algorithm is not numerically stable, so `np.linalg.qr` (and other interfaces to LAPACK) use other algorithms.

```
#
# QR Decomposition
#

# Create a sample matrix
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

# Perform QR decomposition
# Q is an orthonormal matrix
# R is an upper-triangular matrix
Q, R = np.linalg.qr(A)

# Check: does Q @ R get us back to A?
check(np.allclose(A, Q @ R), "A", "QR")
```

$$A = QR$$

With `np.allclose` we've confirmed that the elements of `A` and `Q @ R` each tie within a small tolerance. Let's also check whether the Q from `np.linalg.qr` is really an orthogonal matrix (i.e., a matrix of orthonormal vectors).

```
# Check: is Q really an orthogonal matrix?
check(np.allclose(Q.T @ Q, np.eye(Q.shape[0])), "Q^T Q", "I")
```

$$Q^T Q = I$$

Everything ties.

Eigendecomposition

For $A \in \mathbb{R}^{n \times n}$, provided we have n linearly independent eigenvectors, we can decompose into a diagonal matrix Λ (capital Lambda) with eigenvalues along the diagonal and Q with eigenvectors as columns as:

$$A = Q \Lambda Q^{-1}$$

We can do this in two steps with NumPy. First we call `np.linalg.eig` to get the eigenvalues and right eigenvectors.

```
#
# Eigendecomposition
#

lmbda, v = np.linalg.eig(A)

# Check: is Av about the same as λv?
check(np.allclose(A @ v, lmbda * v), "Av", "λv")
```

$$Av = \lambda v$$

Then we construct Λ using `np.diag`. We can check that $A = Q\Lambda Q^{-1}$ using `np.allclose`.

```
Q = v
Q_inv = np.linalg.inv(Q)
Lambda = np.diag(lmbda)

# Check: does Q @ Λ @ Q^{-1} get us back to A?
check(np.allclose(A, Q @ Lambda @ Q_inv), "A", "Q \Lambda Q^{-1}")
```

$$A = Q\Lambda Q^{-1}$$

This kind of decomposition comes up with Markov chains. For instance, if we have a two-state Markov chain with transition probabilities given by $p, q, p-1, q-1$, we might build a transition matrix as follows:

$$P \in \mathbb{R}^{2\times 2} = \begin{bmatrix} 1-p & p \\ q & 1-q \end{bmatrix}$$

The eigenvalues are $\lambda_1 = 1$ and $\lambda_2 = 1-(p+q)$, and we can factor P as:

$$P = S^{-1}DS = \begin{bmatrix} 1 & \frac{-p}{p+q} \\ 1 & \frac{q}{p+q} \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} \frac{q}{p+q} & \frac{p}{p+q} \\ -1 & 1 \end{bmatrix}$$

The name D in $S^{-1}DS$ evokes "diagonal" while the S is for "similarity".

Left Inverses and Pseudoinverses

Only square matrices have inverses, but if we have full column rank (as we would with a typical tall matrix having many rows for a few columns) then we can get a *left inverse*.

Say we have a tall matrix $A \in \mathbb{R}^{m\times n}$, $m > n$. Then A^TA is an invertible $n \times n$ symmetric matrix, so we get $(A^TA)^{-1}A^TA = I$, and thus a left inverse:

$$A^{-1}_{left} = (A^TA)^{-1}A^T$$

This is often called a *pseudoinverse* of the matrix or the *Moore-Penrose inverse*.

```
#
# Pseudo-inverse for a non-square matrix
#

rng = np.random.default_rng(2026)

# Create tall matrix "T" of 40 rows and 4 columns: (40, 4), will have full column rank
T = rng.integers(-10, 11, size=(40, 4))

# T.T (4, 40) @ T (40, 4) has shape (4, 4), just as with matrix multiplication
# This matrix will be symmetric
TtT = T.T @ T

# (T.T @ T)^{-1} still has shape (4, 4)
TtT_inv = np.linalg.inv(TtT)
```

```
# L (4, 4) @ T.T (4, 40) has shape (4, 40)
# T has full column rank, so (T.T @ T)^{-1} @ T.T is the left inverse of T
# and equals the Moore-Penrose pseudoinverse
L = TtT_inv @ T.T

# I (4, 40) @ T (40, 4) has shape (4, 4)
# Check that L really acts as a left inverse on T.
I = L @ T
check(np.allclose(np.eye(I.shape[0]), I), "LT", "I")
```

$$LT = I$$

This can also work if we have full row rank, for example with some wide matrix, in which case we could construct a *right inverse* so that $AA_{right}^{-1} = I$.

NumPy has a built-in pseudoinverse function `np.linalg.pinv`. Let's see if NumPy yields the same pseudoinverse for our tall matrix.

```
# Check: is the NumPy built-in pseudo-inverse close to the manual left inverse?
check(np.allclose(np.linalg.pinv(T), L), "L_{MP}", "L_{pinv}")
```

$$L_{MP} = L_{pinv}$$

They tie.

Some sources use † (superscript dagger) to indicate a pseudoinverse, e.g., $M^\dagger$ is the pseudoinverse of M.

Singular Value Decomposition (SVD)

SVD is given by

$$A = U\Sigma V^T$$

where U and V are orthogonal matrices and Σ (capital Sigma) is matrix having *singular values* along the diagonal and zeroes elsewhere. Recall that with $A = Q\Lambda Q^{-1}$ we needed A to be square, whereas with $A = U\Sigma V^T$ we can have a non-square A.

NumPy has `np.linalg.svd` for doing an SVD. By default, if you call it on some $A \in \mathbb{R}^{m\times n}$ you get back:

- orthogonal matrix $U \in \mathbb{R}^{m\times m}$ as a shape `(m, m)` array
- singular values in a 1D array having `min(m, n)` elements
- orthogonal matrix $V^T \in \mathbb{R}^{n\times n}$ as a shape `(n, n)` array

Note that the singular values are *not* returned as an $m \times n$ matrix, so we can't just do `U, S, Vt = np.linalg.svd(A)` and immediately try `U @ S @ Vt`. One approach we might use is to form a diagonal $\min(n, m) \times \min(n, m)$ matrix from the array of singular values and use slicing to do a rank-aware reconstruction.

```
#
# Singular Value Decomposition (SVD)
#
```

```python
# Our sample matrix A will be of shape (3, 4)
A = np.array([[1, 2, 3, 2],
              [4, 5, 6, 4],
              [7, 8, 9, 9]])

# Singular Value Decomposition gives us U Sigma V:
# 1. mxm array U is for our orthogonal mxm matrix
# 2. array of singular values "s" [use np.diag(s) to build a Sigma matrix]
# 3. nxn array Vt is for our orthogonal nxn matrix [V is already transposed]
m, n = A.shape
U, s, Vt = np.linalg.svd(A)

# Check: A = U Sigma V.T
check(np.allclose(A, U[:, :n] @ np.diag(s) @ Vt[:m, :]), "A", "U \Sigma V^T")
```

$$A = U\Sigma V^T$$

We can also supply `np.linalg.svd` with `full_matrices=False` so that our V^T comes back as shape `(m, min(m, n))` and we can use `U @ np.diag(s) @ Vt` directly.

```python
# Alternatively: call with full_matrices=False and we get back:
# 1. mxm array U (no change)
# 2. array s (no change)
# 3. m x min(m, n) array V
# (then we have(m x m) @ (m x m) @ (m x min(m, n)))
Vt_full = Vt.copy()
U, s, Vt = np.linalg.svd(A, full_matrices=False)

# Check: does Q @ R get us back to A?
check(np.allclose(A, U @ np.diag(s) @ Vt), "A", "U \Sigma V^T")
```

$$A = U\Sigma V^T$$

We can tie the singular values and singular vectors from SVD back to the eigenvalues and eigenvectors of Eigendecomposition even though our matrix A is not square. Say $A \in \mathbb{R}^{m\times n}$, then We can use $A^T A$ to guarantee a square $n \times n$ matrix since $(n \times m)(m \times n)$ yields an $(n \times n)$ result.

With a little algebra we find:

$$\begin{aligned}
A^T A &= (U\Sigma V^T)^T (U\Sigma V^T) && \text{plug in definition of SVD} \\
&= (V^T)^T \Sigma^T U^T (U\Sigma V^T) && (ABC)^T = C^T B^T A^T \\
&= V\Sigma^T U^T U \Sigma V^T && (V^T)^T = V \\
&= V\Sigma^T I \Sigma V^T && Q^T Q = QQ^T = I \text{ for orthogonal } Q \\
&= V\Sigma^T \Sigma V^T && \\
&= V\Sigma^T \Sigma V^{-1} && Q^T = Q^{-1} \text{ for orthogonal } Q
\end{aligned}$$

Take a look at $\Sigma^T\Sigma$. The matrix Σ was constructed to have singular values along its diagonal and zeroes elsewhere, so $\Sigma^T\Sigma$ will be another diagonal matrix with squares of the singular values: $\sigma_1^2, \sigma_2^2, ...,$ etc. Thus if we let $S = \Sigma^T\Sigma$ we have:

$$\begin{aligned}
A^T A &= V\Sigma^T \Sigma V^{-1} && \\
&= VSV^{-1} && \text{let } S = \Sigma^T\Sigma
\end{aligned}$$

This arrangement of orthogonal matrix times diagonal matrix times the inverse of the orthogonal matrix is just like our form for an Eigendecomposition:

$$A = Q\Lambda Q^{-1}$$

Upshot is:

- the nonzero singular values in $\Sigma^T\Sigma$ are same as the square roots of the nonzero eigenvalues in Λ
- the nonzero singular vectors in V are the same as the eigenvectors in Q after allowing for arbitrary sign differences

Let's test that out in Python. We do need to make sure to select only the *non-zero* eigenvalues & singular values, hence the `~np.isclose(AtA_eigvals, 0)` (`~` is the bitwise NOT operator in Python).

```
#
# SVD back to Eigendecomposition
#

AtA = A.T @ A
AtA_eigvals, AtA_eigvecs = np.linalg.eig(AtA)

# do the nonzero eigenvalues tie with the singular values? we should see
# singular values = sqrt( eigenvalues )
check(np.allclose(s, np.sqrt(AtA_eigvals[~np.isclose(AtA_eigvals, 0)])),
      "s_i", "\sqrt{\lambda_i}")
```

$$s_i = \sqrt{\lambda_i}$$

The eigenvalues and singular values tie perfectly. What about the eigenvectors and singular vectors? When comparing we can wrap them in `np.abs()` to ignore arbitrary sign differences.

```
# will the V and Lambda vectors match after ignoring any sign flips?
Lmda = AtA_eigvecs[:, ~np.isclose(AtA_eigvals, 0)]
V = Vt.T
check(np.allclose(np.abs(V), np.abs(Lmda)), "|V_{ij}|", "| \Lambda_{ij} |")
```

$$|V_{ij}| = |\Lambda_{ij}|$$

Everything ties.

Visualizing SVD

Let's view a 2D SVD. First, we define two 2D shapes and run the SVD. Let's also check that the transformations recover the original matrix.

```
# two shapes
A = np.array([
    [0.0,  0.5,  0.0, -0.9],
    [0.0,  1.0,  1.0,  0.0]
])
```

```
A = np.hstack((A, A[:, [0]]))
B = np.array([
    [0.0,  -0.9,  0.0,  0.5],
    [0.0,   0.0, -1.8, -1.6]
])
B = np.hstack((B, B[:, [0]]))

# matrix M will transform the polygons A and B
M = np.array([[3, 1],
              [1, 2]])

# SVD on a square matrix, no need for full_matrices=False
U, s, Vt = np.linalg.svd(M)
Sigma = np.diag(s)

# quick check: do the transformations take us back?
check(np.allclose(M @ A, U @ Sigma @ Vt @ A), "MA", "U \Sigma V^T A")
```

$$MA = U\Sigma V^T A$$

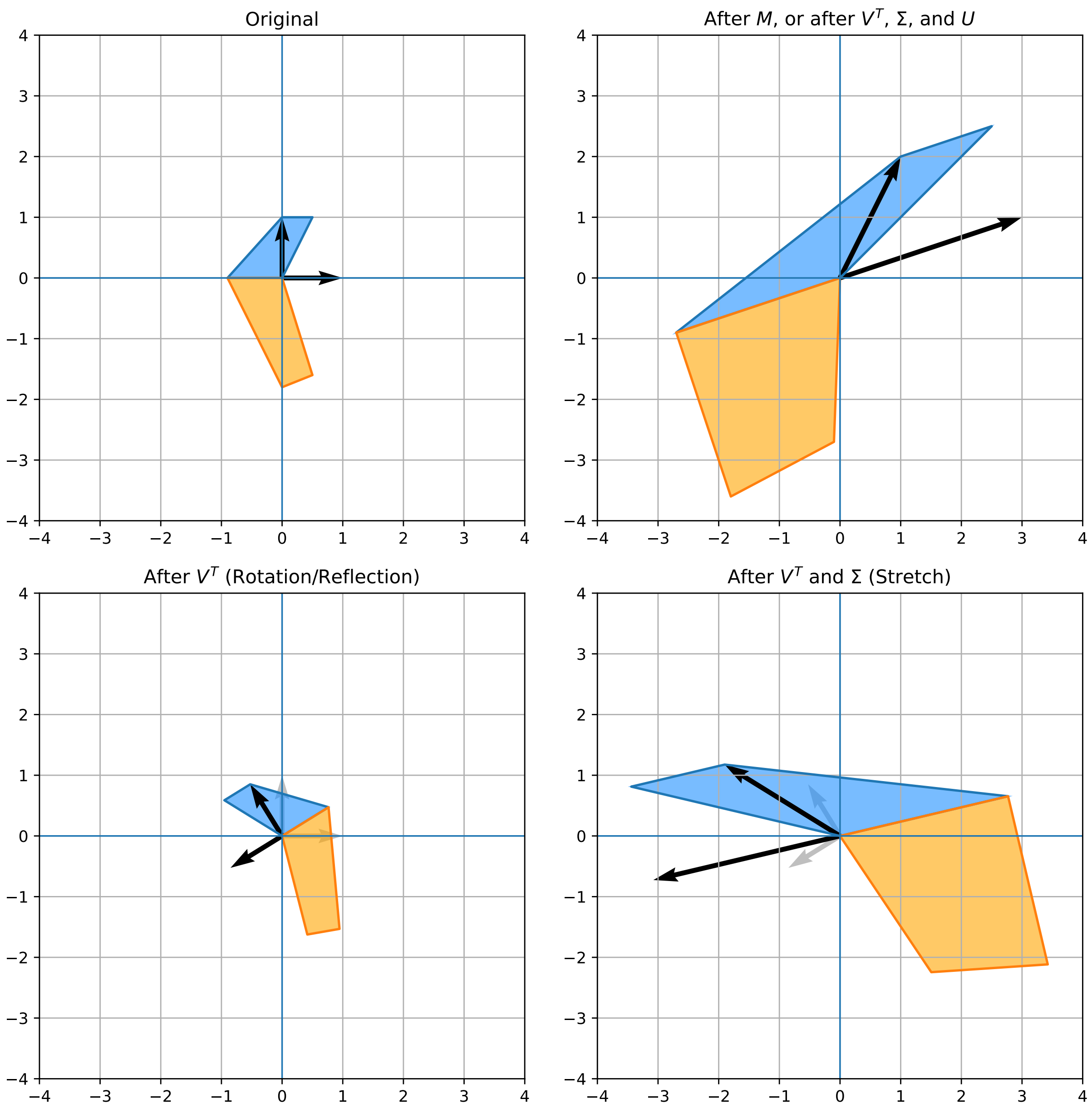
```
check(np.allclose(M @ B, U @ Sigma @ Vt @ B), "MB", "U \Sigma V^T B")
```

$$MB = U\Sigma V^T B$$

They tie. Now let's plot.

```
# set up four subplots
fig, axs = plt.subplots(2, 2, figsize=(10, 10))

def setup(ax, title):
    ax.axhline(0, linewidth=1)
    ax.axvline(0, linewidth=1)
    ax.set_aspect('equal', 'box')
    ax.set_xlim(-4, 4)
    ax.set_ylim(-4, 4)
    ax.set_title(title)
    ax.grid(True)

def draw_shape(ax, pts, fill="black"):
    ax.fill(pts[0], pts[1], alpha=0.6, color=fill)
    ax.plot(pts[0], pts[1])

def draw_vectors(ax, vectors, color, alpha=1.0):
    for i in range(2):
        x, y = vectors[:, i]
        ax.quiver(0, 0, x, y,
                  angles='xy', scale_units='xy', scale=1,
                  color=color, width=0.01, alpha=alpha)

# starting shapes
draw_shape(axs[0, 0], A, fill="dodgerblue")
```

```python
draw_shape(axs[0, 0], B, fill="orange")
draw_vectors(axs[0, 0], np.eye(2), 'black')
setup(axs[0, 0], "Original")

# after Vt (rotation/reflection)
draw_vectors(axs[1, 0], np.eye(2), 'gray', alpha=0.5)
draw_shape(axs[1, 0], Vt @ A, fill="dodgerblue")
draw_shape(axs[1, 0], Vt @ B, fill="orange")
draw_vectors(axs[1, 0], Vt.T, 'black')
setup(axs[1, 0], "After $V^T$ (Rotation/Reflection)")

# after Sigma (stretch)
draw_vectors(axs[1, 1], Vt.T, 'gray', alpha=0.5)
draw_shape(axs[1, 1], Sigma @ Vt @ A, fill="dodgerblue")
draw_shape(axs[1, 1], Sigma @ Vt @ B, fill="orange")
draw_vectors(axs[1, 1], Sigma @ Vt.T, 'black')
setup(axs[1, 1], "After $V^T$ and $\\Sigma$ (Stretch)")

# after U (gets us back to the ellipse from M @ circle)
draw_shape(axs[0, 1], M @ A, fill="dodgerblue")
draw_shape(axs[0, 1], M @ B, fill="orange")
draw_vectors(axs[0, 1], M, 'black')
setup(axs[0, 1], "After $M$, or after $V^T$, $\\Sigma$, and $U$")

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



Linear Regression

What we call linear regression is actually most often affine since we have an intercept term:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$$

Say we represent the weights $\beta_1 \ldots \beta_n$ as a vector w. We're still left with that bias term β_0:

$$\hat{y} = \beta_0 + w^T x$$

We can tidy this up if we add a column of ones to our x, say as either the first or the last column. Now we have a linear formula:

$$\hat{y} = w^T x$$

In Python we might use `np.column_stack` and `np.ones` as in this example.

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

# read CSV
c = pd.read_csv(r"E:\docs\Classes\ISL\Advertising.csv") \
    .rename(columns={"Unnamed: 0": "Id"}) \
    .set_index('Id')

# for Ax = b: sales is our dependent variable b
b = c['sales'].values

# for Ax = b: A will be our one independent var plus a column of ones for the bias
A = np.column_stack((c['TV'].values, np.ones(len(c['TV'].values))))

# linear regression to fit \hat{b} = Ax
x, RSS, rank, S = np.linalg.lstsq(A, b, rcond=None)
```

Least Squares

Let's say we have some "tall" training data set. If A is our tall input matrix of independent variables and b is our target vector (the dependent variable), we might do a least squares linear regression with `x, RSS, rank, S = np.linalg.lstsq(A, b, rcond=None)` to find weights x for $Ax = b$ along with the residual sum of squares (RSS), the rank of the matrix, and any singular values.

If we take a step back and look at this from a linear algebra perspective we'll find our friend the left inverse. Our sum of squared errors is $\text{RSS}(x) = (b - Ax)^2 = (b - Ax)^T(b - Ax) = b^Tb - 2b^TAx + x^TA^TAx$. How would we minimize that function? It's a quadratic, so the minimum is going to be where the gradient is zero. If we use ∇_x to indicate the partial derivative with respect to our weight vector x we get:

$$\begin{aligned}\nabla_x &= 0 - 2A^Tb + 2A^TAx \\ &= 2A^TAx - 2A^Tb\end{aligned}$$

Set it to zero and use a little algebra with our left inverse move to solve.

$$\begin{aligned}0 &= 2A^TAx - 2A^Tb && \text{(set to zero)} \\ 2A^Tb &= 2A^TAx && \text{(add } 2A^Tb \text{ to both sides)} \\ \frac{1}{2}2A^Tb &= \frac{1}{2}2A^TAx \\ A^Tb &= A^TAx \\ (A^TA)^{-1}A^Tb &= (A^TA)^{-1}A^TAx && \text{(multiply both sides by left inverse of } A^TA\text{)} \\ (A^TA)^{-1}A^Tb &= x\end{aligned}$$

So in Python, assuming the columns of our tall matrix are really linearly independent, we could set our weights x to $(A^TA)^{-1}A^Tb$ using `np.linalg.inv(A.T @ A) @ A.T @ b` in lieu of `np.linalg.lstsq()`.

Let's check.

```python
# create three independent variables and random noise
rng = np.random.default_rng(2026)
X = rng.uniform(-1, 1, (100,3)).round(3)
```

```
noise = rng.normal(0, 0.1, (100, 3))

# set up our Ax = b
b = (X + noise) @ [1, 2, 3] + 4
A = np.column_stack([X, np.ones(X.shape[0])])

# left inverse
A_left_inv = np.linalg.inv(A.T @ A) @ A.T @ b

# least squares
lsq = np.linalg.lstsq(A, b, rcond=None)[0]

# check: will left inverse give same result as least np.linalg.lstsq?
check(np.allclose(lsq, A_left_inv), "(A^T A)^{-1} A^T b", "\\text{Least Squares Solution}")
```

$$(A^T A)^{-1} A^T b = \text{Least Squares Solution}$$

The Python results tie.

This was with linearly independent columns. If for some reason not all columns were linearly independent—well really we ought to revisit our columns in that case, but *if* we wanted we could use `np.linalg.pinv(A.T @ A) @ A.T @ b` to get the same result as with `np.linalg.lstsq(A, b, rcond=None)[0]`. Let's break linear independence and see what kind of error we get. In Python, exception handling is via `try/except`. We'll use a new matrix `B`.

```
# try without linear independence
B = A.copy()
B[:, 1] = 2 * B[:, 0]
try:
    np.linalg.inv(B.T @ B) @ B.T @ b
except np.linalg.LinAlgError as e:
    print(e)
```

```
## Singular matrix
```

By setting the second column of `B` equal to exactly twice the first column we introduced linear dependency, hence the `Singular matrix` error message. We could have checked for that with `np.linalg.matrix_rank()` and seen that the rank was only 3 versus the 4 in the original `A`.

```
check(np.allclose(np.linalg.matrix_rank(A), 4), "\\text{rank}(A)", "4")
```

$$\text{rank}(A) = 4$$

```
check(np.allclose(np.linalg.matrix_rank(B), 4), "\\text{rank}(B)", "4")
```

$$\text{rank}(B) \neq 4$$

```
check(np.allclose(np.linalg.matrix_rank(B), 3), "\\text{rank}(B)", "3")
```

$$\mathrm{rank}(B) = 3$$

Now to check that `np.linalg.pinv(B.T @ B) @ B.T @ b` really does work.

```python
# check: will pseudoinverse give same result as least np.linalg.lstsq?
B_pseudo = np.linalg.pinv(B.T @ B) @ B.T @ b
lsq = np.linalg.lstsq(B, b, rcond=None)[0]
check(np.allclose(lsq, B_pseudo), "(B^T B)^{\dagger} B^T b", "\\text{Least Squares Solution}")
```

$$(B^T B)^{\dagger} B^T b = \text{Least Squares Solution}$$

It works.

We're giving the computer more work if we swap out `np.linalg.lstsq()` for three `@`s and either `np.linalg.inv()` or `np.linalg.pinv()`, so we would probably never run it this way in practice, but it's nice to see the connection.

Principal Component Analysis (PCA)

We can try out PCA on the Olivetti faces dataset. Each face is 112 * 92 = 10,304 pixels (i.e., 10,304 dimensions). We'll use PCA to see how well we can reconstruct using just 10 or 100 principal components. We could use eigendeomposition or SVD. Let's try SVD.

First we load the data file.

```python
# load the Olivetti face dataset (400 faces, 112 pixels high by 92 pixes wide)
import scipy.io
X = scipy.io.loadmat(r'E:\docs\Classes\CSE 41287\pca\facesOlivetti')["X"]
h = 112
w = 92
```

Let's find the mean face, and save a recentered `X`.

```python
# find the mean face and save recentered pixels
mean_face = np.mean(X, axis=0)
X_r = X - mean_face
```

SVD is a one-liner.

```python
# SVD
U, s, Vt = np.linalg.svd(X_r, full_matrices=False)
```

Finally we construct the projection and reconstruct images using 0, 10, and 100 principal components.

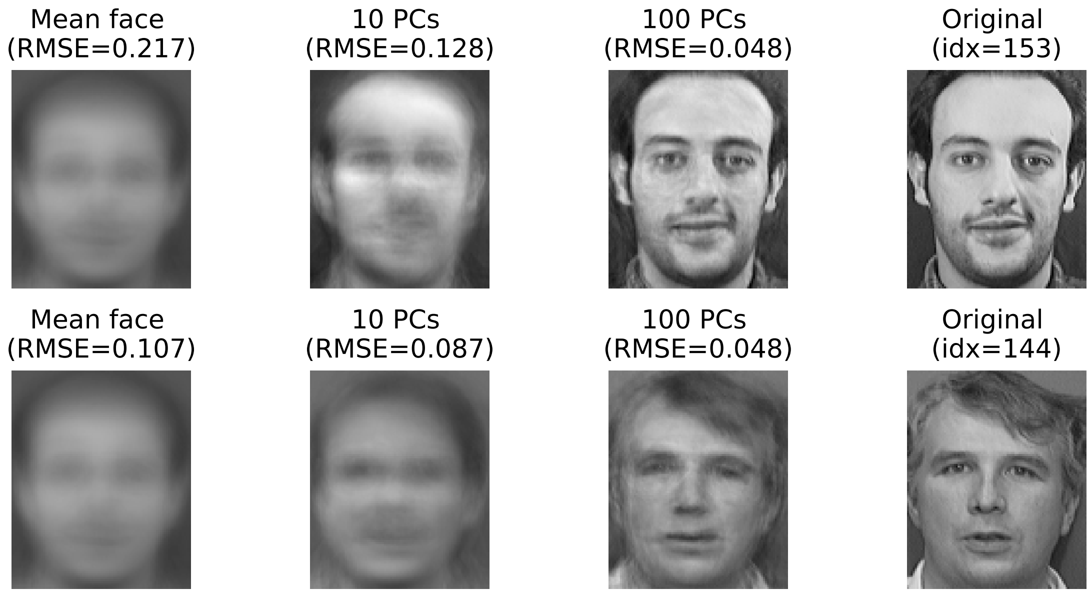
```python
# find the projection
X_proj = X_r @ Vt.T

def reconstruct(k: int = 1):
    """Reconstruct X from X_proj and Vt"""
    X_rec = (X_proj[:, :k] @ Vt[:k, :]) + mean_face
    mse_per = np.mean((X - X_rec)**2, axis=1)
    rmse_per = np.sqrt(mse_per)
    return rmse_per, X_rec

# try it with mean face only, 10 PCs, and 100 PCs
rmse0, Xrec0 = reconstruct(0)
rmse10, Xrec10 = reconstruct(10)
rmse100, Xrec100 = reconstruct(100)
```

Let's visualize the face most similar to the mean and the face least similar to the mean. We can use `np.argmin()` and `np.argmax()` to look them up.

```
# use the most mean and least mean sample faces for visualizing
idx_mm = np.argmax(rmse0)
idx_lm = np.argmin(rmse0)
indices = [idx_mm, idx_lm]
```

Plotting the faces and reconstructions, it's interesting that after 100 PCs both the most mean and least mean faces have converged on the same reconstruction error. It's also interesting how much the mean face looks like a blurry face.

```
# plot with 2 rows of four subplots: mean face, 10 PCs, 100 PCs, orig
fig, axes = plt.subplots(2, 4, figsize=(8, 4))

for i, idx in enumerate(indices):

    # mean face
    axes[i, 0].imshow(mean_face.reshape(w, h).T, cmap='gray', vmin=0, vmax=1)
    axes[i, 0].set_title(f"Mean face \n(RMSE={rmse0[idx]:.3f})");
    axes[i, 0].axis('off');

    # 10 PCs
    axes[i, 1].imshow(Xrec10[idx, :].reshape(w, h).T, cmap='gray', vmin=0, vmax=1)
    axes[i, 1].set_title(f"10 PCs \n(RMSE={rmse10[idx]:.3f})");
    axes[i, 1].axis('off');

    # 100 PCs
    axes[i, 2].imshow(Xrec100[idx, :].reshape(w, h).T, cmap='gray', vmin=0, vmax=1)
    axes[i, 2].set_title(f"100 PCs \n(RMSE={rmse100[idx]:.3f})");
    axes[i, 2].axis('off');

    # original
    axes[i, 3].imshow(X[idx, :].reshape(w, h).T, cmap='gray', vmin=0, vmax=1)
    axes[i, 3].set_title(f"Original \n(idx={idx})");
    axes[i, 3].axis('off');

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


Mean face (RMSE=0.217)
10 PCs (RMSE=0.128)
100 PCs (RMSE=0.048)
Original (idx=153)
Mean face (RMSE=0.107)
10 PCs (RMSE=0.087)
100 PCs (RMSE=0.048)
Original (idx=144)