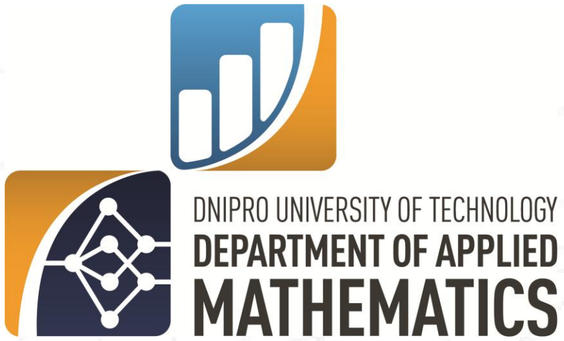


Linear Algebra for Machine Learning

Course "Machine Learning: From Mathematical Foundations to Implementation in PYTHON"

Lecture by prof. Dmytro Babets

1. What Is Linear Algebra and Why Learn It?

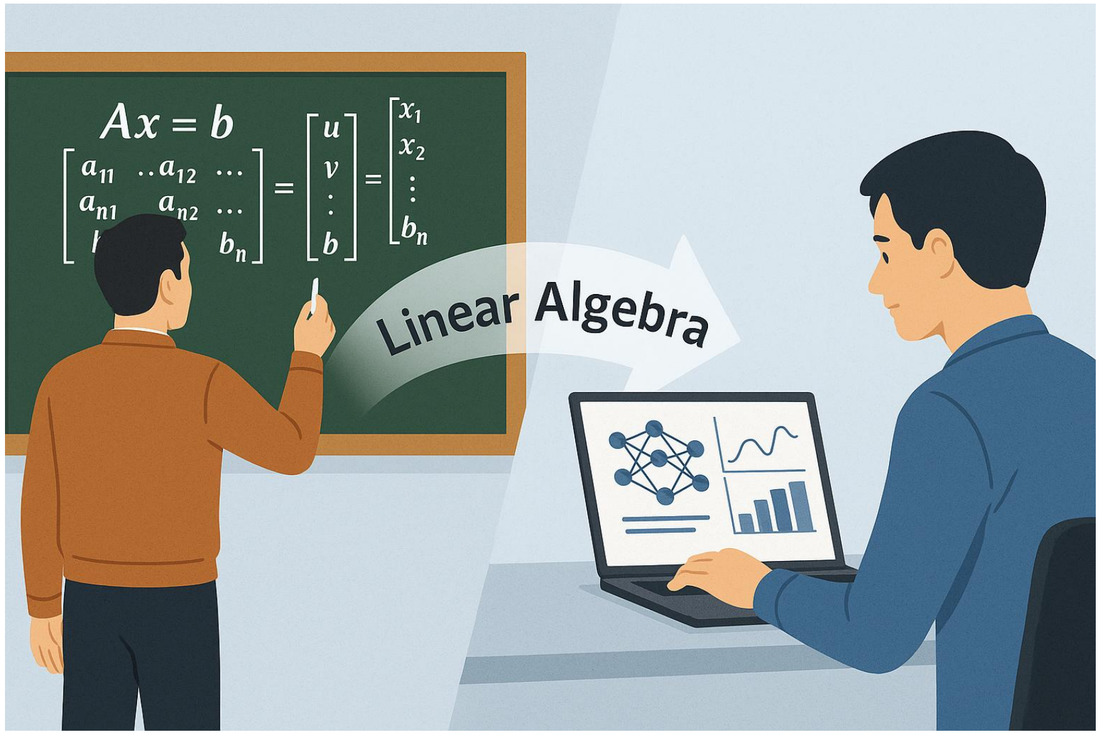
Modern **linear algebra is computational**, whereas traditional linear algebra is abstract. It is best learned through code and applications in graphics, statistics, data science, Machine Learning, and numerical simulations.

Modern linear algebra provides the structural beams that support nearly every algorithm implemented on computers.

Should you learn linear algebra?

If you either want to know how ML algorithms work or want to develop or adapt computational methods.

Yes, you should learn linear algebra!



How linear algebra is used in machine learning?

- Data in ML is represented as vectors and matrices
- Core for model design, optimization, and prediction
- Used in:
  - ✓ Linear Regression (matrix form of solution)
  - ✓ PCA (eigenvectors, eigenvalues)
  - ✓ Neural Networks (matrix of weights and activations)

Example 1: Linear Regression

Suppose you have a dataset with n samples and d features.
Each sample is a row vector: $x^{(i)} = [x_1^{(i)}, x_2^{(i)}, \ldots, x_d^{(i)}]$

We can represent all the data as a matrix:

$$X = \begin{bmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ & \vdots & \\ - & x^{(n)} & - \end{bmatrix}_{n \times d}$$

The linear regression model predicts:

$$\hat{y} = Xw$$

where: X — matrix of inputs (size $n \times d$)

w — weight vector (size $d \times 1$)

$\hat{y}$ — predicted outputs (size $n \times 1$)

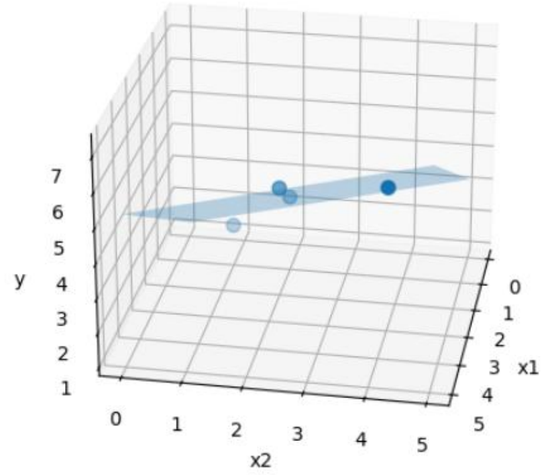
Solution via least squares: $w = (X^T X)^{-1} X^T y$

x1	x2	y
1	1	2
1	2	3
2	2	4
3	4	5

ω = [1.2, 0.8, 0.4]

$\hat{y} = 1.2 + 0.8x_1 + 0.4x_2$

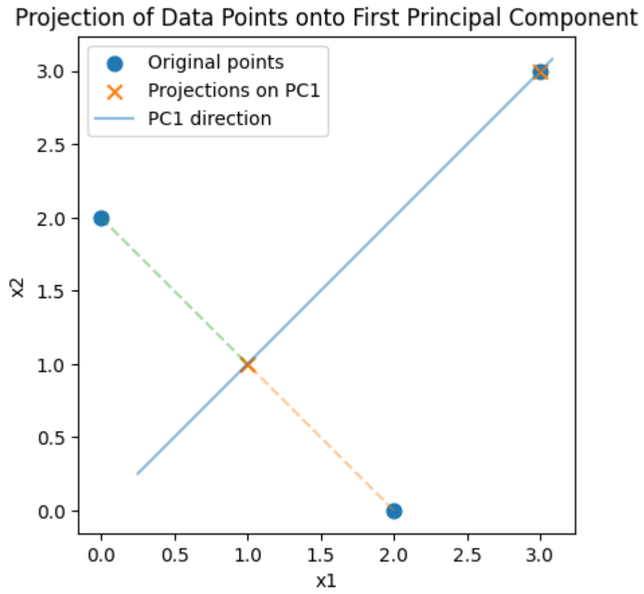
3D Visualization of Data Points and Regression Plane



Example 2: Principal Component Analysis (PCA)

PCA uses eigenvectors and eigenvalues of the covariance matrix: $C = \frac{1}{n} X^T X$

- The principal components are the eigenvectors of C.
- The first eigenvector captures the direction of maximum variance



Input data: 2D points (n = 3)

We start with a simple dataset: $X = \begin{bmatrix} 2 & 0 \\ 0 & 2 \\ 3 & 3 \end{bmatrix}$

1. Center the data $\mu = \frac{1}{3}\begin{bmatrix} 2+0+3 \\ 0+2+3 \end{bmatrix} = \begin{bmatrix} 1.67 \\ 1.67 \end{bmatrix}$

$$X_{\text{centered}} = \begin{bmatrix} 2-1.67 & 0-1.67 \\ 0-1.67 & 2-1.67 \\ 3-1.67 & 3-1.67 \end{bmatrix} = \begin{bmatrix} 0.33 & -1.67 \\ -1.67 & 0.33 \\ 1.33 & 1.33 \end{bmatrix}$$

2. Compute the covariance matrix: $C \approx \begin{bmatrix} 2.33 & 0.33 \\ 0.33 & 2.33 \end{bmatrix}$
3. Eigenvalues and Eigenvectors

$$\lambda_1 = 2.67, \quad \lambda_2 = 2.00$$

$$v_1 = \frac{1}{\sqrt{2}}[1,1] \quad v_2 = \frac{1}{\sqrt{2}}[-1,1]$$

The first principal component (PC1) is the direction [1,1], along which the data has the highest variance. By projecting the original 2D points onto PC1, we get a 1D representation that preserves the maximum amount of information.

Example 3: Neural Networks

In a fully connected (dense) layer:

$$a^{[l]} = \sigma(W^{[l]}a^{[l-1]} + b^{[l]})$$

where:

$W^{[l]}$: weight matrix

$a^{[l-1]}$: activations from previous layer

$b^{[l]}$: bias vector

σ: activation function (e.g. ReLU, sigmoid)

Each layer is a matrix transformation

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]} = \begin{bmatrix} 0.5 & -1.0 \\ 1.5 & 2.0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix}$$

$$W^{[1]}a^{[0]} = \begin{bmatrix} 0.5 * 1 + (-1.0) * 2 \\ 1.5 * 1 + 2.0 * 2 \end{bmatrix} = \begin{bmatrix} 0.5 - 2.0 \\ 1.5 + 4.0 \end{bmatrix} = \begin{bmatrix} -1.5 \\ 5.5 \end{bmatrix}$$

$$z^{[1]} = \begin{bmatrix} -1.5 + 0.1 \\ 5.5 - 0.2 \end{bmatrix} = \begin{bmatrix} -1.4 \\ 5.3 \end{bmatrix}$$

$$a^{[1]} = \mathrm{ReLU}(z^{[1]}) = \begin{bmatrix} \max(0, -1.4) \\ \max(0, 5.3) \end{bmatrix} = \begin{bmatrix} 0.0 \\ 5.3 \end{bmatrix}$$

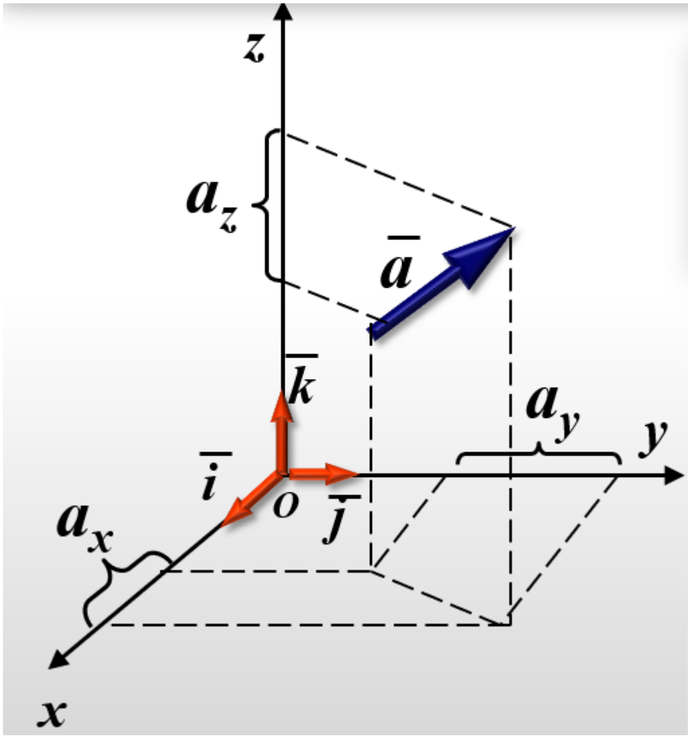
Vectors (algebraic interpretation)

Vectors provide the foundations upon which all linear algebra is built

In linear algebra, a **vector is an ordered list of numbers**

Vectors have several important characteristics:

- Dimensionality – the number of numbers in the vector ($\mathbb{R}^N$)
- Orientation - whether the vector is in column orientation or row orientation



Vectors in Python can be represented using several data types:

```
asList  = [1,2,3]
asArray = np.array([1,2,3]) # 1D array
rowVec  = np.array([ [1,2,3] ]) # row
colVec  = np.array([ [1],[2],[3] ]) # column
```

Many linear algebra operations won't work on Python lists. Therefore, most of the time it's best to create vectors as **NumPy** arrays

Geometry of Vectors

Vector is a straight line with a specific length (also called magnitude) and direction.

The two points of a vector are called the **tail** (A) and the **head** (B); the head often has an arrow tip to disambiguate from the tail

Conceptualizing vectors either geometrically or algebraically facilitates intuition in different applications, but these are simply two sides of the same coin. For example, the geometric interpretation of a vector is useful in physics and engineering (e.g., representing physical forces), and the **algebraic interpretation of a vector is useful in data science** (e.g., storing sales data over time)

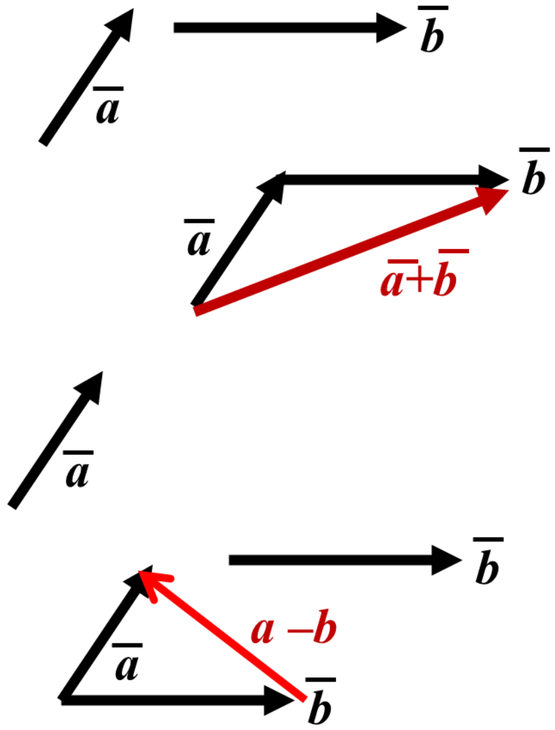
Operations on Vectors: Adding Two Vectors

To add (subtract) two vectors, simply add (subtract) each corresponding element

$$\bar{a} \pm \bar{b} = (a_x \pm b_x, a_y \pm b_y, a_z \pm b_z)$$

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 14 \\ 25 \\ 36 \end{bmatrix}$$

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} - \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} -6 \\ -15 \\ -24 \end{bmatrix}$$



Adding vectors is straightforward in Python:

```
import numpy as np
v = np.array([4,5,6])
w = np.array([10,20,30])
u = np.array([0,3,6,9])
vPlusW = v+w
print(vPlusW)
uPlusW = u+w # error! dimensions mismatched!
```

```
[14 25 36]
```

important: two vectors can be added together only if they have the same dimensionality and the same orientation

Vector-Scalar Multiplication

A scalar in linear algebra is a number on its own, not embedded in a vector or matrix.
Scalars are typically indicated using lowercase Greek letters such as a or λ
Vector-scalar multiplication is indicated as, for example, $\beta \boldsymbol{u}$ or $\beta \bar{u}$

Multiply each vector element by the scalar: $\lambda \bar{a} = (\lambda a_x, \lambda a_y, \lambda a_z)$ ➡ $\lambda = 4,\ \mathbf{w} = \begin{bmatrix} 9 \\ 4 \\ 1 \end{bmatrix}, \quad \lambda \mathbf{w} = \begin{bmatrix} 36 \\ 16 \\ 4 \end{bmatrix}$

In **Python** vector-scalar multiplication is an example where data type matters:

```
s = 2
a = [3,4,5] # as list
b = np.array(a) # as np array
print(a*s)
print(b*s)
```

```
[3, 4, 5, 3, 4, 5]
[ 6  8 10]
```

The code creates a scalar (variable s) and a vector as a list (variable a), then converts that into a NumPy array (variable b).
The asterisk is overloaded in Python, meaning its behavior depends on the variable type: **scalar multiplying a list repeats the list s times**, which is definitely *not* the linear algebra operation.
When the vector is stored as a NumPy array, the asterisk is interpreted as **element-wise multiplication**

Vector Magnitude and Unit Vectors

The **magnitude** of a vector — also called the **geometric length** or the **norm** — is the distance from tail to head of a vector, and is computed using the standard *Euclidean distance formula*:

$$\| \mathbf{v} \| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Vector magnitude is indicated using double-vertical bars

```
v = np.array([1,2,3,7,8,9])
v_dim = len(v) # math dimensionality
print(v_dim)
v_mag = np.linalg.norm(v) # math magnitude, length, or norm
print(v_mag)
```

```
6
14.422205101855956
```

In mathematics, the dimensionality of a vector is the number of elements in that vector, while the length is a geometric distance.
In Python, the function len() (where len is short for length) returns the **dimensionality of an array**, while the function np.norm() returns the **geometric length (magnitude)**

A **unit vector ($\hat{\mathbf{V}}$)** is defined as $\| \mathbf{v} \| = 1$

$$\hat{\mathbf{v}} = \frac{1}{\| \mathbf{v} \|} \mathbf{v}$$

```
v = np.array([1,2,2])
v_mag = np.linalg.norm(v)
v_un = v/v_mag
print(v_un)
```

```
[0.33333333 0.66666667 0.66666667]
```

The Vector Dot Product

The **dot product** is a single number that provides information about the relationship between two vectors

Dot product formula: $\delta = \sum_{i=1}^{n} a_i b_i$ *or* $\bar{a} \cdot \bar{b} = |\bar{a}||\bar{b}| \cos\varphi$

```
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
a_b = np.dot(v,w)
print(a_b)
```

```
70
```

In machine learning, the **dot product** plays a key role as a way to **quantify the similarity or alignment** between two feature vectors.

For example, suppose you're analyzing a dataset that includes height and weight measurements for 20 individuals. These two variables are typically correlated—taller individuals often weigh more—so the dot product of the height and weight vectors would likely be relatively large.

However, it's important to note that the **magnitude** of the dot product is **scale-dependent**. If the measurements are in grams and centimeters, the resulting dot product will be much larger than if the same values are expressed in pounds and feet. To make meaningful comparisons across variables or datasets, we often **normalize the vectors** before computing the dot product.

This normalized version of the dot product is better known as the **Pearson correlation coefficient**, a widely used statistic in data science and machine learning to assess the **linear relationship** between two variables—regardless of their original scale.

Vector Sets

A collection of vectors is called a set

Vector sets are indicated using capital italics letters, like S or V. Mathematically, we can describe sets as the following:

$$V = \{\mathbf{v_1}, \ldots, \mathbf{v_n}\}$$

Vector sets can contain a finite or an infinite number of vectors.

Vector sets can also be empty, and are indicated as V = { }.

Linear Independence

A set of vectors is **linearly dependent** if at least one vector in the set can be expressed as a **linear weighted combination** of other vectors in that set

$$\mathbf{w} = \lambda_1\mathbf{v}_1 + \lambda_2\mathbf{v}_2 + \ldots + \lambda_n\mathbf{v}_n$$

A set of vectors is **linearly independent** if **no vector can be expressed** as a linear weighted combination of other vectors in the set

Example

Try to determine whether each set is dependent or independent:

$$V = \left\{ \begin{bmatrix}1\\3\end{bmatrix}, \begin{bmatrix}2\\7\end{bmatrix} \right\} \qquad S = \left\{ \begin{bmatrix}1\\3\end{bmatrix}, \begin{bmatrix}2\\6\end{bmatrix} \right\}$$

Vector set V is **linearly independent**: it is impossible to express one vector in the set as a linear multiple of the other vector in the set, then there is **no possible** scalar λ for which $\mathbf{v}_1 = \lambda\mathbf{v}_2$.

Vector set S is **dependent**, because we can use linear weighted combinations of some vectors in the set to obtain other vectors in the set. There is an infinite number of such combinations, one of which is $\mathbf{s}_1 = 0.5\mathbf{s}_2$

$$T = \left\{ \begin{bmatrix}8\\-4\\14\\6\end{bmatrix}, \begin{bmatrix}4\\6\\0\\3\end{bmatrix}, \begin{bmatrix}14\\2\\4\\7\end{bmatrix}, \begin{bmatrix}13\\2\\9\\8\end{bmatrix} \right\}$$

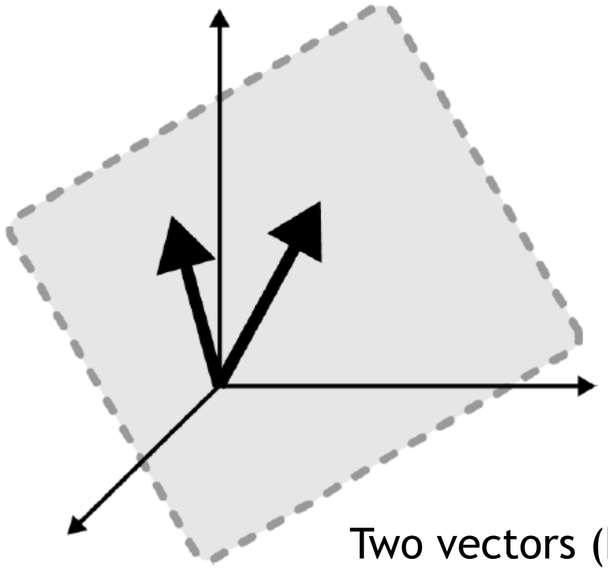
?

The way to determine linear independence is to create a **matrix** from the vector set, compute the **rank** of the matrix, and compare the rank to the smaller of the number of rows or columns

Subspace (підпростір) and Span (лінійна оболонка)

A subspace is the infinite set of all possible linear weighted combinations of a set of vectors.
The dimensionality of the subspace spanned by a set of vectors is the smallest number of vectors that forms a linearly independent set.
If a vector set is linearly independent, then the dimensionality of the subspace spanned by the vectors in that set equals the number of vectors in that set.

The formal definition of a vector subspace is a subset that is closed under addition and scalar multiplication and includes the origin of the space.



Two vectors (black) and the subspace they span (gray)

! Span is the mechanism of creating a subspace. (On the other hand, when you use span as a noun, then span and subspace refer to the same infinite vector set)

Basis

The most common basis set is the **Cartesian axis**: the familiar XY plane that you've used since elementary school. We can write out the basis sets for the 2D and 3D Cartesian graphs as follows:

$$S_2 = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\} \qquad S_3 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

But those are not the only basis sets. The following set is a different basis set for $\mathbb{R}^2$ ⟹ $T = \left\{ \begin{bmatrix} 3 \\ 1 \end{bmatrix}, \begin{bmatrix} -3 \\ 1 \end{bmatrix} \right\}$

Let us describe data points p and q.

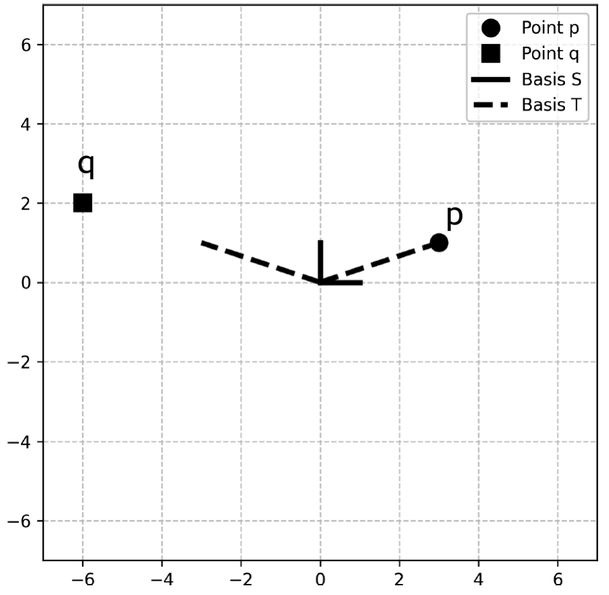
In basis S, those two coordinates are p = (3, 1) and q = (−6, 2).

In this case, that combination is $3\mathbf{s}_1 + 1\mathbf{s}_2$ for point p, and $-6\mathbf{s}_1 + 2\mathbf{s}_2$ for point q

In basis T, we have p = (1, 0) and q = (0, 2).

And in terms of basis vectors, we have $1\mathbf{t}_1 + 0\mathbf{t}_2$ for point p and $0\mathbf{t}_1 + 2\mathbf{t}_2$ for point q.

Again, the data points p and q are the same regardless of the basis set, but T provided a **compact and orthogonal description**



Basis

Definition of Basis. Basis is the combination of span and independence: a set of vectors can be a basis for some subspace if it (1) spans that subspace and (2) is an independent set of vectors.

- The basis needs to span a subspace for it to be used as a basis for that subspace, because you cannot describe something that you cannot measure.
- Why does a basis set require linear independence? The reason is that any given vector in the subspace must have a unique coordinate using that basis.

Bases are extremely important in data science and machine learning. In fact, many problems in applied linear algebra can be conceptualized as finding the best set of basis vectors to describe some subspace:

- dimension reduction,
- feature extraction,
- principal components analysis,
- independent components analysis,
- factor analysis,
- singular value decomposition,
- linear discriminant analysis,
- image approximation,
- data compression

Vectors. Summary

- A vector set is a collection of vectors. There can be a finite or an infinite number of vectors in a set.

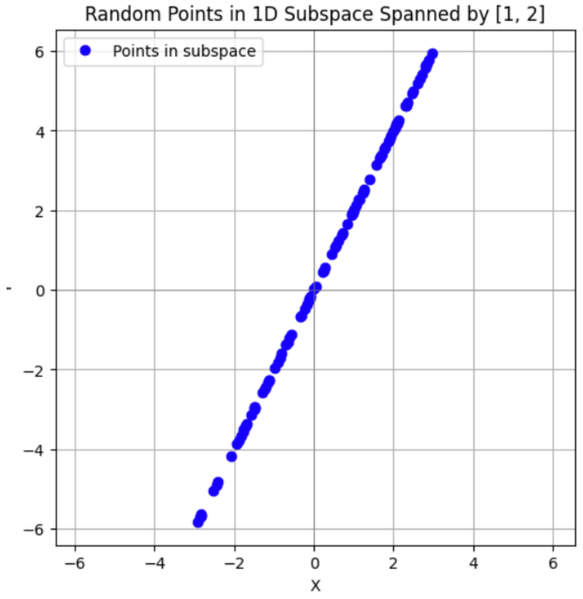
- Linear weighted combination means to scalar multiply and add vectors in a set. Linear weighted combination is one of the single most important concepts in linear algebra.

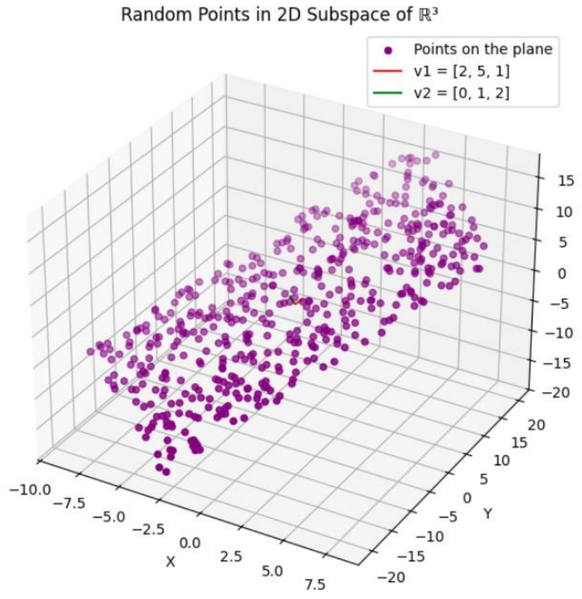
- A set of vectors is linearly dependent if a vector in the set can be expressed as a linear weighted combination of other vectors in the set. And the set is linearly independent if there is no such linear weighted combination.
- A subspace is the infinite set of all possible linear weighted combinations of a set of vectors.

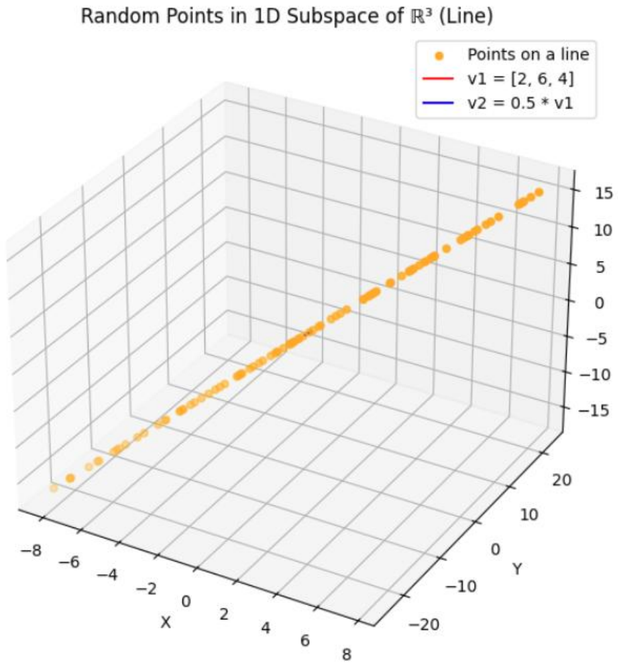
- A basis is a ruler for measuring a space. A vector set can be a basis for a subspace if it (1) spans that subspace and (2) is linearly independent. A major goal in data science is to discover the best basis set to describe datasets or to solve problems.

Code Exercise 1

1. Define a vector set containing one vector [1, 2]. Then create 100 numbers drawn randomly from a uniform distribution between −3 and +3. Those are your random scalars. Multiply the random scalars by the basis vector to create 100 random points in the subspace. Plot those points.
2. Repeat the procedure but using two vectors in $\mathbb{R}^3$: [2, 6, 4] and [-1, 1, 2] and 500 numbers. Note that you need 500 × 2 random scalars for 500 points and two vectors. The resulting random dots will be on a plane.
3. Finally, repeat the $\mathbb{R}^3$ case but setting the second vector to be 1/2 times the first.







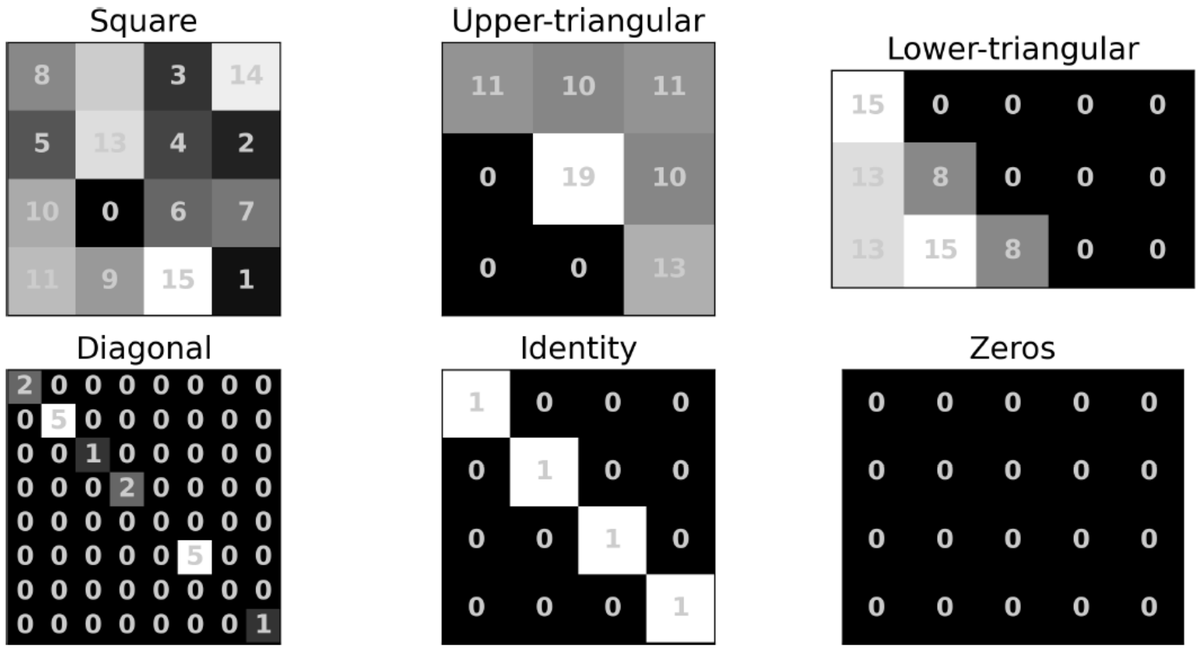
Matrices

A matrix is a vector taken to the next level. Matrices are highly versatile mathematical objects. They can store sets of equations, geometric transformations, the positions of particles over time, financial records, and myriad other things.
In data science, **matrices** are sometimes called **data tables**, in which rows correspond to observations and columns correspond to features.

Matrices are indicated using bold-faced capital letters, like matrix **A** or **M**. The size of a matrix is indicated using (row, column) convention.
You can refer to specific elements of a matrix by indexing the row and column position: the element in the 3rd row and 4th column of matrix **A** is indicated as $a_{3,4}$.

$$\begin{bmatrix} 1 & 3 & 5 & 7 & 9 \\ 0 & 2 & 4 & 6 & 8 \\ 1 & 4 & 7 & 8 & 9 \end{bmatrix}$$

Special Matrices



Matrix Math: Addition and Subtraction

You add two matrices by adding their corresponding elements

$$\begin{bmatrix} 2 & 3 & 4 \\ 1 & 2 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 3 & 1 \\ -1 & -4 & 2 \end{bmatrix} = \begin{bmatrix} (2+0) & (3+3) & (4+1) \\ (1-1) & (2-4) & (4+2) \end{bmatrix} = \begin{bmatrix} 2 & 6 & 5 \\ 0 & -2 & 6 \end{bmatrix}$$

! matrix addition (subtraction) is defined only between two matrices of the same size

$$\mathbf{B} - \mathbf{A} = \begin{pmatrix} -3 & 4 \\ 6 & 1 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 3 & 2 \end{pmatrix} = \begin{pmatrix} -4 & 4 \\ 3 & -1 \end{pmatrix}$$

“Shifting” a Matrix

As with vectors, it is not formally possible to add a scalar to a matrix, as in λ + A. Python allows such an operation, which involves broadcast adding the scalar to each element of the matrix. That is a convenient computation, but it is not formally a linear algebra operation. There is a linear-algebra way to add a scalar to a square matrix, and that is called **shifting a matrix**.

$$\mathbf{A} + \lambda\mathbf{I} \Longrightarrow \begin{bmatrix} 4 & 5 & 1 \\ 0 & 1 & 11 \\ 4 & 9 & 7 \end{bmatrix} + 6 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 10 & 5 & 1 \\ 0 & 7 & 11 \\ 4 & 9 & 13 \end{bmatrix}$$

Only the diagonal elements change; the rest of the matrix is unadulterated by shifting

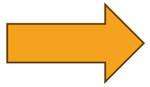
```
A = np.array([ [4,5,1],[0,1,11],[4,9,7] ])
s = 6
A + s # NOT shifting!
A + s*np.eye(len(A)) # shifting
```

```
array([[10.,  5.,  1.],
       [ 0.,  7., 11.],
       [ 4.,  9., 13.]])
```

Scalar and Hadamard Multiplications

Scalar-matrix multiplication means to multiply each element in the matrix by the same scalar.

$$\gamma\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} \gamma a & \gamma b \\ \gamma c & \gamma d \end{bmatrix}$$

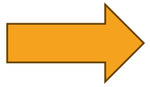


```
[13] A = np.array([ [4,5,1],[0,1,11],[4,9,7] ])
     s = 6
     sA = s * A
     print(sA)
```

```
[[24 30  6]
 [ 0  6 66]
 [24 54 42]]
```

Hadamard multiplication involves multiplying two matrices element-wise (hence the alternative terminology element-wise multiplication)

$$\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \odot \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 2a & 3b \\ 4c & 5d \end{bmatrix}$$



```
A = np.array([ [4,5,1],[0,1,11],[4,9,7] ])
B = np.array([ [0,2,1],[3,1,1],[2,-1,5] ])
C = A*B # Hadamard multiplication
print(C)
np.multiply(A,B) # also Hadamard
D = A@B # NOT Hadamard!
print(D)
```

```
[[ 0 10  1]
 [ 0  1 11]
 [ 8 -9 35]]
[[ 17  12  14]
 [ 25 -10  56]
 [ 41  10  48]]
```

Standard Matrix Multiplication

Matrix multiplication is valid only when the “inner” dimensions match, and the size of the product matrix is defined by the “outer” dimensions

$$A = \begin{pmatrix} \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare \end{pmatrix}$$

$$B = \begin{pmatrix} \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \end{pmatrix}$$

$$\underset{2\times3}{A} \times \underset{3\times4}{B} =$$

The size of the resulting matrix

$$C = A \cdot B = \begin{pmatrix} \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \end{pmatrix}$$

$$D = B \cdot A$$

The reason why matrix multiplication is valid only if the number of columns in the left matrix matches the number of rows in the right matrix is that the $(i,j)^{th}$ element in the product matrix is the dot product between the i^{th} row of the left matrix and the j^{th} column in the right matrix

Example of matrix multiplication

$$A = \begin{pmatrix} 2 & 3 & 1 \\ 0 & -1 & 2 \end{pmatrix}; \quad B = \begin{pmatrix} 3 & 2 & 1 \\ 0 & 1 & -1 \\ 2 & 1 & 2 \end{pmatrix}.$$

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{pmatrix} = \begin{pmatrix} 8 & 8 & 1 \\ 4 & 1 & 5 \end{pmatrix}$$

$$c_{11} = 2 \cdot 3 + 3 \cdot 0 + 1 \cdot 2 = 8$$

$$c_{12} = 2 \cdot 2 + 3 \cdot 1 + 1 \cdot 1 = 8$$

$$c_{13} = 2 \cdot 1 + 3 \cdot (-1) + 1 \cdot 2 = 1$$

$$c_{21} = 0 \cdot 3 + (-1) \cdot 0 + 2 \cdot 2 = 4$$

$$c_{22} = 0 \cdot 2 + (-1) \cdot 1 + 2 \cdot 1 = 1$$

$$c_{23} = 0 \cdot 1 + (-1) \cdot (-1) + 2 \cdot 2 = 5$$

```
import numpy as np
A = np.array([ [2,3,1],[0,-1,2] ])
B = np.array([ [3,2,1],[0,1,-1],[2,1,2] ])
C = A@B # Matrix Multiplication
print(C)
```

```
[[8 8 1]
 [4 1 5]]
```

Geometric transforms

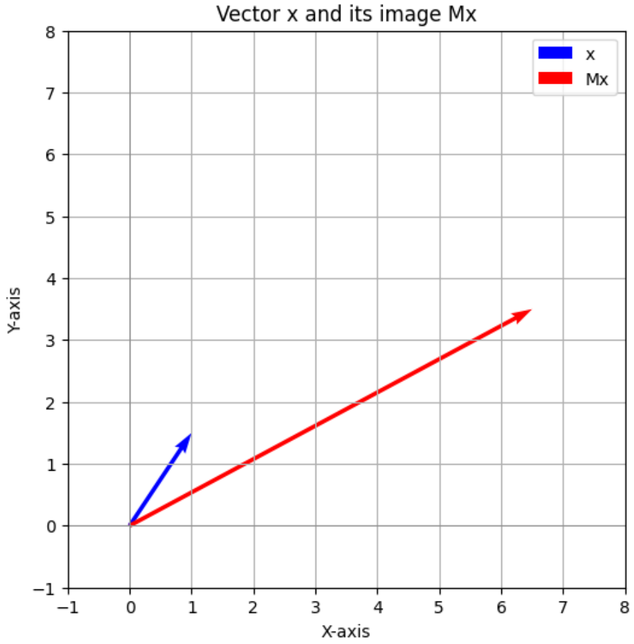
When we think of a vector as a geometric line, then matrix-vector multiplication becomes a way of rotating and scaling that vector.

Matrix-vector multiplication is a Standard Matrix Multiplication where one “matrix” is a vector.

2D case for easy visualization:

```
M = np.array([ [2,3],[2,1] ])
x = np.array([ [1,1.5] ]).T
Mx = M@x
print(Mx)
```

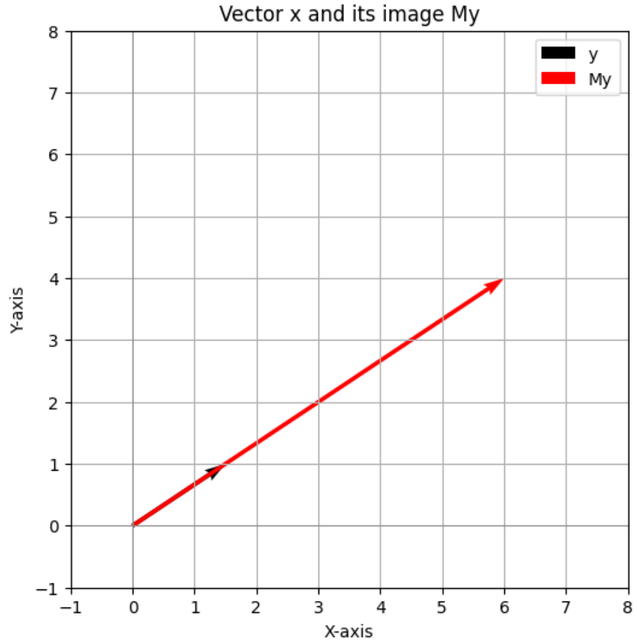
```
[[6.5]
 [3.5]]
```



Graph visualizes these two vectors. You can see that the matrix **M** both rotated and stretched the original vector **x**

```
[7] M = np.array([ [2,3],[2,1] ])
    y = np.array([ [1.5,1] ]).T
    My = M@y
    print(My)
```

```
[[6.]
 [4.]]
```



The matrix-vector product is no longer rotated into a different direction.
That is not a random event: in fact, vector **y** is an **eigenvector** of matrix **M**, and the amount by which **M** stretched **y** is its **eigenvalue**.

Code Exercises 2

1. This exercise will help you gain familiarity with indexing matrix elements. Create a 4 × 5 matrix using `np.arange(20).reshape(4,5)`. Then write Python code to extract the element in the third row, fourth column. Print out a message like the following: The matrix element at index (3,4) is 13.

```
The matrix A is:
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
The matrix element at index (3,4) is 13.
```

2. Code matrix multiplication using `for` loops. Confirm your results against using the numpy `@` operator. This exercise will help you solidify your understanding of matrix multiplication.

```
Result using for loops:
 [[ 58.  64.]
 [139. 154.]]

Result using NumPy @ operator:
 [[ 58  64]
 [139 154]]

✅ The results match!
```

Note: When comparing the manually computed result with the NumPy `@` operator, use `np.allclose()` instead of `==`. This accounts for small numerical differences due to floating-point precision, ensuring a reliable comparison.

Matrix Norms

There is no “the matrix norm”; there are **multiple distinct norms** that can be computed from a matrix.

Matrix norms can be divided into two families:

- Element-wise (computed based on the individual elements of the matrix, and thus these norms can be interpreted to reflect the **magnitudes of the elements** in the matrix);
- Induced (one of the functions of a matrix is to **encode a transformation of a vector**; the induced norm of a matrix is a measure of how much that transformation scales (stretches or shrinks) that vector)

We will consider Element-wise Euclidean norm (extension of the vector norm to matrices). It is is called **Frobenius norm**

$$\| \mathbf{A} \|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{M} \sum_{j=1}^{N} a_{ij}^2}$$

```
import numpy as np

A = np.array([[2, 3],
              [2, 1]])

frobenius_norm = np.linalg.norm(A, 'fro')
print("Frobenius norm of A:", frobenius_norm)
```

```
Frobenius norm of A: 4.242640687119285
```

Matrix norms have several applications in ML.

One of the important applications is in **regularization**, which aims to improve model fitting and increase generalization of models to unseen data.

The basic idea of regularization is to add a matrix norm as a cost function to a minimization algorithm. That norm will help prevent model parameters from becoming too large (ℓ2 regularization) or encouraging sparse solutions (ℓ1 regularization).

In fact, modern deep learning architectures rely on matrix norms to achieve such impressive performance at solving computer vision problems.

Rank

Rank is a number associated with a matrix. It is related to the dimensionalities of matrix subspaces, and has important implications for matrix operations, including inverting matrices and determining the number of solutions to a system of equations.

Rank is the largest number of columns (or rows) that form a linearly independent set

$$\mathbf{A} = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 3 \\ 2 & 6 \\ 4 & 12 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 1 & 3.1 \\ 2 & 6 \\ 4 & 12 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 1 & 3 & 2 \\ 6 & 6 & 1 \\ 4 & 2 & 0 \end{bmatrix}, \quad \mathbf{E} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

r(A) = 1 r(B) = 1 r(C) = 2 r(D) = 3 r(E) = 1 r(F) = 0

Determinant

The **determinant** is a number associated with a square matrix

The determinant of a square matrix is the sum of the products of the elements of any row (or column) and their corresponding cofactors:

$$\Delta = detA = \sum_{j=1}^{m} a_{ij} \cdot C_{ij}$$ - Expanding by the i[th] row

or

$$\Delta = detA = \sum_{i=1}^{n} a_{ij} \cdot C_{ij}$$ - Expanding by the j[th] column

The **cofactor** of element a_{ij} in a matrix is: $C_{ij} = (-1)^{i+j} \cdot \det(M_{ij})$

Computing the determinant of a 2 × 2 matrix

$$det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

Computing the determinant of a 3 × 3 matrix

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh$$

where: M_{ij} is the minor of a_{ij}: the determinant of the submatrix obtained by removing row i and column j from the original matrix. The sign factor $(-1)^{i+j}$ ensures the correct sign pattern

The Characteristic Polynomial

Combining matrix shifting with the determinant is called the characteristic polynomial of the matrix

$det(\mathbf{A} - \lambda\mathbf{I}) = \Delta$ – The characteristic polynomial of the matrix

The matrix below is full rank (Δ = −8), but I'm going to assume that it has a determinant of 0 after being shifted by some scalar λ; the question is, what values of λ will make this matrix reduced-rank?

$$det\left(\begin{bmatrix}1 & 3\\ 3 & 1\end{bmatrix} - \lambda\mathbf{I}\right) = 0 \quad \Rightarrow \quad \begin{vmatrix}1-\lambda & 3\\ 3 & 1-\lambda\end{vmatrix} = 0 \quad \Rightarrow \quad (1-\lambda)^2 - 9 = 0 \quad \Rightarrow$$

$$\lambda = -2 \quad \Rightarrow \quad \begin{bmatrix}3 & 3\\ 3 & 3\end{bmatrix}$$

$$\lambda = 4 \quad \Rightarrow \quad \begin{bmatrix}-3 & 3\\ 3 & -3\end{bmatrix}$$

The solutions to the characteristic polynomial set to Δ = 0 are the **eigenvalues** of the matrix

Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors are important concepts in linear algebra, with applications in various machine learning algorithms. Given a square matrix **A**, an eigenvector **v** and its corresponding eigenvalue λ satisfy the equation:

$$\mathbf{Av} = \lambda\mathbf{v}$$

Eigenvectors represent the direction of linear transformations, while eigenvalues represent the scalar factor by which the eigenvector is scaled.

```python
import numpy as np
A = np.array([[1, 2], [3, 4]])
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigenvalues:")
print(eigenvalues)
print("Eigenvectors:")
print(eigenvectors)
```

```
Eigenvalues:
[-0.37228132  5.37228132]
Eigenvectors:
[[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

Matrix Inverse

The inverse of matrix **A** is another matrix $\mathbf{A}^{-1}$ (pronounced “A inverse”) that multiplies **A** to produce the identity matrix. In other words, $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$.

$$A^{-1} = \frac{1}{\det A}\begin{pmatrix} C_{11} & C_{21} & \cdots & C_{n1} \\ C_{12} & C_{22} & \cdots & C_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ C_{1n} & C_{2n} & \cdots & C_{nn} \end{pmatrix}$$

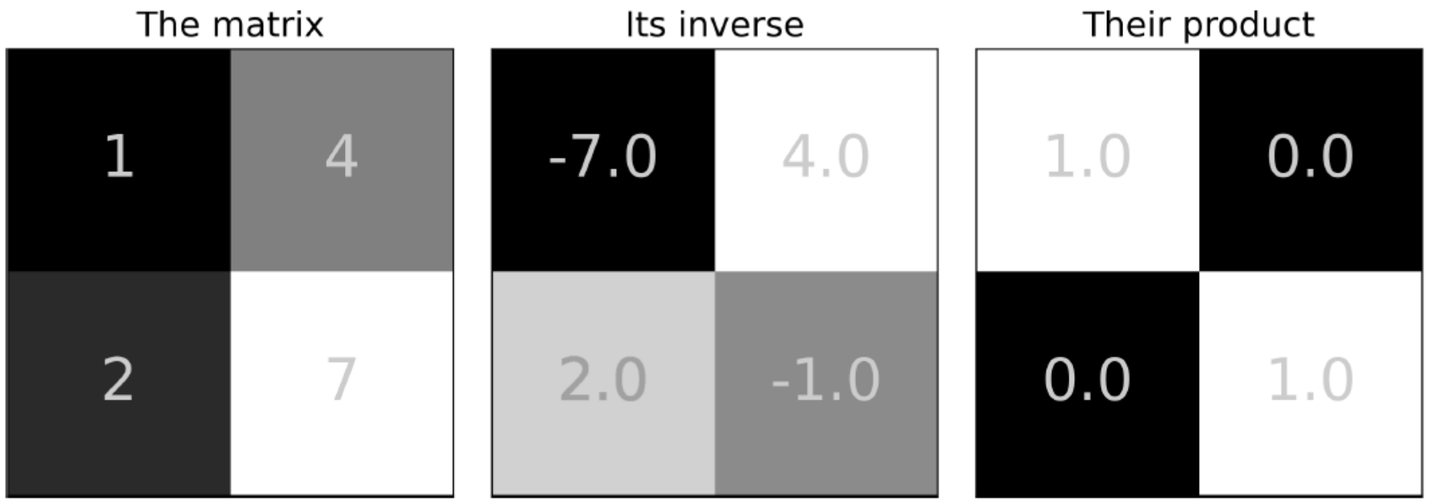
$C_{ij} = (-1)^{i+j} M_{ij}$ - is a cofactor for element a_{ij} of the matrix **A**.

Computing the inverse in Python is easy:

```
import numpy as np

A = np.array([ [1,4],[2,7] ])
Ainv = np.linalg.inv(A)
print(Ainv)
print(A@Ainv)
```

```
[[-7.  4.]
 [ 2. -1.]]
[[1. 0.]
 [0. 1.]]
```



You can confirm that `A@Ainv` gives the identity matrix

Code Exercises

1. The inverse of the inverse is the original matrix; in other words, $(A^{-1})^{-1} = A$. Illustrate this using Python.

2. The norm of a matrix is related to the scale of the numerical values in the matrix. In this exercise, you will create an experiment to demonstrate this. In each of 10 experiment iterations, create a 10 × 10 random numbers matrix and compute its Frobenius norm. Then repeat this experiment 40 times, each time scalar multiplying the matrix by a different scalar that ranges between 0 and 50. The result of the experiment will be a 40 × 10 matrix of norms. Figure 6-7 shows the resulting norms, averaged over the 10 experiment iterations. This experiment also illustrates two additional properties of matrix norms: they are strictly nonnegative and can equal 0 only for the zeros matrix.

3. Check for Linear Independence of Vectors. Write a function that takes a 2D NumPy array where each row is a vector and determines whether the set of vectors is linearly independent.

4. Matrix Multiplication and Associativity. Demonstrate the associativity of matrix multiplication: $(A \cdot B) \cdot C = A \cdot (B \cdot C)$. Use randomly generated 3x3 matrices.

5. Apply Linear Transformation to 2D Vectors. Apply a 2D transformation matrix (e.g., rotation) to a set of vectors and visualize both the original and transformed vectors.

Let’s proceed to the practical exercises

link:
https://colab.research.google.com/drive/1MsiEeZ2IHUmxUcJW81t_PUsLSHKnC1T5?usp=sharing

REFERENCES

1. Cohen, M. X. (2022). *Practical linear algebra for data science: From Core Concepts to Applications Using Python*. “O’Reilly Media, Inc.”
2. Cohen, M. X. (2021). *Linear algebra: theory, intuition, code*.
3. Елементи лінійної алгебри: навчальний посібник / П.М. Щербаков, С.Є. Тимченко, А.Г. Шпорта, Д.В. Бабець, Ю.М. Головко; М-во освіти і науки України, Нац. техн. ун-т «Дніпровська політехніка». – Дніпро: НТУ «ДП», 2023. – 166 с.