Linear independence

Linear dependence

A collection or list of n-vectors $a_1, \ldots, a_k$ (with $k \geq 1$) is called *linearly dependent* if

$$\beta_1 a_1 + \cdots + \beta_k a_k = 0$$

holds for some $\beta_1, \ldots, \beta_k$ that are not all zero. In other words, we can form the zero vector as a linear combination of the vectors, with coefficients that are not all zero.

- Linear dependence of a list of vectors does not depend on the ordering of the vectors in the list.

Linear combinations of linearly independent vectors.

Suppose a vector x is a linear combination of $a_1, \ldots, a_k$,

$$x = \beta_1 a_1 + \cdots + \beta_k a_k.$$

When the vectors $a_1, \ldots, a_k$ are linearly independent, the coefficients that form x are unique: If we also have

$$x = \gamma_1 a_1 + \cdots + \gamma_k a_k$$

then $\beta_i = \gamma_i$ for $i = 1, \ldots, k$. This tells us that, in principle at least, we can find the coefficients that form a vector x as a linear combination of linearly independent vectors.

In [ ]:

```
import numpy as np
from scipy.linalg import null_space

# Define the vectors
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
v3 = np.array([7, 8, 9])

# Stack the vectors into a matrix
A = np.c_[v1, v2, v3]
print("Matrix A:")
print(A)

# Find the null space of the matrix A
coefficients = null_space(A)

print("Null space of the matrix A:")
print(coefficients)

# Since the null space returns a basis for the null space, we can scale it
# to get integer coefficients if needed
if coefficients.size > 0:  # Check if the null space is not empty
    # Get the first non-zero vector from the null space
    coeffs = coefficients[:, 0]
    # Scale to get integer coefficients
    coeffs = np.round(coeffs / np.min(np.abs(coeffs))).astype(int)
    print("Coefficients of linear dependence:", coeffs)
    print("v3 represented as a linear combination of v1 and v2:")
    print(f"v3 = {coeffs[0]} * v1 + {coeffs[1]} * v2")
    print(f"v3 = {coeffs[0] * v1 + coeffs[1] * v2}")
else:
    print("The vectors are linearly independent.")
```

Basis

Independence-dimension inequality.

If the n-vectors $a_1, \ldots, a_k$ are linearly independent, then $k \leq n$. In words:

> A linearly independent collection of n-vectors can have at most n elements.

Put another way:

> Any collection of $n + 1$ or more n-vectors is linearly dependent.

As a very simple example, we can conclude that any three 2-vectors must be linearly dependent. We will prove this fundamental fact below; but first, we describe the concept of basis, which relies on the independence-dimension inequality.

Basis.

A collection of n linearly independent n-vectors (i.e., a collection of linearly independent vectors of the maximum possible size) is called a **basis**.

If the n-vectors $a_1, \ldots, a_n$ are a basis, then any n-vector b can be written as a linear combination of them. To see this, consider the collection of $n + 1$ n-vectors $a_1, \ldots, a_n, b$. By the independence-dimension inequality, these vectors are linearly dependent, so there are $\beta_1, \ldots, \beta_{n+1}$, not all zero, that satisfy

$$\beta_1 a_1 + \cdots + \beta_n a_n + \beta_{n+1} b = 0$$

If $\beta_{n+1} = 0$, then we have

$$\beta_1 a_1 + \cdots + \beta_n a_n = 0$$

which, since $a_1, \ldots, a_n$ are linearly independent, implies that $\beta_1 = \cdots = \beta_n = 0$. But then all the β_i are zero, a contradiction. So we conclude that $\beta_{n+1} \neq 0$. It follows that

$$b = (-\beta_1/\beta_{n+1})\, a_1 + \cdots + (-\beta_n/\beta_{n+1})\, a_n$$

i.e., b is a linear combination of $a_1, \ldots, a_n$.

Combining this result with the observation above that any linear combination of linearly independent vectors can be expressed in only one way, we conclude:

> Any n-vector b can be written in a unique way as a linear combination of a basis $a_1, \ldots, a_n$.

Expansion in a basis.

When we express an n-vector b as a linear combination of a basis $a_1, \ldots, a_n$, we refer to

$$b = \alpha_1 a_1 + \cdots + \alpha_n a_n$$

as the expansion of b in the $a_1, \ldots, a_n$ basis. The numbers $\alpha_1, \ldots, \alpha_n$ are called the coefficients of the expansion of b in the basis $a_1, \ldots, a_n$.

Examples

- The n standard unit n vectors $e_1, \ldots, e_n$ are a basis. Any n-vector b can be written as the linear combination

$$b = b_1 e_1 + \cdots + b_n e_n.$$

This expansion is unique, which means that there is no other linear combination of $e_1, \ldots, e_n$ that equals b.

- The vectors

$$a_1 = \begin{bmatrix} 1.2 \\ -2.6 \end{bmatrix}, \quad a_2 = \begin{bmatrix} -0.3 \\ -3.7 \end{bmatrix}$$

are a basis. The vector $b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ can be expressed in only one way as a linear combination of them:

$$b = 0.6513 a_1 - 0.7280 a_2.$$

(The coefficients are given here to 4 significant digits. We will see later how these coefficients can be computed.)

In [ ]:
```
a = np.array([1.2, -2.6])
b = np.array([-0.3, -3.7])

# show that a and b forms a basis for R^2
A = np.column_stack((a, b))

print("Matrix A:")
print(A)

coefficients = null_space(A)
print("Null space of the matrix A:")
print(coefficients)
if coefficients.size == 0:
    print("The vectors are linearly independent.")
else:
    print("The vectors are linearly dependent.")

# express the vector [1, 1] as a linear combination of a and b
v = np.array([1, 1]).reshape(-1, 1)
coeffs = np.linalg.solve(A, v)
coeffs = coeffs[:, 0]
print("Coefficients of linear dependence:")
print(coeffs)
print(f"v = {coeffs[0]:.4f} * a - {-coeffs[1]:.4f} * b")
print(f"v = {coeffs[0] * a + coeffs[1] * b}")
```

Orthonormal vectors

A collection of vectors $a_1, \ldots, a_k$ is orthogonal or mutually orthogonal if $a_i \perp a_j$ for any i, j with $i \neq j, i, j = 1, \ldots, k$. A collection of vectors $a_1, \ldots, a_k$ is orthonormal if it is orthogonal and $\|a_i\| = 1$ for $i = 1, \ldots, k$.

- A vector of norm one is called normalized; dividing a vector by its norm is called normalizing it.
- Thus, each vector in an orthonormal collection of vectors is normalized, and two different vectors from the collection are orthogonal.

These two conditions can be combined into one statement about the inner products of pairs of vectors in the collection: $a_1, \ldots, a_k$ is orthonormal means that

$$a_i^T a_j = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}.$$

- Orthonormality, like linear dependence and independence, is an attribute of a collection of vectors, and not an attribute of vectors individually.
- By convention, though, we say "The vectors $a_1, \ldots, a_k$ are orthonormal" to mean "The collection of vectors $a_1, \ldots, a_k$ is orthonormal".

In [ ]:
```
vectors = {
    "a1": np.array([0, 0, -1]),
    "a2": np.array([1, 1, 0]) / np.sqrt(2),
    "a3": np.array([1, -1, 0]) / np.sqrt(2),
}

for name, vector in vectors.items():
    print(f"norm of {name}: {np.linalg.norm(vector)}")

for name1, vector1 in vectors.items():
    for name2, vector2 in vectors.items():
        if name1 != name2:
            print(f"inner product of {name1} and {name2}: {np.dot(vector1, vector2)}"
```

In [ ]:
```
x = np.array([1, 2, 3])

betas = {}
for name, vector in vectors.items():
    betas[name] = np.dot(vector, x)
    print(f"beta_{name}: {betas[name]}")

print(f"betas: {betas}")

xexp = sum([betas[name] * vectors[name] for name in vectors.keys()])
print(f"expansion: {xexp}")
print(f"original: {x}")
```

Gram–Schmidt algorithm

Algorithm: given n-vectors $a_1, \ldots, a_k$ for $i = 1, \ldots, k$,

1. Orthogonalization. $\tilde{q}_i = a_i - \left(q_1^T a_i\right) q_1 - \cdots - \left(q_{i-1}^T a_i\right) q_{i-1}$
2. Test for linear dependence. if $\tilde{q}_i = 0$, quit.
3. Normalization. $q_i = \tilde{q}_i / \|\tilde{q}_i\|$

In [5]:
```
def gram_schmidt(a, tol=1e-10):
    q = []
    for i in range(len(a)):
        qtilde = a[i]
        for j in range(i):
            qtilde = qtilde - np.inner(q[j], a[i]) * q[j]
        if np.linalg.norm(qtilde) < tol:
            print("Vectors linearly dependent")
        q.append(qtilde / np.linalg.norm(qtilde))
    return q
```

In [ ]:
```
a = [[-1, 1, -1, 1], [-1, 3, -1, 3], [1, 3, 5, 7]]
q = gram_schmidt(a)
print(f"The orthonormal vectors are:\n{q}")
# print(np.array(q))
print(f"Norms of vectors {np.linalg.norm(q, axis=1)}")
```

> Portfolio return time series.

Suppose that R is a $T \times n$ asset return matrix, that gives the returns of n assets over T periods. A common trading strategy maintains constant investment weights given by the n-vector w over the T periods.

For example, $w_4 = 0.15$ means that 15% of the total portfolio value is held in asset 4. (Short positions are denoted by negative entries in w.) Then Rw, which is a T-vector, is the time series of the portfolio returns over the periods $1, \ldots, T$.

As an example, consider a portfolio of the 4 assets in table, with weights $w = (0.4, 0.3, -0.2, 0.5)$.

The product $Rw = (0.00213, -0.00201, 0.00241)$ gives the portfolio returns over the three periods in the example.

	Apple (AAPL)	Google (GOOG)	3M (MMM)	Amazon (AMZN)
March 1, 2016	0.00219	0.00006	-0.00113	0.00202
March 2, 2016	0.00744	-0.00894	-0.00019	-0.00468
March 3, 2016	0.01488	-0.00215	0.00433	-0.00407

Table: Daily returns of Apple (AAPL), Google (GOOG), 3M (MMM), and Amazon (AMZN), on March 1, 2, and 3, 2016 (based on closing prices).

In [ ]:
```python
# daily returns for 4 assets (AAPL, GOOG, MMM, AMZN)
base_returns = np.array(
    [
        [0.00219, 0.00006, -0.00113, 0.00202],
        [0.00744, -0.00894, -0.00019, -0.00468],
        [0.01488, -0.00215, 0.00433, -0.00407],
    ]
)

# Define the investment weights
w = np.array([0.4, 0.3, -0.2, 0.5])  # Portfolio weights

# Calculate the portfolio returns Rw
portfolio_returns = base_returns @ w  # Matrix multiplication

print("Asset Returns:")
print(base_returns)
print("\nPortfolio Weights:")
print(w)
print("\nPortfolio Returns:")
print(portfolio_returns.round(5))
```

Polynomial evaluation at multiple points. Suppose the entries of the n-vector c are the coefficients of a polynomial p of degree $n-1$ or less:

$$p(t) = c_1 + c_2 t + \cdots + c_{n-1} t^{n-2} + c_n t^{n-1}.$$

Let $t_1, \ldots, t_m$ be m numbers, and define the m-vector y as $y_i = p(t_i)$. Then we have $y = Ac$, where A is the $m \times n$ matrix

$$A = \begin{bmatrix} 1 & t_1 & \cdots & t_1^{n-2} & t_1^{n-1} \\ 1 & t_2 & \cdots & t_2^{n-2} & t_2^{n-1} \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & t_m & \cdots & t_m^{n-2} & t_m^{n-1} \end{bmatrix}$$

So multiplying a vector c by the matrix A is the same as evaluating a polynomial with coefficients c at m points. The matrix A comes up often, and is called a Vandermonde matrix (of degree $n-1$, at the points $t_1, \ldots, t_m$), named for the mathematician Alexandre-Théophile Vandermonde.

In [ ]:
```
# Define the polynomial coefficients
# Example: p(t) = 2 + 3t + t^2 (coefficients: [2, 3, 1])
coefficients = np.array([2, 3, 1])

# Define the points at which we want to evaluate the polynomial
t_values = np.array([0, 1, 2, 3, 4, 5, 6])

# Create the Vandermonde matrix
# The shape of the matrix will be (m, n) where m is the number of points and n is the
# m = len(t_values)
n = len(coefficients)
A = np.vander(t_values, N=n, increasing=True)

print("Vandermonde Matrix A:")
print(A)
```

In [ ]:
```
# Evaluate the polynomial at the given points
y_values = A @ coefficients
y_values
```

In [ ]:
```
import matplotlib.pyplot as plt

# Create a range of t values for a smooth curve
t_curve = np.linspace(t_values[0], t_values[-1], 100)

y_curve = np.polyval(coefficients[::-1], t_curve)  # Reverse coefficients for np.poly

fig = plt.figure(figsize=(10, 6))

# Scatter plot of the evaluated points
plt.scatter(t_values, y_values, color="red", label="Evaluated Points", zorder=5)

# Plot the polynomial curve
plt.plot(t_curve, y_curve, label="Polynomial Curve", color="blue")

plt.title("Polynomial Evaluation using Vandermonde Matrix")
plt.xlabel("t")
plt.ylabel("p(t)")
plt.axhline(0, color="black", linewidth=0.5, ls="--")
plt.axvline(0, color="black", linewidth=0.5, ls="--")
plt.grid()
plt.legend()
plt.show()
plt.close(fig)
```

Total price from multiple suppliers. Suppose the $m \times n$ matrix P gives the prices of n goods from m suppliers (or in m different locations). If q is an n-vector of quantities of the n goods (sometimes called a basket of goods), then $c = Pq$ is an N-vector that gives the total cost of the goods, from each of the N suppliers.

In [ ]:
```
# Example prices for 3 suppliers and 4 computer parts
P = np.array(
    [
        [100, 200, 150, 300],  # Supplier 1 prices
        [110, 190, 160, 290],  # Supplier 2 prices
```

```
        [105, 205, 155, 310],  # Supplier 3 prices
    ]
)

# Example quantities for the 4 computer parts
q = np.array([2, 3, 1, 4])  # 2 CPUs, 3 GPUs, 1 Motherboard, 4 RAM sticks


c = P @ q
print("Total costs from each supplier:", c)

supplier_labels = ["Supplier 1", "Supplier 2", "Supplier 3"]
plt.bar(supplier_labels, c, color=["blue", "orange", "green"])
plt.xlabel("Suppliers")
plt.ylabel("Total Cost ($)")
plt.title("Total Cost of Goods from Different Suppliers")
plt.grid(axis="y")
plt.show()
```

> **Document scoring.** Suppose A in an $N \times n$ document-term matrix, which gives the word counts of a corpus of N documents using a dictionary of n words, so the rows of A are the word count vectors for the documents. Suppose that w in an n-vector that gives a set of weights for the words in the dictionary. Then $s = Aw$ is an N-vector that gives the scores of the documents, using the weights and the word counts.
>
> A search engine, for example, might choose w (based on the search query) so that the scores are predictions of relevance of the documents (to the search).

In [ ]:

```
# Step 1: Define the documents
doc1 = """
Renewable energy sources, such as solar and wind, are becoming increasingly vital in 
The transition from fossil fuels to renewable energy can significantly reduce greenho
Solar panels harness sunlight, converting it into electricity, while wind turbines ca
Governments worldwide are investing in renewable energy infrastructure to promote sus
Additionally, renewable energy creates jobs in manufacturing, installation, and maint
The use of renewable resources can lead to energy independence, reducing reliance on 
As technology advances, the efficiency of renewable energy systems continues to impro
Public awareness and education about renewable energy are essential for widespread ad
The future of energy lies in sustainable practices that protect our planet.
"""
doc2 = """
Artificial Intelligence (AI) is revolutionizing the healthcare industry by enhancing 
Machine learning algorithms analyze vast amounts of medical data to identify patterns
AI-powered tools assist doctors in diagnosing diseases earlier and more accurately.
For instance, AI can analyze medical images to detect anomalies that may be missed by
Furthermore, AI chatbots provide patients with immediate responses to their inquiries
The integration of AI in healthcare also streamlines administrative tasks, allowing h
However, ethical considerations regarding data privacy and algorithmic bias must be a
As AI technology evolves, its potential to transform healthcare continues to grow.
Collaboration between technologists and healthcare providers is crucial for successfu
"""

documents = [doc1, doc2]

# Step 2: Create the document-term matrix (A)
# For simplicity, let's assume we have a vocabulary of 5 words.
# Word dictionary: [renewable, energy, sources, AI, healthcare]
# Initialize the document-term matrix with zeros
A = np.zeros((len(documents), 5))

# Define the vocabulary
vocabulary = ["renewable", "energy", "sources", "ai", "healthcare"]
```

```python
# Count word occurrences in each document
for i, doc in enumerate(documents):
    for word in doc.replace("-", " ").lower().split():
        # Remove punctuation and check for word in vocabulary
        word_cleaned = word.strip(",.!?()")  # Remove punctuation
        if word_cleaned in vocabulary:
            A[i, vocabulary.index(word_cleaned)] += 1


print("Document-Term Matrix (A):")
print(A)

# Step 3: Define the weight vector (w)
# Let's assign weights based on the importance of each word for the query "renewable e
w = np.array([3, 2, 0, 0, 0])  # Higher weight for 'renewable' and 'energy'

# Step 4: Calculate the document scores (s)
s = np.dot(A, w)
```

In [ ]:
```python
# Step 5: Visualizations
# 1. Document-Term Matrix Visualization
fig = plt.figure(figsize=(10, 4))
plt.imshow(A, cmap="hot", interpolation="nearest")
plt.colorbar(label="Word Count")
plt.title("Document-Term Matrix")
plt.xlabel("Words")
plt.ylabel("Documents")
plt.xticks(ticks=np.arange(5), labels=vocabulary)
plt.yticks(
    ticks=np.arange(len(documents)),
    labels=[f"Document {i + 1}" for i in range(len(documents))],
)
plt.show()
plt.close(fig)
```

In [ ]:
```python
# 2. Weight Vector Visualization
fig = plt.figure(figsize=(6, 4))
plt.bar(vocabulary, w, color="skyblue")
plt.title('Weight Vector for Query "Renewable Energy"')
plt.xlabel("Words")
plt.ylabel("Weights")
plt.xticks(rotation=45)
plt.grid(axis="y")
plt.show()
plt.close(fig)
```

In [ ]:
```python
# 3. Score Visualization
fig = plt.figure(figsize=(6, 4))
plt.bar([f"Document {i + 1}" for i in range(len(documents))], s, color="lightgreen")
plt.title("Document Scores Based on Weights")
plt.xlabel("Documents")
plt.ylabel("Scores")
plt.ylim(0, max(s) + 1)  # Set y-limit for better visualization
plt.grid(axis="y")
plt.show()
```

In [ ]:
```python
# Step 6: Output Summary
print("Document Word Counts:")
for i, doc in enumerate(documents):
    word_count = len(doc.split())
    print(f"Document {i + 1} Word Count: {word_count}")

print("\nDocument Scores:")
for i, score in enumerate(s):
```

```python
    print(f"Score for Document {i + 1}: {score}")
```

> **Audio mixing.** Suppose the k columns of A are vectors representing audio signals or tracks of length T, and w is a k-vector. Then $b = Aw$ is a T-vector representing the mix of the audio signals, with track weights given by the vector w.

In [20]:
```python
import IPython.display as ipd

# Set parameters
sample_rate = 44100  # Samples per second
duration = 10  # Duration of each track in seconds
t = np.linspace(0, duration, int(sample_rate * duration), endpoint=False)

# Create synthetic audio signals
# Track 1: Drum beat (simple square wave)
drum_beat = 0.5 * np.sign(np.sin(2 * np.pi * 2 * t))  # 2 Hz square wave

# Track 2: Guitar riff (sine wave)
guitar_riff = 0.5 * np.sin(2 * np.pi * 440 * t)  # (440 Hz)

# Track 3: Vocal line (sine wave)
vocal_line = 0.5 * np.sin(2 * np.pi * 550 * t)  # (550 Hz)

# Combine tracks into matrix A
A = np.column_stack((drum_beat, guitar_riff, vocal_line))

# Define weights for mixing
w = np.array([0.5, 0.3, 0.2])  # Weights for each track

# Mix the audio signals
b = A @ w  # Resulting mixed audio signal

# Normalize the mixed signal to prevent clipping
b = b / np.max(np.abs(b))
```

In [ ]:
```python
# Play the mixed audio directly
ipd.Audio(b, rate=sample_rate)
```

In [ ]:
```python
# Visualization of waveforms
plt.figure(figsize=(12, 8))

# Plot individual tracks
plt.subplot(4, 1, 1)
plt.title("Drum Beat")
plt.plot(t, drum_beat)
plt.xlim(0, duration)
plt.ylim(-1, 1)

plt.subplot(4, 1, 2)
plt.title("Guitar Riff")
plt.plot(t, guitar_riff)
plt.xlim(0, duration)
plt.ylim(-1, 1)

plt.subplot(4, 1, 3)
plt.title("Vocal Line")
plt.plot(t, vocal_line)
plt.xlim(0, duration)
plt.ylim(-1, 1)

# Plot mixed track
plt.subplot(4, 1, 4)
plt.title("Mixed Audio Signal")
```

```
plt.plot(t, b)
plt.xlim(0, duration)
plt.ylim(-1, 1)

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