Brief Review of Linear Algebra with NumPy

David I. Inouye

NumPy is the core numerical library in Python

- We will review linear algebra alongside NumPy implementations to help translate from math to code and vice versa.
- Acknowledgement: Some content adapted from: http://www.deeplearningbook.org/contents/linear_algebra.html

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

Scalars

- Single number
- Denoted as lowercase letter
- Examples
  - $x \in \mathbb{R}$ - Real number
  - $y \in \{0, 1, \dots, C\}$ - Finite set
  - $u \in [0, 1]$ - Bounded set

```
1 x = 1.1343
2 print(x)
3 z = int(-5)
4 print(z)
```

```
1.1343
-5
```

Vectors

- An array of numbers/scalars
- In notation, we usually consider vectors to be "column vectors"
- Denoted as lowercase letter (often bolded)
- Dimension is often denoted by d or D.
- Access elements via subscript, e.g., x_i is the i-th element

Vectors

- Examples
  - $\mathbf{x} \in \mathbb{R}^d$
  - $\mathbf{x} = [x_1, x_2, \dots, x_d]^T$
  - $\mathbf{z} = [\sqrt{x_1}, \sqrt{x_2}, \dots, \sqrt{x_d}]^T$
  - $\mathbf{y} \in \{0, 1, \dots, C\}^d$ - Finite set
  - $\mathbf{u} \in [0, 1]^d$ - Bounded set

```
x = np.array([1.1343, 6.2345, 35])
print(x)
z = 5 * np.ones(3, dtype=int)
print(z)
```

```
[ 1.1343  6.2345 35.    ]
[5 5 5]
```

Note: The operator + does different things on numpy arrays vs Python lists

- For lists, Python concatenates the lists
- For numpy arrays, numpy performs an element-wise addition
- Similarly, for other binary operators such as -, +, *, and /

```
1 a_list = [1, 2]
2 b_list = [30, 40]
3 c_list = a_list + b_list
4 print(c_list)
5 a = np.array(a_list)  # Create numpy array from Python list
6 b = np.array(b_list)
7 c = a + b
8 print(c)
```

```
[1, 2, 30, 40]
[31 42]
```

```
1 type(a_list)
```

```
list
```

```
1 type(a)
```

```
numpy.ndarray
```

Matrices

- 2D array of numbers
- Denoted as uppercase letter
- Number of samples often denoted by n or N.
- Access rows or columns via subscript or numpy notation:
  - $X_{i,:}$ is the i-th row, $X_{:,j}$ is the jth column
  - (Sometimes) X_i, $\mathbf{x}_i$ is the i-th row or column depending on context
- Access elements by double subscript $X_{i,j}$ or $x_{i,j}$ is the i, j-th entry of the matrix

Matrices

- Examples
  - $X \in \mathbb{R}^{n \times d}$ - Real number
  - $X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ - Real number
  - $Y \in \{0, 1, \dots, C\}^{k \times d}$ - Finite set
  - $U \in [0, 1]^{n \times d}$ - Bounded set

```python
X = np.arange(12).reshape(3,4)
print(X)
W = np.array([
    [1.1343 + 2.1j, 1j, 0.1 + 3.5j],
    [3, 4, 5],
])
print(W)
Z = 5 * np.ones((3, 3), dtype=int)
print(Z)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[1.1343+2.1j 0.    +1.j  0.1   +3.5j]
 [3.    +0.j  4.    +0.j  5.    +0.j ]]
[[5 5 5]
 [5 5 5]
 [5 5 5]]
```

Tensors

- n-D arrays
- Examples
  - $X \in \mathbb{R}^{3 \times h \times w}$, single color image in PyTorch
  - $X \in \mathbb{R}^{n \times 3 \times h \times w}$, multiple color images in PyTorch
  - $X \in \mathbb{R}^{h \times w \times 3}$, single color image for matplotlib imshow

```
1 from sklearn.datasets import load_sample_image
2 china = load_sample_image('china.jpg')
3 print('Shape of image (height, width, channels):', china.shap
4 ax = plt.axes(xticks=[], yticks=[])
5 ax.imshow(china);
```

```
Shape of image (height, width, channels): (427, 640, 3)
```

Matrix transpose

- Changes columns to rows and rows to columns
- Denoted as A^T
- For vectors $\mathbf{v}$, the transpose changes from a column vector to a row vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}, \qquad \mathbf{x}^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}^T = [x_1, x_2, \dots, x_d]$$

```
A = np.arange(6).reshape(2,3)
print(A)
print(A.T)
```

```
[[0 1 2]
 [3 4 5]]
[[0 3]
 [1 4]
 [2 5]]
```

Let's look at the transpose of a row vector (i.e., 1D array) in numpy

```
1  v = np.arange(5)
2  print(v)
3  print(v.shape)
```

```
[0 1 2 3 4]
(5,)
```

(Discussion) What will be the output of the following?

```
1  print(v.T)
2  print(v.T.shape)
```

```
[0 1 2 3 4]
(5,)
```

In numpy, there is only a “vector” (i.e., a 1D array), not really a row or column vector per se.

```
1  v = np.arange(5)
2  print('A numpy vector', v)
3  print('Transpose of numpy vector', v.T)
4  print('A matrix with one column')
5  print(v.shape)
6  print(len(v.shape))
7  V = v.reshape(-1, 1)
8  print('V shape: ', V.shape)
9  print(V)
10 np.dot(v.T, v)
```

```
A numpy vector [0 1 2 3 4]
Transpose of numpy vector [0 1 2 3 4]
A matrix with one column
(5,)
1
V shape:  (5, 1)
[[0]
 [1]
 [2]
 [3]
 [4]]
np.int64(30)
```

Inner product or vector-vector product

- **Inner product** or **vector-vector** multiplication produces *scalar*:

$$\mathbf{x}^T\mathbf{y} = (\mathbf{x}^T\mathbf{y})^T = \mathbf{y}^T\mathbf{x} = x_1y_1 + x_2y_2 + \cdots + x_dy_d$$

Also denoted as:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T\mathbf{y}$$

Can be executed via `np.dot` or `np.matmul`

```
## Inner product
a = np.arange(3)
print(a)
b = np.array([11, 22, 33])
print(b)
np.dot(a, b)
```

```
[0 1 2]
[11 22 33]
np.int64(88)
```

Outer product or vector outer product

- **Vector outer product** of $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$ produces a *matrix*:

$$A = \mathbf{x}\mathbf{y}^T \in \mathbb{R}^{m \times n}$$

  where $A_{i,j} = x_i y_j$.

- Can be executed via `np.outer` or `np.matmul`
- Notice difference from inner product, which produces a scalar and is denoted as $\mathbf{x}^T\mathbf{y}$
  - $\mathbf{x}\mathbf{y}^T$ is also known as the **outer product** of two vectors

```
1 ## Outer product
2 a = np.arange(3)
3 print(a)
4 b = np.array([11, 22])
5 print(b)
6 print(np.outer(a, b))
```

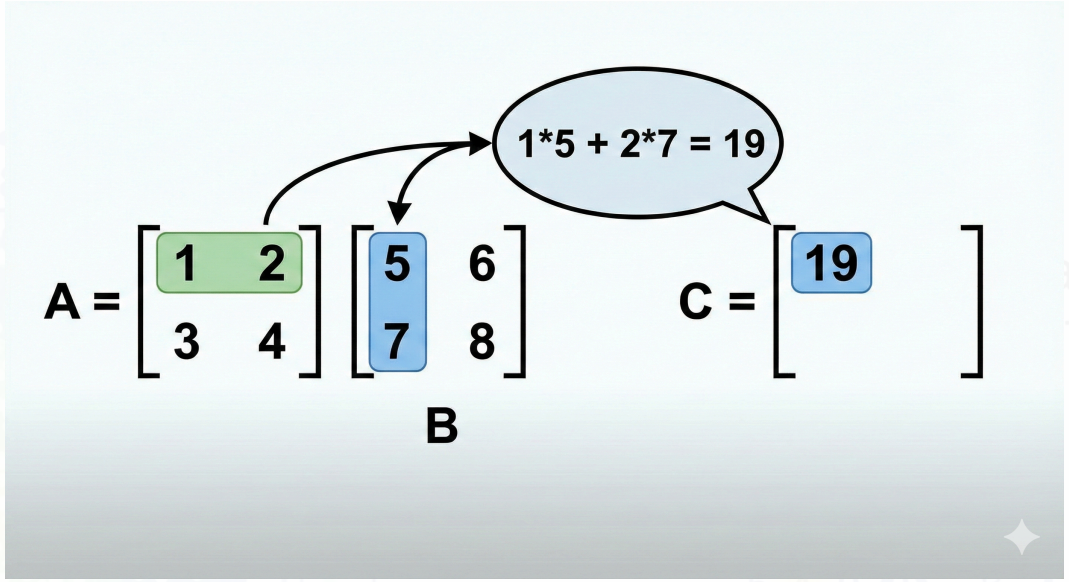
```
[0 1 2]
[11 22]
[[ 0  0]
 [11 22]
 [22 44]]
```

Matrix Product: The Inner Product View (Row × Column)

- This is the standard definition usually taught first.
- To get a **single entry** $C_{i,j}$, we take the **inner product** (dot product) of the i-th row of A (trated as column vector) and the j-th column of B.

$$C_{i,j} = \mathbf{a}_{i,:}^T \mathbf{b}_{:,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

- **Intuition:** We calculate the result one output entry at a time.



Inner product viewpoint illustration.

- **Complexity:** Calculating one entry takes $O(n)$. Calculating all $m \times p$ entries takes $O(mnp)$ or $O(n^3)$ if $m = n = p$.

Matrix Product: The Inner Product View (Row × Column)

```python
A = np.arange(6).reshape(3, 2)
print(A)
B = np.arange(6).reshape(2, 3)
print(B)
C = np.zeros((A.shape[0], B.shape[1]))
for i in range(C.shape[0]):
    for j in range(C.shape[1]):
        for k in range(A.shape[1]):
            C[i, j] += A[i, k] * B[k, j]
print(C)
print(np.matmul(A, B))
print(A @ B)
```

```
[[0 1]
 [2 3]
 [4 5]]
[[0 1 2]
 [3 4 5]]
[[ 3.  4.  5.]
 [ 9. 14. 19.]
 [15. 24. 33.]]
[[ 3  4  5]
 [ 9 14 19]
 [15 24 33]]
[[ 3  4  5]
 [ 9 14 19]
 [15 24 33]]
```

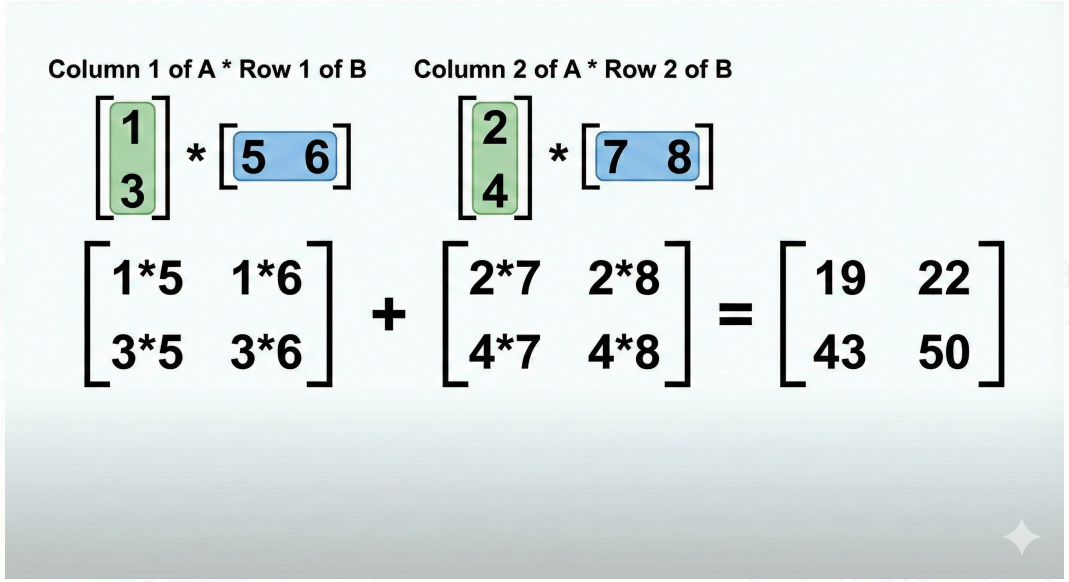
Matrix Product: The Outer Product View (Column × Row)

- We can view the product $C = AB$ as a **sum of simple matrices (i.e., rank 1 which we will define later)** based on the **outer product** of the k-th column of A and the k-th row of B.

$$C = \sum_{k=1}^{n} \mathbf{a}_{:,k} \mathbf{b}_{k,:}^{T}$$

where $\mathbf{a}_{:,k}$ is the k-th column of A $(m \times 1)$ and $\mathbf{b}_{k,:}$ is the k-th row of B $(1 \times p)$ treated as a column vector.

- **Intuition:** We compute one "simple" matrix (i.e., rank 1) and then sum them up.



Outer product viewpoint illustration.

- **Complexity:** Each outer product takes $O(mp)$ to compute. There are n such products, so total complexity is still $O(mnp)$ or $O(n^3)$ if $m = n = p$ (though more memory would be used).

Matrix Product: The Outer Product View (Column × Row)

We can verify that summing outer products equals the standard matrix multiplication.

```python
# Define matrices
m, n, p = 3, 4, 3
A = np.random.randn(m, n)
B = np.random.randn(n, p)

# 1. Standard Matrix Multiplication (@)
C_standard = A @ B

# 2. Sum of Outer Products
C_outer_sum = np.zeros((m, p))
for k in range(n):
    # Outer product of k-th column of A and k-th row of B
    # Shape: (m,) outer (p,) -> (m, p)
    rank_1_component = np.outer(A[:, k], B[k, :])
    C_outer_sum += rank_1_component

print("Are they equivalent?", np.allclose(C_standard, C_outer
```

```
Are they equivalent? True
```

Comparison between these two viewpoints

They compute the same result but in different orders.

1. The Inner Product Viewpoint (Sums inside entries)

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (\mathbf{1 \cdot 5}) + (\mathbf{2 \cdot 7}) & (\mathbf{1 \cdot 6}) + (\mathbf{2 \cdot 8}) \\ (\mathbf{3 \cdot 5}) + (\mathbf{4 \cdot 7}) & (\mathbf{3 \cdot 6}) + (\mathbf{4 \cdot 8}) \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

2. The Outer Product Viewpoint (Sums across matrices)

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} \mathbf{1 \cdot 5} & \mathbf{1 \cdot 6} \\ \mathbf{3 \cdot 5} & \mathbf{3 \cdot 6} \end{bmatrix} + \begin{bmatrix} \mathbf{2 \cdot 7} & \mathbf{2 \cdot 8} \\ \mathbf{4 \cdot 7} & \mathbf{4 \cdot 8} \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Computational Viewpoint: Conceptually both viewpoints calculate a 3D-array of pairwise multiplications and then sum along 1 axis

- Matrix-matrix multiplication has high computational complexity of $O(mnp)$ or $O(n^3)$ if equal dimensionality.
- Theoretically, special linear algebra algorithms can do it $O(n^{2.803})$ (4x faster for $n = 1000$)
- More importantly, careful hardware optimizations like parallelization and cache-sensitive implementations can massively reduce wall-clock time (10x, 100x or even 1000x)
- Takeaway - Use numpy `np.matmul` or `@` operator for matrix multiplication
  - (`np.dot` also works for matrix multiplication but is different in PyTorch and is less explicit so I suggest the two methods above for matrix multiplication)

Element-wise product *NOT equal* to matrix multiplication

- Normal matrix mutiplication $C = AB$ is very different from **element-wise** (or more formally **Hadamard**) multiplication, denoted $F = A \odot D$, which in numpy is just the star *

```
A = np.arange(6).reshape(3, 2)
print(A)
B = np.arange(6).reshape(2, 3)
print(B)
try:
    A * B  # Fails since matrix shapes don't match and cannot
except ValueError as e:
    print('Operation failed! Message below:')
    print(e)
```

```
[[0 1]
 [2 3]
 [4 5]]
[[0 1 2]
 [3 4 5]]
Operation failed! Message below:
operands could not be broadcast together with shapes (3,2)
(2,3)
```

```
print(A)
D = 10*B.T
print(D)
F = A * D  # Element-wise / Hadamard product
print(F)

print(2*F)
```

```
[[0 1]
 [2 3]
 [4 5]]
[[ 0 30]
 [10 40]
 [20 50]]
[[  0  30]
 [ 20 120]
 [ 80 250]]
[[  0  60]
 [ 40 240]
 [160 500]]
```

Properties of matrix product

- Distributive: $A(B+C) = AB + AC$
- Associative: $A(BC) = (AB)C$
- **NOT** commutative, i.e., $AB = BA$ does **NOT** always hold
- Transpose of multiplication (**switch order** and transpose of both): $(AB)^T = B^T A^T$

```
1 print('AB')
2 print(np.matmul(A, B))
3 print('BA')
4 print(np.matmul(B, A))
5 print('(AB)^T')
6 print((A @ B).T)
7 print('B^T A^T')
8 print(np.dot(B.T, A.T))
```

```
AB
[[ 3  4  5]
 [ 9 14 19]
 [15 24 33]]
BA
[[10 13]
 [28 40]]
(AB)^T
[[ 3  9 15]
 [ 4 14 24]
 [ 5 19 33]]
B^T A^T
[[ 3  9 15]
 [ 4 14 24]
 [ 5 19 33]]
```

Identity matrix keeps vectors unchanged

- Multiplying by the identity does not change vector (generalizing the concept of the scalar 1)
- Formally, $I_n \in \mathbb{R}^{n \times n}$, and $\forall \mathbf{x} \in \mathbb{R}^n, I_n \mathbf{x} = \mathbf{x}$
- Structure is ones on the diagonal, zero everywhere else:
- `np.eye` function to create identity

```
1 I3 = np.eye(3)
2 print(I3)
3 x = np.random.randn(3)
4 print(x)
5 print(np.matmul(I3, x))
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[ 0.00679887 -0.40643867 -0.81006979]
[ 0.00679887 -0.40643867 -0.81006979]
```

Matrix inverse times the original matrix is the identity

- The inverse of *square* matrix $A \in \mathbb{n} \times \mathbb{n}$ is denoted as A^{-1} and defined as: $ A^{-1} A = I $
- The "right" inverse is similar and is equal to the left inverse: $ A A^{-1} = I $
- Generalizes the concept of inverse x and $\frac{1}{x}$
- Does **NOT** always exist, similar to how the inverse of x only exists if $x \neq 0$

```python
A = 100 * np.array([[1, 0.5], [0.2, 1]])
print(A)
Ainv = np.linalg.inv(A)
print(Ainv)
print('A^{-1} A = ')
print(np.matmul(Ainv, A))
print('A A^{-1} = ')
print(np.matmul(A, Ainv))
```

```
[[100.  50.]
 [ 20. 100.]]
[[ 0.01111111 -0.00555556]
 [-0.00222222  0.01111111]]
A^{-1} A =
[[1.00000000e+00 0.00000000e+00]
 [2.77555756e-17 1.00000000e+00]]
A A^{-1} =
[[1.00000000e+00 0.00000000e+00]
 [2.77555756e-17 1.00000000e+00]]
```

Singular matrices are similar to zeros

- Informally, singular matrices are matrices that do not have an inverse (similar to the idea that 0 does not have an inverse)
- Consider the 1D equation $ax = b$
  - Usually we can solve for x by multiplying both sides by $1/a$
  - But what if $a = 0$?
  - What are the solutions to the equation?
- Called "singular" because a random matrix is unlikely to be singular just like choosing a random number is unlikely to be 0.

Singular matrices are similar to zeros

```python
from numpy.linalg import LinAlgError
def try_inv(A):
    print('A = ')
    print(np.array(A))
    try:
        np.linalg.inv(A)
    except LinAlgError as e:
        print(e)
    else:
        print('Not singular!')
    print()

try_inv([[0, 0], [0, 0]])
try_inv([[1, 1], [1, 1]])
try_inv([[1, 10], [2, 20]])
try_inv([[1, 10], [20, 2]])
try_inv(np.eye(3))
```

```
A =
[[0 0]
 [0 0]]
Singular matrix

A =
[[1 1]
 [1 1]]
Singular matrix

A =
[[ 1 10]
 [ 2 20]]
Singular matrix

A =
[[ 1 10]
 [20  2]]
Not singular!

A =
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Not singular!
```

Singular matrices are similar to zeros

```
## Random matrix is very unlikely to be 0
for j in range(5):
    try_inv(np.random.randn(2, 2))
```

```
A =
[[-0.44716344 -0.20590876]
 [-0.82002601 -0.21906926]]
Not singular!

A =
[[-0.38478911  1.83891639]
 [-1.11096727 -0.8125224 ]]
Not singular!

A =
[[-0.78075549 -0.90609318]
 [ 1.19261508 -0.46767018]]
Not singular!

A =
[[-0.80698321 -0.10439412]
 [ 0.61246759 -0.90664941]]
Not singular!

A =
[[ 1.6936278  -0.9519357 ]
 [-0.71237146 -0.39254261]]
Not singular!
```

Norms: The "size" of a vector or matrix

- Informally, a generalization of the absolute value of a scalar
- Formally, a norm is an function f that has the following three properties:
  - $f(\mathbf{x}) = 0 \Rightarrow \mathbf{x} = \mathbf{0}$ (zero point)
  - $f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y})$ (Triangle inequality)
  - $\forall \alpha \in \mathbb{R}, f(\alpha \mathbf{x}) = |\alpha| \, f(\mathbf{x})$ (absolutely homogenous)

Examples of Norms

- Absolute value of scalars
- p-norm (also denoted ℓ_p-norm)

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{d} |x_i|^p\right)^{\frac{1}{p}}$$

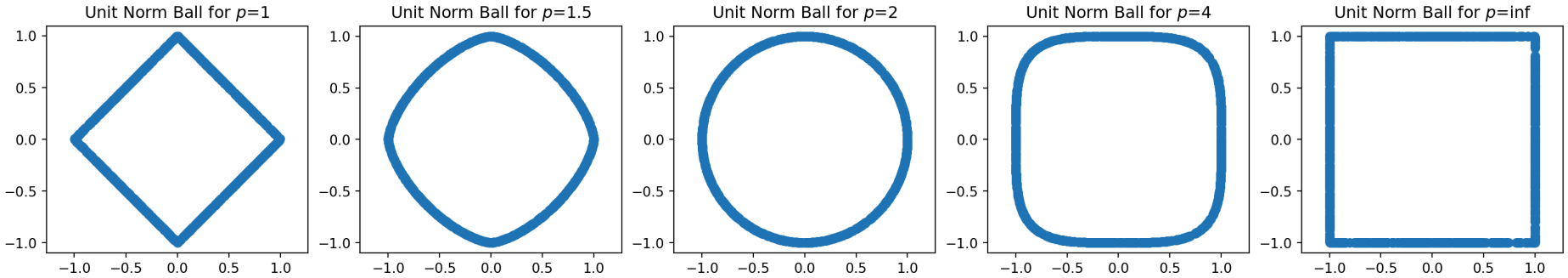
  - (Discussion) What does this represent when $p = 2$ (for simplicity you can assume $d = 2$)?
    - When $p = 2$, we often merely denote as $\|\mathbf{x}\|$.
  - What about when $p = 1$?
  - What about when $p = \infty$ (or more formally the limit as $p \to \infty$)?

```
x = np.array([1, 1])
print(np.linalg.norm(x, ord=2))
print(np.linalg.norm(x, ord=1))
print(np.linalg.norm(x, ord=np.inf))
```

```
1.4142135623730951
2.0
1.0
```

Vectors that have the same norm form a “ball” that isn’t necessarily circular

```
rng = np.random.RandomState(0)
X = rng.randn(1000, 2)

p_vals = [1, 1.5, 2, 4, np.inf]
fig, axes = plt.subplots(1, len(p_vals), figsize=(len(p_vals)*4, 3))

for p, ax in zip(p_vals, axes):
    # Normalize them to have the unit norm
    Z = (X.T / np.linalg.norm(X, ord=p, axis=1)).T
    ax.scatter(Z[:, 0], Z[:, 1])
    ax.axis('equal')
    ax.set_title('Unit Norm Ball for $p$=%g' % p)
```



Squared ℓ_2 norm is quite common since it simplifies to a simple summation

$$\|\mathbf{x}\|_2^2 = \left(\left(\sum_{i=1}^{d} |x_i|^2 \right)^{\frac{1}{2}} \right)^2 = \sum_{i=1}^{d} |x_i|^2 = \sum_{i=1}^{d} x_i^2$$

- Additionally, this can be computed as $\|\mathbf{x}\|_2^2 = \mathbf{x}^T\mathbf{x}$
- Informally, this is analogous to taking the square of a scalar number

```
x = np.arange(4)
print(np.linalg.norm(x, ord=2)**2)
print(np.dot(x, x))
```

```
14.0
14
```

Examples of Matrix Norms

- Frobenius norm (similar to ℓ_2-norm for vectors)
  - $$\|A\|_F = \left\| \begin{bmatrix} a & b \\ c & d \end{bmatrix} \right\|_F = \|[a, b, c, d]\|_2$$
- Spectral norm (max scaling of unit vector)
  - Defined as the maximum scaling effect of the matrix on any unit vector:

$$\|A\|_2 = \max_{\mathbf{x}:\|\mathbf{x}\|_2=1} \|A\mathbf{x}\|_2$$

  - It is equivalent to largest singular value (which we will describe later):

$$\|A\|_2 = \sigma_{\max}$$

Orthogonal vectors

- Orthogonal vectors are vectors such that $\mathbf{x}^T\mathbf{y} = 0$
- The dot product between vectors can be written in terms of norms and the cosine of the angle:

$$\mathbf{x}^T\mathbf{y} = \|\mathbf{x}\|_2\|\mathbf{y}\|_2 \cos\theta$$

- (Discussion) Suppose $\mathbf{x}$ and $\mathbf{y}$ are non-zero vectors, what must θ be if the vectors are orthogonal?

```
print(np.matmul([0, 1], [1, 0]))
theta = np.pi/2
x = np.array([np.cos(theta), -np.sin(theta)])
y = np.array([np.sin(theta), np.cos(theta)])
print(x)
print(y)
print(np.dot(x, y))
```

```
0
[ 6.123234e-17 -1.000000e+00]
[1.000000e+00 6.123234e-17]
0.0
```

Special matrices: Orthogonal matrices

- Informally, an orthogonal matrix only rotates (or reflects) vectors around the origin (zero point), but does not change the size of the vectors.
- Informally, almost analagous to a 1 or -1 for matrices but more general
- A *square* matrix such that $Q^T Q = Q Q^T = I$
- Or, equivalently $Q^{-1} = Q^T$
- Or, equivalently:
  - Every column (or row) is orthogonal to every other column (or row)
  - Every column (or row) has unit ℓ_2-norm, i.e., $\|Q_{i,:}\|_2 = \|Q_{:,j}\|_2 = 1$

Special matrices: Orthogonal matrices

```python
print('Identity matrix')
Q = np.eye(2) # Identity
print(Q)
print(np.allclose(np.eye(2), np.matmul(Q.T, Q)))

print('Reflection matrix')
Q = np.array([[1, 0], [0, -1]]) # Reflection
print(Q)
print(np.allclose(np.eye(2), np.matmul(Q.T, Q)))

print('Rotation matrix')
theta = np.pi/3
Q = np.array([
    [np.cos(theta), -np.sin(theta)],
    [np.sin(theta), np.cos(theta)]
])
print(Q)
print(np.allclose(np.eye(2), np.matmul(Q.T, Q)))
```

```
Identity matrix
[[1. 0.]
 [0. 1.]]
True
Reflection matrix
[[ 1  0]
 [ 0 -1]]
True
Rotation matrix
[[ 0.5       -0.8660254]
 [ 0.8660254  0.5      ]]
True
```

Other special matrices: Symmetric, Triangular, Diagonal

- Symmetric matrices are symmetric around the diagonal; formally, $A = A^T$
- Triangular matrices only have non-zeros in the upper or lower triangular part of the matrix
- Diagonal matrices only have non-zeros along the diagonal of a matrix

```
A = np.arange(9).reshape(3, 3)+1
print('Symmetric')
print(A + A.T)
print('Upper triangular')
print(np.triu(A))
print('Lower triangular')
print(np.tril(A))
print('Diagonal (both upper and lower triangular)')
print(np.diag(np.arange(3) + 1))
```

```
Symmetric
[[ 2  6 10]
 [ 6 10 14]
 [10 14 18]]
Upper triangular
[[1 2 3]
 [0 5 6]
 [0 0 9]]
Lower triangular
[[1 0 0]
 [4 5 0]
 [7 8 9]]
Diagonal (both upper and lower triangular)
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

Multiplying by a diagonal matrix scales the columns or rows

- Right multiplication scales columns; Left multiplication scales rows

```python
A = np.arange(9).reshape(3, 3)
print(A)
D = np.diag(10**(np.arange(3)))
diag_vec = np.diag(D)
print(D)
print('AD')
print(np.matmul(A, D))
print('AD (via numpy * and broadcasting)')
print(A * diag_vec)
print('DA')
print(np.matmul(D, A))
print('DA (via numpy * and broadcasting)')
print(A * diag_vec.reshape(-1, 1))
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[  1   0   0]
 [  0  10   0]
 [  0   0 100]]
AD
[[  0  10 200]
 [  3  40 500]
 [  6  70 800]]
AD (via numpy * and broadcasting)
[[  0  10 200]
 [  3  40 500]
 [  6  70 800]]
DA
[[  0   1   2]
 [ 30  40  50]
 [600 700 800]]
DA (via numpy * and broadcasting)
[[  0   1   2]
 [ 30  40  50]
 [600 700 800]]
```

Programming Topic: NumPy Broadcasting!

> The term broadcasting describes how NumPy treats arrays with **different shapes** during arithmetic operations.

> Broadcasting provides a means of **vectorizing array operations so that looping occurs in C instead of Python**.

> It does this **without making needless copies of data** and usually leads to **efficient algorithm implementations**.

(Quotes from documentation below, emphasis mine)

We will review the basics of broadcasting rules but please see the numpy broadcasting documentation below https://numpy.org/doc/stable/user/basics.broadcasting.html

Broadcasting: Scalar addition / multiplication is a simple case of broadcasting that is already obvious and used even in math notation.

```
1  a = np.arange(5)
2  print(a, 5 + a, 5 * a)
```

```
[0 1 2 3 4] [5 6 7 8 9] [ 0  5 10 15 20]
```

Broadcasting: Subtract column mean or row mean from matrix

To subtract the column mean, we need to reshape the vector into a 2D array:

- Use `a[:, None]` where `None` introduces a new dimension
- Can add multiple dimensions: `a[None, :, None, None]` changes shape `(5,)` to `(1,5,1,1)`
- Alternative: `a.reshape(-1, 1)` where `-1` infers the dimension size
- Reshaping is required for broadcasting (details explained in upcoming examples)

```python
X = np.outer(np.arange(3)+1, np.arange(4)+1)
row_mean = np.mean(X, axis=0)
col_mean = np.mean(X, axis=1)
print(X)
print('Row mean', row_mean)
print('Column mean', col_mean)
print('Remove mean from each row\n',
    X - row_mean)
print('Remove mean from each column\n',
    X - col_mean[:, None])
```

```
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]]
Row mean [2. 4. 6. 8.]
Column mean [2.5 5.  7.5]
Remove mean from each row
 [[-1. -2. -3. -4.]
 [ 0.  0.  0.  0.]
 [ 1.  2.  3.  4.]]
Remove mean from each column
 [[-1.5 -0.5  0.5  1.5]
 [-3.  -1.   1.   3. ]
 [-4.5 -1.5  1.5  4.5]]
```

Broadcasting: You can also divide each row and column by a vector (e.g., to normalize to have standard deviation of 1)

```
1 row_std = np.std(X, axis=0)
2 col_std = np.std(X, axis=1)
3 print('Row normalized X\n', (X - row_mean) / row_std)
4 print('Col normalized X\n', (X - col_mean[:, None]) / col_std
```

```
Row normalized X
 [[-1.22474487 -1.22474487 -1.22474487 -1.22474487]
 [ 0.          0.          0.          0.        ]
 [ 1.22474487  1.22474487  1.22474487  1.22474487]]
Col normalized X
 [[-1.34164079 -0.4472136   0.4472136   1.34164079]
 [-1.34164079 -0.4472136   0.4472136   1.34164079]
 [-1.34164079 -0.4472136   0.4472136   1.34164079]]
```

Broadcasting: Two simple rules for broadcasting

(Copied and slightly modified from documentation)

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. **rightmost**) dimension and works its way left. Two dimensions are compatible when

1. they are equal, or

2. one of them is 1.

In addition, if the number of dimensions is different (e.g., comparing a 4D array to a 2D array), **missing dimensions will be assumed to have a size of 1.**

If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have incompatible shapes.

Broadcasting: Examples of two simple rules for broadcasting

```python
def test_shapes(shape_1, shape_2):
  input_1 = np.ones(shape_1)
  input_2 = np.ones(shape_2)
  print(f'Input 1 shape: {str(input_1.shape):>20}')
  print(f'Input 2 shape: {str(input_2.shape):>20}')

  try:
    output = input_1 * input_2
  except ValueError as e:
    output = None
  if output is None:
    output_shape = 'Invalid input shapes'
  else:
    output_shape = output.shape
  print(f'Output shape:  {str(output_shape):>20}\n')

print('X and row_mean')
test_shapes(X.shape, row_mean.shape)
print('X and col_mean')
test_shapes(X.shape, col_mean.shape)
print('X and col_mean.reshape')
test_shapes(X.shape, col_mean[:, None].shape)
```

```
X and row_mean
Input 1 shape:               (3, 4)
Input 2 shape:                (4,)
Output shape:                (3, 4)

X and col_mean
Input 1 shape:               (3, 4)
Input 2 shape:                (3,)
Output shape:  Invalid input shapes

X and col_mean.reshape
Input 1 shape:               (3, 4)
Input 2 shape:               (3, 1)
Output shape:                (3, 4)
```

Broadcasting: Consider scaling each channel of an RGB image example

```python
scale = np.arange(3)
print('RGB image (matplotlib) and scale of each channel')
test_shapes((256, 256, 3), scale.shape)

print('RGB image (pytorch) and scale of each channel')
test_shapes((3, 256, 256), scale.shape)

print('RGB image (pytorch) and scale of each channel')
test_shapes((3, 256, 256), scale[:, None, None].shape)
```

```
RGB image (matplotlib) and scale of each channel
Input 1 shape:        (256, 256, 3)
Input 2 shape:                (3,)
Output shape:         (256, 256, 3)

RGB image (pytorch) and scale of each channel
Input 1 shape:        (3, 256, 256)
Input 2 shape:                (3,)
Output shape:  Invalid input shapes

RGB image (pytorch) and scale of each channel
Input 1 shape:        (3, 256, 256)
Input 2 shape:            (3, 1, 1)
Output shape:         (3, 256, 256)
```

Broadcasting: Scaling each channel of each image in batch of 32 images

```
1 print('Scaling the channels of a batch of 32 pytorch RGB imag
2 test_shapes((32, 3, 256, 256), scale.shape)
3
4 print('Scaling the channels of a batch of 32 pytorch RGB imag
5 test_shapes((32, 3, 256, 256), scale[:, None, None].shape)
```

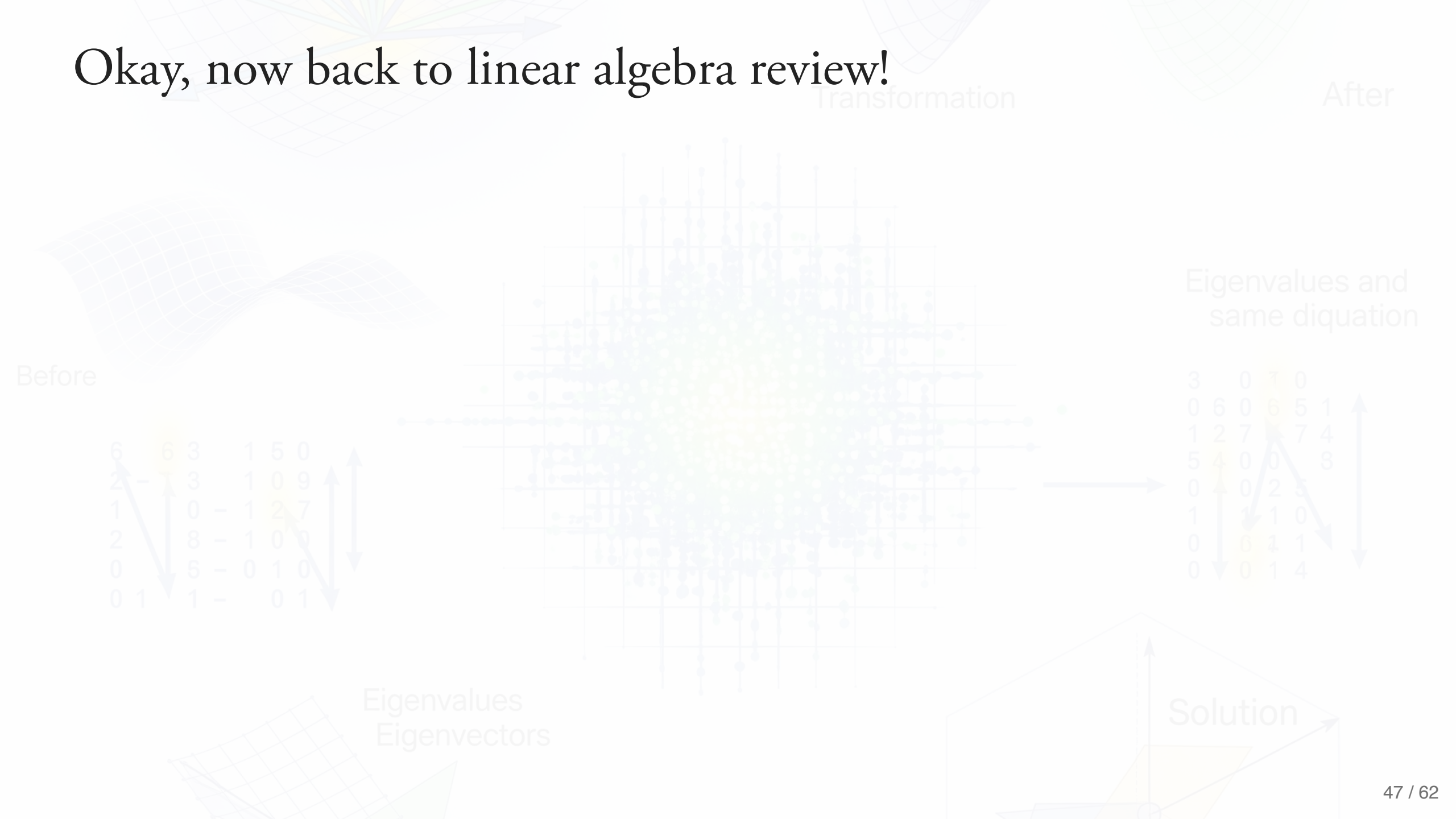
```
Scaling the channels of a batch of 32 pytorch RGB images
Input 1 shape:    (32, 3, 256, 256)
Input 2 shape:                (3,)
Output shape:  Invalid input shapes

Scaling the channels of a batch of 32 pytorch RGB images
Input 1 shape:    (32, 3, 256, 256)
Input 2 shape:             (3, 1, 1)
Output shape:     (32, 3, 256, 256)
```

Broadcasting: Other examples from documentation

```
test_shapes((8,1,6,1), (5,1,3))
test_shapes((5,4), (1,))
test_shapes((5,4), (4,))
test_shapes((15,3,5), (15,1,5))
test_shapes((3,), (4,))
test_shapes((2,1), (8,4,3))
```

```
Input 1 shape:          (8, 1, 6, 1)
Input 2 shape:             (5, 1, 3)
Output shape:           (8, 5, 6, 3)

Input 1 shape:                (5, 4)
Input 2 shape:                 (1,)
Output shape:                 (5, 4)

Input 1 shape:                (5, 4)
Input 2 shape:                 (4,)
Output shape:                 (5, 4)

Input 1 shape:             (15, 3, 5)
Input 2 shape:             (15, 1, 5)
Output shape:              (15, 3, 5)

Input 1 shape:                 (3,)
Input 2 shape:                 (4,)
Output shape:  Invalid input shapes

Input 1 shape:                (2, 1)
Input 2 shape:             (8, 4, 3)
Output shape:  Invalid input shapes
```

Okay, now back to linear algebra review!



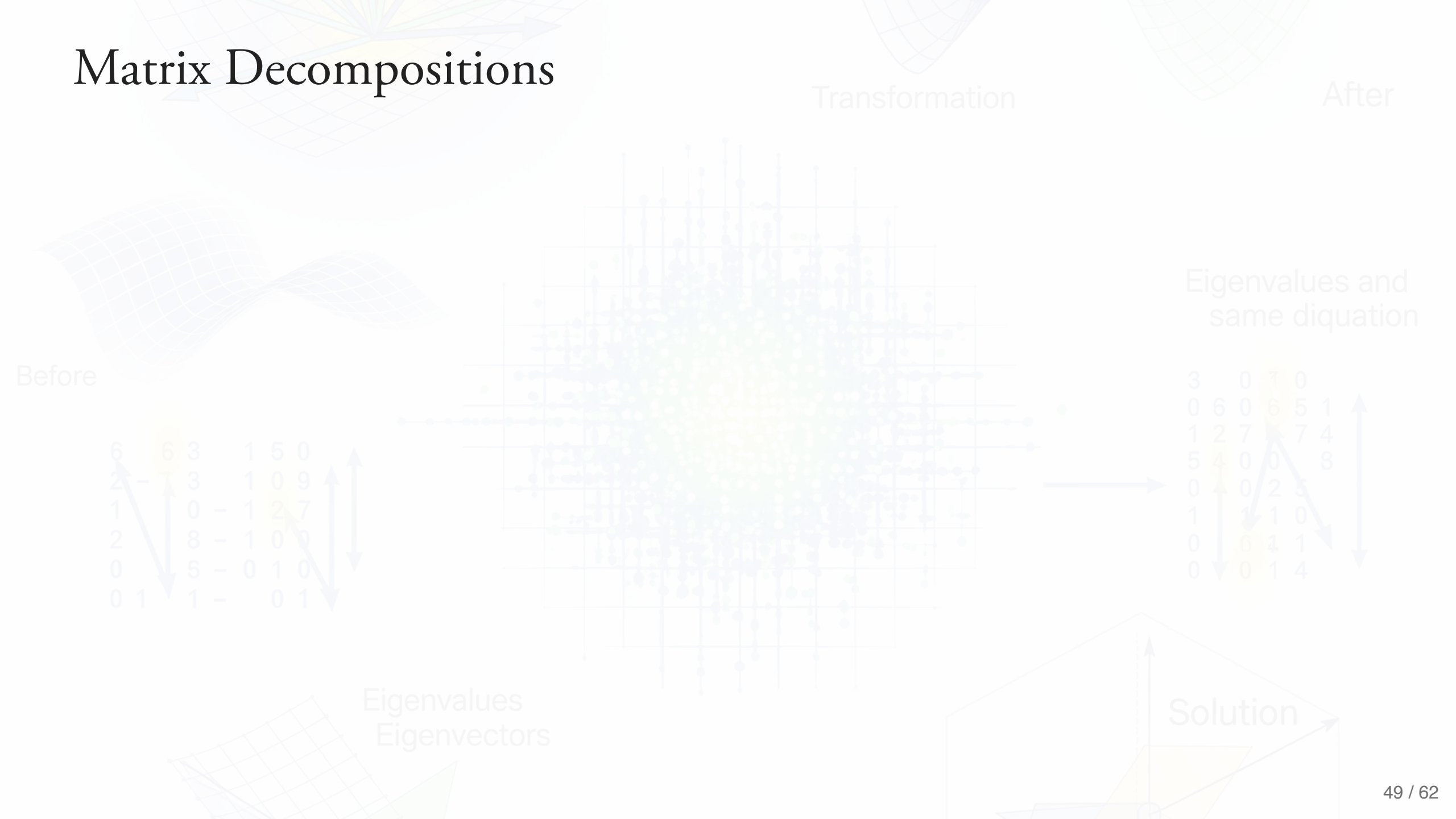
Inverse of diagonal matrix is formed merely by taking inverse of diagonal elements

- Most operations on diagonal matrices are just the scalar versions of their entries

```
A = np.diag(np.arange(5)+1)
print(A)
diag_A = np.diag(A)
print('diag_A', diag_A)
diag_A_inv = 1 / diag_A
print('diag_A_inv', diag_A_inv)
Ainv = np.diag(diag_A_inv)
print(Ainv)
Ainv_full = np.linalg.inv(A)
print(Ainv_full)
```

```
[[1 0 0 0 0]
 [0 2 0 0 0]
 [0 0 3 0 0]
 [0 0 0 4 0]
 [0 0 0 0 5]]
diag_A [1 2 3 4 5]
diag_A_inv [1.         0.5        0.33333333 0.25       0.2
]
[[1.         0.         0.         0.         0.        ]
 [0.         0.5        0.         0.         0.        ]
 [0.         0.         0.33333333 0.         0.        ]
 [0.         0.         0.         0.25       0.        ]
 [0.         0.         0.         0.         0.2       ]]
[[1.         0.         0.         0.         0.        ]
 [0.         0.5        0.         0.         0.        ]
 [0.         0.         0.33333333 0.         0.        ]
 [0.         0.         0.         0.25       0.        ]
 [0.         0.         0.         0.         0.2       ]]
```

Matrix Decompositions



Motivation: Matrix decompositions allow us to *understand* and *manipulate* matrices both theoretically and practically

- Analagous to prime factorization of an integer, e.g., $12 = 2 \times 2 \times 3$
  - Allows us to determine whether things are divisible by other integers
- Analagous to representing a signal in the time versus frequency domain
  - Both domains represent the same object but are useful for different computations and derivations

Eigendecomposition

- For real **symmetric** matrices, the eigendecomposition is:

$$A = Q\Lambda Q^T$$

  where Q is an **orthogonal** matrix and Λ is a **diagonal** matrix.

- λ are known as the **eigenvalues** and Q is known as the **eigenvector matrix**

```python
rng = np.random.RandomState(0)
B = rng.randn(4,4)
A = B + B.T # Make symmetric
lam, Q = np.linalg.eig(A)
print(np.diag(lam))
print(Q)
A_reconstructed = np.matmul(np.matmul(Q, np.diag(lam)), Q.T)
print('Are all entries equal up to machine precision?')
print('Yes' if np.allclose(A, A_reconstructed) else 'No')
```

```
[[ 6.54930093  0.          0.          0.        ]
 [ 0.         -3.728219    0.          0.        ]
 [ 0.          0.          0.45077461  0.        ]
 [ 0.          0.          0.         -0.7428718 ]]
[[ 0.77115168  0.36010163  0.51908231 -0.07877468]
 [ 0.25392564 -0.75129904  0.0518548  -0.60694531]
 [ 0.31251286  0.37021589 -0.78092889 -0.394241  ]
 [ 0.49313545 -0.41087317 -0.34353267  0.68555523]]
Are all entries equal up to machine precision?
Yes
```

Simple properties based on eigendecomposition

- A^{-1} is easy to compute
  - $A^{-1} = (Q\Lambda Q^T)^{-1} = (Q^T)^{-1}\Lambda^{-1}Q^{-1} = Q\Lambda^{-1}Q^T$
  - Easy to solve equation $A\mathbf{x} = \mathbf{b}$
- Powers of matrix is easy to compute $A^3 = AAA$.
  - $A^3 = (Q\Lambda Q^T)(Q\Lambda Q^T)(Q\Lambda Q^T) = Q\Lambda(Q^TQ)\Lambda(Q^TQ)\Lambda Q^T = Q\Lambda^3 Q^T$
- The matrix is singular if and only if there is a zero in λ

Singular value decomposition of *any* matrix (The decomposition to end all decompositions)

- For **any** matrix $A \in \mathbb{R}^{m \times n}$ (even non-square), the singular value decomposition is:

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

  where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are **orthogonal** matrices and $\Sigma \in \mathbb{R}^{m \times n}$ is a **diagonal** (though not necessarily square) matrix.
- Often in notation, it is assumed that the diagonal of Σ, denoted σ is ordered by decreasing values, i.e., $\sigma_1 \geq \sigma_2, \geq \cdots \geq \sigma_d$.
- σ are known as the **singular values** and U and V are known as the **left singular vectors** and the **right singular vectors** respectively.

Singular value decomposition of *any* matrix (The decomposition to end all decompositions)

```python
rng = np.random.RandomState(0)
A = np.arange(6).reshape(2, 3)
print('A', A.shape)
print(A)

## Note returns V^T (i.e. transpose) rather than V
U, s, Vt = np.linalg.svd(A, full_matrices=True)

## Convert singular vector to matrix
Sigma = np.zeros_like(A, dtype=float)
Sigma[:2, :2] = np.diag(s)

print('U', U.shape)
print('Sigma', Sigma.shape)
print('Vt', Vt.shape)

A_reconstructed = np.matmul(U, np.matmul(Sigma, Vt))
print('Are all entries equal up to machine precision?')
print('Yes' if np.allclose(A, A_reconstructed) else 'No')
```

```
A (2, 3)
[[0 1 2]
 [3 4 5]]
U (2, 2)
Sigma (2, 3)
Vt (3, 3)
Are all entries equal up to machine precision?
Yes
```

Rank (denoted $\text{rank}(A)$) is the number of linearly independent columns

- Consider an example of two equations with two unknowns (Is there a unique solution?):
  - $2x + 3y = 0$
  - $4x + 6y = 1$
- Similar to a matrix $A = \begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix}$, notice "redundancy"
- SVD -> Rank = Number of non-zero singular values
- If $A \in \mathbb{R}^{d \times d}$, A is not singular if and only if $\text{rank}(A) = d$.
- Simplest case is rank 1 matrix: $\mathbf{x}\mathbf{y}^T$, which is an outer product

SVD can be used to create compressed (though lossy) approximations of matrices

- SVD provides decomposition of matrix as sum of weighted outer products (i.e., simple rank-1 matrices):

$$A = U\Sigma V^T = \sum_{i=1}^{\text{rank}(A)} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

- If you truncate the sum to the first $k < \text{rank}(A)$ terms, you get a rank k approximation of A:

$$A \approx \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

  - This is known as the **low-rank approximation** of A.
  - The matrix can be stored implicitly using only the first k singular values and vectors.
  - This solves the following optimization problem: $\min_B \|A - B\|_F^2 \quad \text{s.t.} \quad \text{rank}(B) \leq k$.

Image Compression: Let's look at an example of image compression using SVD

```
from sklearn.datasets import load_sample_image
china = load_sample_image('china.jpg')
gray_china = china[:,:,0]/255.0
print('china matrix', gray_china.shape)
#print(gray_china)

U, s, Vt = np.linalg.svd(gray_china)
Sigma = np.zeros_like(gray_china, dtype=float)
Sigma[:427, :427] = np.diag(s)
max_rank = np.min(gray_china.shape)
rank_arr = [0,1, 2, 4, 8, 16, 32, 64,  max_rank]
fig, axes = plt.subplots(3, 3, figsize=(len(rank_arr)*2, 3*4)
for r, ax in zip(rank_arr, axes.ravel()):
    china_approx = np.matmul(U[:, :r], np.matmul(Sigma[:r,:r]
    compression = r/max_rank
    ax.imshow(china_approx, cmap='gray')
    ax.set_title('Rank=%d, Compression=%.1f%%' % (r, compress
```

```
china matrix (427, 640)
```

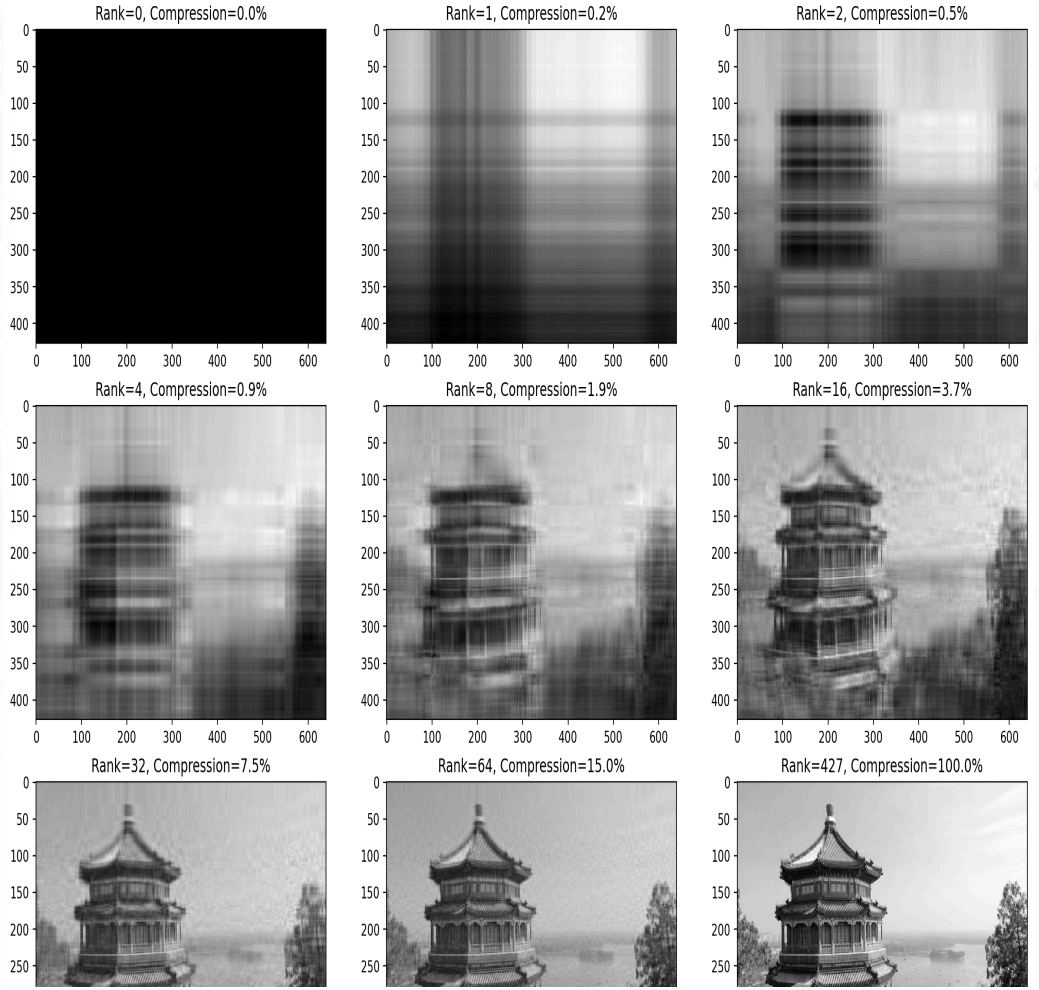


Image Compression: Let's look at the difference between the approximation and the original

```
max_rank = np.min(gray_china.shape)
rank_arr = [0,1, 2, 4, 8, 16, 32, 64, max_rank]
fig, axes = plt.subplots(3, 3, figsize=(len(rank_arr)*2, 3*4))
for r, ax in zip(rank_arr, axes.ravel()):
    china_diff = np.matmul(U[:, :r], np.matmul(Sigma[:r,:r],
    compression = r/max_rank
    ax.imshow(china_diff, cmap='gray', vmin=-1, vmax=1)
    ax.set_title('Difference, Rank=%d, Compression=%.1f%%' %
```

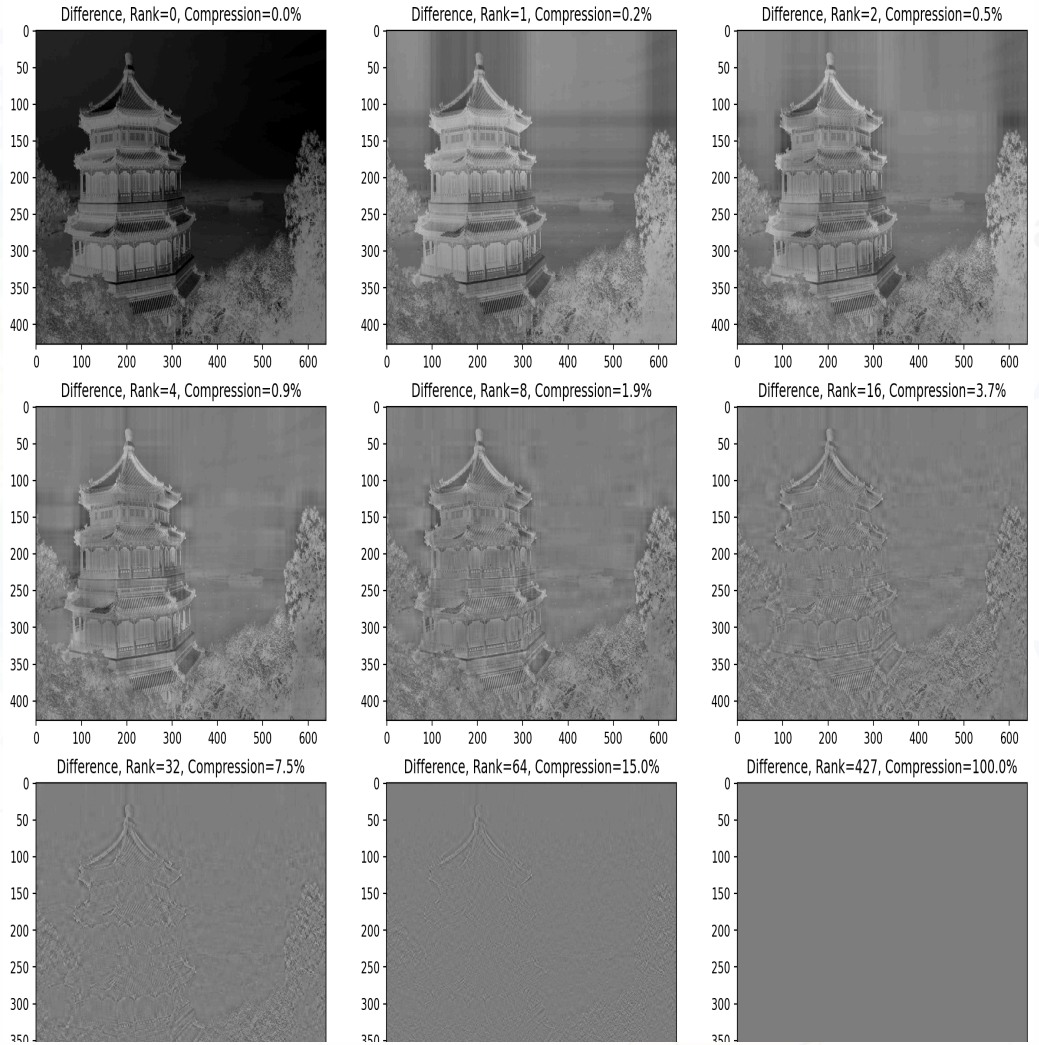
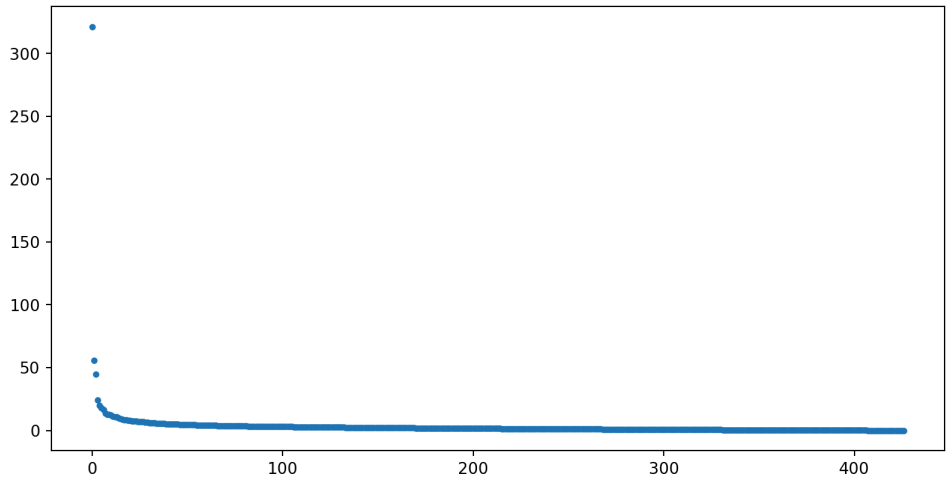


Image Compression: Usually the most important information is in the first few singular values

```
1 plt.plot(s, '.')
```



Matrix trace $\mathrm{Tr}(A)$ operation

- Trace is just the sum of the diagonal elements of a matrix

$$\mathrm{Tr}(A) = \sum_{i=1}^{d} a_{i,i}$$

- Most useful property is rotational equivalence:

$$\mathrm{Tr}(ABC) = \mathrm{Tr}(CAB) = \mathrm{Tr}(BCA)$$

- In particular, (even if different dimensions)

$$\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$$

- Also, trace operator is linear so we have the following properties:

$$\mathrm{Tr}(\alpha A + \beta B) = \alpha \mathrm{Tr}(A) + \beta \mathrm{Tr}(B)$$

Matrix trace $\mathrm{Tr}(A)$ operation

```python
A = np.arange(2*3).reshape(2,3)
B = A.copy().T
print('AB')
print(np.matmul(A, B))
print('Tr(AB)')
print(np.trace(np.matmul(A, B)))
print('Tr(BA)')
print(np.trace(np.matmul(B, A)))
print('Tr(A^T B^T)')
print(np.trace(np.matmul(A.T, B.T)))
print('Tr(B^T A^T)')
print(np.trace(np.matmul(B.T, A.T)))
```

```
AB
[[ 5 14]
 [14 50]]
Tr(AB)
55
Tr(BA)
55
Tr(A^T B^T)
55
Tr(B^T A^T)
55
```

Summary: Linear Algebra with NumPy

- **NumPy vs. Lists**: NumPy arrays perform element-wise operations (e.g., `+`, `*`), whereas Python lists concatenate.
- **Matrix Multiplication**:
  - Use `@` or `np.matmul` for standard multiplication; use `*` for element-wise (Hadamard).
  - **Dual Perspectives**: Can be viewed as **Row** $\times$ **Column** (Inner Products) or **Sum of Column** $\times$ **Row** (Outer Products).
- **Broadcasting**: Vectorizes arithmetic between arrays of different shapes (aligning trailing dimensions) to avoid slow loops.
- **Key Algebra Properties**:
  - Multiplication is **associative** but **not commutative** ($AB \neq BA$).
  - **Norms** (ℓ_p, Frobenius) generalize the concept of absolute value/size.
  - **Orthogonal Matrices** preserve vector norms and angles.
- **Decompositions**:
  - **Eigendecomposition**: $A = Q\Lambda Q^T$ (Symmetric matrices).
  - **SVD**: $A = U\Sigma V^T$ (Any matrix). Fundamental for low-rank approximations and data compression.