Introduction to Linear Algebra for Applied Machine Learning with Python

26 May 2020

Linear algebra is to machine learning as flour to bakery: **every machine learning model is based in linear algebra, as every cake is based in flour**. It is not the only ingredient, of course. Machine learning models need vector calculus, probability, and optimization, as cakes need sugar, eggs, and butter. Applied machine learning, like bakery, is essentially about combining these mathematical ingredients in clever ways to create useful (tasty?) models.

This document contains **introductory level linear algebra notes for applied machine learning**. It is meant as a reference rather than a comprehensive review. If you ever get confused by matrix multiplication, don't remember what was the L_2 norm, or the conditions for linear independence, this can serve as a quick

reference. It also a good introduction for people that don't need a deep understanding of linear algebra, but still want to learn about the fundamentals to read about machine learning or to use pre-packaged machine learning solutions. Further, it is a good source for people that learned linear algebra a while ago and need a refresher.

These notes are based in a series of (mostly) freely available textbooks, video lectures, and classes I've read, watched and taken in the past. If you want to obtain a deeper understanding or to find exercises for each topic, you may want to consult those sources directly.

Free resources:

- **Mathematics for Machine Learning** by Deisenroth, Faisal, and Ong. 1st Ed. Book link.
- **Introduction to Applied Linear Algebra** by Boyd and Vandenberghe. 1sr Ed. Book link
- **Linear Algebra Ch. in Deep Learning** by Goodfellow, Bengio, and Courville. 1st Ed. Chapter link.
- **Linear Algebra Ch. in Dive into Deep Learning** by Zhang, Lipton, Li, And Smola. Chapter link.
- **Prof. Pavel Grinfeld's Linear Algebra Lectures** at Lemma. Videos link.
- **Prof. Gilbert Strang's Linear Algebra Lectures** at MIT. Videos link.
- **Salman Khan's Linear Algebra Lectures** at Khan Academy. Videos link.
- **3blue1brown's Linear Algebra Series** at YouTube. Videos link.

Not-free resources:

- **Introduction to Linear Algebra** by Gilbert Strang. 5th Ed. Book link.
- **No Bullshit Guide to Linear Algebra** by Ivan Savov. 2nd Ed. Book Link.

I've consulted all these resources at one point or another. Pavel Grinfeld's lectures are my absolute favorites. Salman Khan's lectures are really good for absolute beginners (they are long though). The famous 3blue1brown series in linear algebra is delightful to watch and to get a solid high-level view of linear

algebra.

If you have to pic one book, I'd pic **Boyd's and Vandenberghe's Intro to applied linear algebra**, as it is the most beginner friendly book on linear algebra I've encounter. Every aspect of the notation is clearly explained and pretty much all the key content for applied machine learning is covered. The Linear Algebra Chapter in Goodfellow et al is a nice and concise introduction, but it may require some previous exposure to linear algebra concepts. Deisenroth et all book is probably the best and most comprehensive source for linear algebra for machine learning I've found, although it assumes that you are good at reading math (and at math more generally). Savov's book it's also great for beginners but requires time to digest. Professor Strang lectures are great too but I won't recommend it for absolute beginners.

I'll do my best to keep notation consistent. Nevertheless, learning to adjust to changing or inconsistent notation is a useful skill, since most authors will use their own preferred notation, and everyone seems to think that its/his/her own notation is better.

To make everything more dynamic and practical, I'll introduce bits of Python code to exemplify each mathematical operation (when possible) with `NumPy`, which is the facto standard package for scientific computing in Python.

Finally, keep in mind this is created by a non-mathematician for (mostly) non-mathematicians. I wrote this as if I were talking to myself or a dear friend, which explains why my writing is sometimes conversational and informal.

If you find any mistake in notes feel free to reach me out at pcaceres@wisc.edu and to https://pablocaceres.org/ so I can correct the issue.

Table of contents

Note: *underlined sections* are the newest sections and/or corrected ones.

Preliminary concepts

While writing about linear mappings, I realized the importance of having a basic understanding of a few concepts before approaching the study of linear algebra. If you are like me, you may not have formal mathematical training beyond high school. If so, I encourage you to read this section and spent some time wrapping your head around these concepts before going over the linear algebra content (otherwise, you might prefer to skip this part). I believe that reviewing these concepts is of great help to understand the *notation*, which in my experience is one of the main barriers to understand mathematics for nonmathematicians: we are *non*native speakers, so we are continuously building up our vocabulary. I'll keep this section very short, as is not the focus of this mini-course.

For this section, my notes are based on readings of:

>> **Geometric transformations (Vol. 1)** (1966) by Modenov & Parkhomenko
>> **Naive Set Theory** (1960) by P.R. Halmos
>> **Abstract Algebra: Theory and Applications** (2016) by Judson & Beeer. Book link

Sets

Sets are one of the most fundamental concepts in mathematics. They are so fundamental that they are not defined in terms of anything else. On the contrary, other branches of mathematics are defined in terms of sets, including linear algebra. Put simply, **sets are well-defined collections of objects**. Such objects are called **elements or members** of the set. The crew of a ship, a caravan of camels, and the LA Lakers roster, are all examples of sets. The captain of the ship, the first camel in the caravan, and LeBron James are all

examples of "members" or "elements" of their corresponding sets. We denote a set with an upper case italic letter as A. In the context of linear algebra, we say that a line is a set of points, and the set of all lines in the plane is a set of sets. Similarly, we can say that *vectors* are sets of points, and *matrices* sets of vectors.

Belonging and inclusion

We build sets using the notion of **belonging**. We denote that a *belongs* (or is an *element* or *member* of) to A with the Greek letter epsilon as:

$$a \in A$$

Another important idea is **inclusion**, which allow us to build *subsets*. Consider sets A and B. When every element of A is an element of B, we say that A is a *subset* of B, or that B *includes* A. The notation is:

$$A \subset B$$

or

$$B \supset A$$

Belonging and inclusion are derived from **axion of extension**: *two sets are equal if and only if they have the same elements*. This axiom may sound trivially obvious but is necessary to make belonging and inclusion rigorous.

Set specification

In general, anything we assert about the elements of a set results in **generating a subset**. In other words, asserting things about sets is a way to manufacture subsets. Take as an example the set of all dogs, that I'll denote as D. I can assert now "d is black". Such an assertion is true for some members of the set of all dogs and false for others. Hence, such a sentence, evaluated for *all* member of D, generates a subset: *the set of all black dogs*. This is denoted as:

$$B = \{d \in D : d \text{ is black}\}$$

or

$$B = \{d \in D \mid d \text{ is black}\}$$

The colon ($:$) or vertical bar ($|$) read as "such that". Therefore, we can read the above expression as: *all elements of d in D such that d is black*. And that's how we obtain the set B from A.

Set generation, as defined before, depends on the **axiom of specification**: *to every set A and to every condition $S(x)$ there corresponds a set B whose elements are exactly those elements $a \in A$ for which $S(x)$ holds.*

A condition $S(x)$ is any *sentence* or *assertion* about elements of A. Valid sentences are either of *belonging* or *equality*. When we combine belonging and equality assertions with logic operators (not, if, and or, etc), we can build any legal set.

Ordered pairs

Pairs of sets come in two flavors: *unordered* and *ordered*. We care about pairs of sets as we need them to define a notion of relations and functions (from here I'll denote sets with lower-case for convenience, but keep in mind we're still talking about sets).

Consider a pair of sets x and y. An **unordered pair** is a set whose elements are x, y, and $x, y = y, x$. Therefore, presentation order does not matter, the set is the same.

In machine learning, we usually do care about presentation order. For this, we need to define an **ordered pair** (I'll introduce this at an intuitive level, to avoid to introduce too many new concepts). An **ordered pair** is denoted as (x, y), with x as the *first coordinate* and y as the *second coordinate*. A valid ordered pair has the property that $(x, y) \neq (y, x)$.

Relations

From ordered pairs, we can derive the idea of **relations** among sets or between elements and sets. Relations can be binary, ternary, quaternary, or N-ary. Here we are just concerned with binary relationships. In set theory, **relations** are defined as *sets of ordered pairs*, and denoted as R. Hence, we can express the relation

between x and y as:

$$x \, R \, y$$

Further, for any $z \in R$, there exist x and y such that $z = (x, y)$.

From the definition of R, we can obtain the notions of **domain** and **range**. The **domain** is a set defined as:

$$\text{dom } R = \{x: \text{for some } y \, (x \, R \, y)\}$$

This reads as: the values of x such that for at least one element of y, x has a relation with y.

The **range** is a set defined as:

$$\text{ran } R = \{y: \text{for some } x \, (x \, R \, y)\}$$

This reads: the set formed by the values of y such that at least one element of x, x has a relation with y.

Functions

Consider a pair of sets X and Y. We say that a **function** from X to Y is relation such that:

- $dom \, f = X$ and
- such that for each $x \in X$ there is a unique element of $y \in Y$ with $(x, y) \in f$

More informally, we say that a function "*transform*" or "*maps*" or "*sends*" x onto y, and for each "*argument*" x there is a unique value y that f "*assummes*" or "*takes*".

We typically denote a relation or function or transformation or mapping from X onto Y as:

$$f: X \rightarrow Y$$

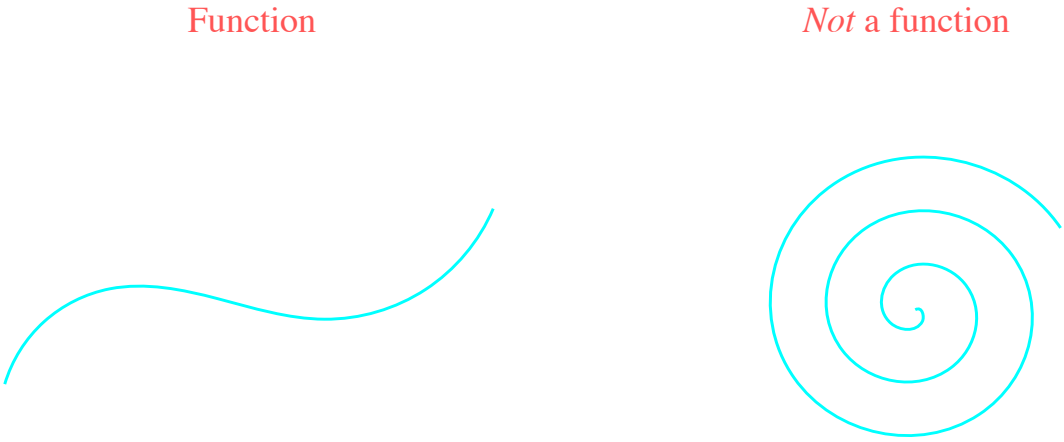
or

$$f(x) = y$$

The simples way to see the effect of this definition of a function is with a chart. In **Fig. 1**, the left-pane shows a valid function, i.e., each value $f(x)$ *maps* uniquely onto one value of y. The right-

pane is not a function, since each value $f(x)$ *maps* onto multiple values of y.

Fig. 1: Functions



For $f: X \rightarrow Y$, the *domain* of f equals to X, but the *range* does not necessarily equals to Y. Just recall that the *range* includes only the elements for which Y has a relation with X.

The ultimate goal of machine learning is learning functions from data, i.e., transformations or mappings from the *domain* onto the *range* of a function. This may sound simplistic, but it's true. The *domain* X is usually a vector (or set) of *variables* or *features* mapping onto a vector of *target* values. Finally, I want to emphasize that in machine learning the words transformation and mapping are used interchangeably, but both just mean function.

This is all I'll cover about sets and functions. My goals were just to introduce: (1) **the concept of a set**, (2) **basic set notation**, (3) **how sets are generated**, (4) **how sets allow the definition of functions**, (5) **the concept of a function**. Set theory is a monumental field, but there is no need to learn everything about sets to understand linear algebra. Halmo's **Naive set theory** (not free, but you can find a copy for ~$8-$10 US) is a fantastic book for people that just need to understand the most fundamental ideas in a relatively informal manner.

```
# Libraries for this section
import numpy as np
import pandas as pd
import altair as alt
alt.themes.enable('dark')
```

```
ThemeRegistry.enable('dark')
```

Vectors

Linear algebra is the study of vectors. At the most general level, vectors are **ordered finite lists of numbers**. Vectors are the most fundamental mathematical object in machine learning. We use them to **represent attributes of entities**: age, sex, test scores, etc. We represent vectors by a bold lower-case letter like $\mathbf{v}$ or as a lower-case letter with an arrow on top like $\vec{v}$.

Vectors are a type of mathematical object that can be **added together** and/or **multiplied by a number** to obtain another object of **the same kind**. For instance, if we have a vector $\mathbf{x} = \mathbf{age}$ and a second vector $\mathbf{y} = \mathbf{weight}$, we can add them together and obtain a third vector $\mathbf{z} = \mathbf{x} + \mathbf{y}$. We can also multiply $2 \times \mathbf{x}$ to obtain $2\mathbf{x}$, again, a vector. This is what we mean by *the same kind*: the returning object is still a *vector*.

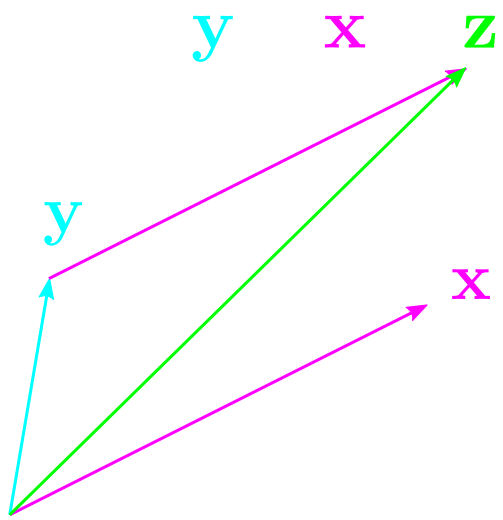
Types of vectors

Vectors come in three flavors: (1) **geometric vectors**, (2) **polynomials**, (3) and **elements of $\mathbf{R^n}$ space**. We will defined each one next.

Geometric vectors

Geometric vectors are oriented segments. Therse are the kind of vectors you probably learned about in high-school physics and geometry. Many linear algebra concepts come from the geometric point of view of vectors: space, plane, distance, etc.

Fig. 2: Geometric vectors

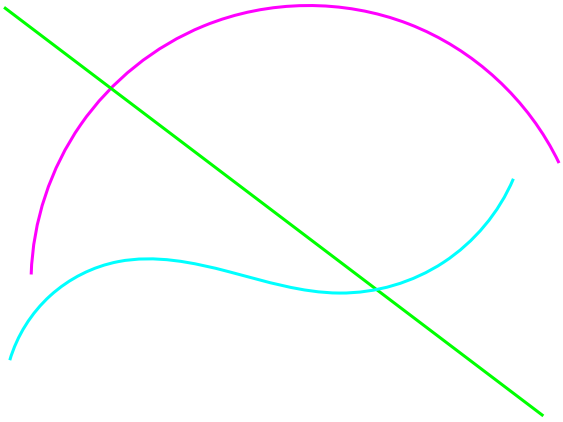


Polynomials

A polynomial is an expression like $f(x) = x^2 + y + 1$. This is, a expression adding multiple "terms" (nomials). Polynomials are vectors because they meet the definition of a vector: they can be added together to get another polynomial, and they can be multiplied together to get another polynomial.

function addition is valid$f(x) + g(x)$ *and* multiplying by a scalar is valid$5 \times f(x)$

Fig. 3: Polynomials



Elements of R

Elements of $\mathbf{R}^n$ are sets of real numbers. This type of representation is arguably the most important for applied machine learning. It is how data is commonly represented in computers to build machine learning models. For instance, a vector in R^3 takes the shape of:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \in \mathrm{R}^3$$

Indicating that it contains three dimensions.

addition is valid $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$ *and*

multiplying by a scalar is valid $5 \times \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 10 \\ 15 \end{bmatrix}$

In NumPy vectors are represented as n-dimensional arrays. To create

a vector in R^3:

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

We can inspect the vector shape by:

```
x.shape # (3 dimensions, 1 element on each)
```

```
(3, 1)
```

```
print(f'A 3-dimensional vector:\n{x}')
```

```
A 3-dimensional vector:
[[1]
 [2]
 [3]]
```

Zero vector, unit vector, and sparse vector

There are a couple of "special" vectors worth to remember as they will be mentioned frequently on applied linear algebra: (1) zero vector, (2) unit vector, (3) sparse vectors

Zero vectors, are vectors composed of zeros, and zeros only. It is common to see this vector denoted as simply 0, regardless of the dimensionality. Hence, you may see a 3-dimensional or 10-dimensional with all entries equal to 0, refered as "the 0" vector. For instance:

$$\mathbf{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Unit vectors, are vectors composed of a single element equal to one, and the rest to zero. Unit vectors are important to understand

applications like norms. For instance, $\mathbf{x_1}$, $\mathbf{x_2}$, and $\mathbf{x_3}$ are unit vectors:

$$\mathbf{x_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \mathbf{x_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \mathbf{x_3} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Sparse vectors, are vectors with most of its elements equal to zero. We denote the number of nonzero elements of a vector $\mathbf{x}$ as $nnz(x)$. The sparser possible vector is the zero vector. Sparse vectors are common in machine learning applications and often require some type of method to deal with them effectively.

Vector dimensions and coordinate system

Vectors can have any number of dimensions. The most common are the 2-dimensional cartesian plane, and the 3-dimensional space. Vectors in 2 and 3 dimensions are used often for pedgagogical purposes since we can visualize them as geometric vectors. Nevetheless, most problems in machine learning entail more dimensions, sometiome hundreds or thousands of dimensions. The notation for a vector $\mathbf{x}$ of arbitrary dimensions, n is:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathrm{R}^{\mathbf{n}}$$

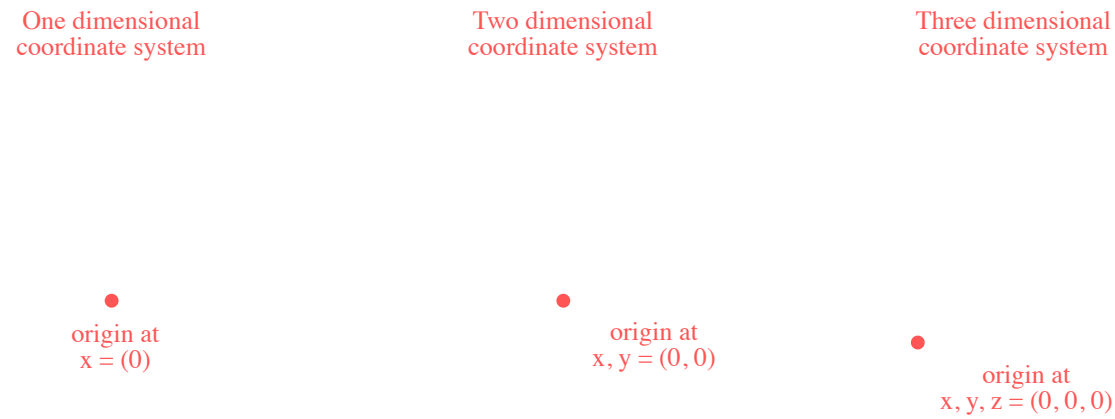
Vectors dimensions map into **coordinate systems or perpendicular axes**. Coordinate systems have an origin at $(0, 0, 0)$, hence, when we define a vector:

$$\mathbf{x} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \in \mathrm{R}^{\mathbf{3}}$$

we are saying: starting from the origin, move 3 units in the 1st

perpendicular axis, 2 units in the 2nd perpendicular axis, and 1 unit in the 3rd perpendicular axis. We will see later that when we have a set of perpendicular axes we obtain the basis of a vector space.

Fig. 4: Coordinate systems



Basic vector operations

Vector-vector addition

We used vector-vector addition to define vectors without defining vector-vector addition. Vector-vector addition is an element-wise operation, only defined for vectors of the same size (i.e., number of elements). Consider two vectors of the same size, then:

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

For instance:

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 + 1 \\ 2 + 2 \\ 3 + 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

Vector addition has a series of **fundamental properties** worth

mentioning:

1. Commutativity: $x + y = y + x$
2. Associativity: $(x + y) + z = x + (y + z)$
3. Adding the zero vector has no effect: $x + 0 = 0 + x = x$
4. Substracting a vector from itself returns the zero vector: $x - x = 0$

In NumPy, we add two vectors of the same with the + operator or the add method:

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

```
x + y
```

```
array([[2],
       [4],
       [6]])
```

```
np.add(x,y)
```

```
array([[2],
       [4],
       [6]])
```

Vector-scalar multiplication

Vector-scalar multiplication is an element-wise operation. It's defined as:

$$\alpha \mathbf{x} = \begin{bmatrix} \alpha \mathbf{x_1} \\ \vdots \\ \alpha \mathbf{x_n} \end{bmatrix}$$

Consider $\alpha = 2$ and $\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$:

$$\alpha\mathbf{x} = \begin{bmatrix} 2 \times 1 \\ 2 \times 2 \\ 2 \times 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

Vector-scalar multiplication satisfies a series of important properties:

1. Associativity: $(\alpha\beta)\mathbf{x} = \alpha(\beta\mathbf{x})$
2. Left-distributive property: $(\alpha + \beta)\mathbf{x} = \alpha\mathbf{x} + \beta\mathbf{x}$
3. Right-distributive property: $\mathbf{x}(\alpha + \beta) = \mathbf{x}\alpha + \mathbf{x}\beta$
4. Right-distributive property for vector addition: $\alpha(\mathbf{x} + \mathbf{y}) = \alpha\mathbf{x} + \alpha\mathbf{y}$

In NumPy, we compute scalar-vector multiplication with the $*$ operator:

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


alpha * x
```

```
array([[2],
       [4],
       [6]])
```

Linear combinations of vectors

There are only two legal operations with vectors in linear algebra: **addition** and **multiplication by numbers**. When we combine those, we get a **linear combination**.

$$\alpha\mathbf{x} + \beta\mathbf{y} = \alpha\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \beta\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \alpha x_1 + \alpha x_2 \\ \beta y_1 + \beta y_2 \end{bmatrix}$$

Consider $\alpha = 2$, $\beta = 3$, $\mathbf{x} = \begin{bmatrix} 2 & 3 \end{bmatrix}$, and $\begin{bmatrix} 4 & 5 \end{bmatrix}$.

We obtain:

$$\alpha\mathbf{x} + \beta\mathbf{y} = \mathbf{2}\begin{bmatrix}2\\3\end{bmatrix} + \mathbf{3}\begin{bmatrix}4\\5\end{bmatrix} = \begin{bmatrix}2\times 2 + 2\times 4\\2\times 3 + 3\times 5\end{bmatrix} = \begin{bmatrix}10\\21\end{bmatrix}$$

Another way to express linear combinations you'll see often is with summation notation. Consider a set of vectors $x_1, \ldots, x_k$ and scalars $\beta_1, \ldots, \beta_k \in \mathrm{R}$, then:

$$\sum_{i=1}^{k} \beta_i x_i := \beta_1 x_1 + \ldots + \beta_k x_k$$

Note that $:=$ means "*is defined as*".

Linear combinations are the most fundamental operation in linear algebra. Everything in linear algebra results from linear combinations. For instance, linear regression is a linear combination of vectors. Fig. 2 shows an example of how adding two geometrical vectors looks like for intuition.

In NumPy, we do linear combinations as:

```
a, b = 2, 3
x , y = np.array([[2],[3]]), np.array([[4], [5]])
```

```
a*x + b*y
```

```
array([[16],
       [21]])
```

Vector-vector multiplication: dot product

We covered vector addition and multiplication by scalars. Now I will define vector-vector multiplication, commonly known as a **dot product** or **inner product**. The dot product of $\mathbf{x}$ and $\mathbf{y}$ is defined as:

$$\mathbf{x}\cdot\mathbf{y} := \begin{bmatrix}x_1\\x_2\end{bmatrix}^{\mathbf{T}}\begin{bmatrix}y_1\\y_2\end{bmatrix} = \begin{bmatrix}x_1 & x_2\end{bmatrix}\begin{bmatrix}y_1\\y_2\end{bmatrix} = \mathbf{x_1}\times\mathbf{y_1} + \mathbf{x_2}\times\mathbf{y_2}$$

Where the T superscript denotes the transpose of the vector.

Transposing a vector just means to "flip" the column vector to a row vector counterclockwise. For instance:

$$\mathbf{x} \cdot \mathbf{y} = \begin{bmatrix} -2 \\ 2 \end{bmatrix} \begin{bmatrix} 4 \\ -3 \end{bmatrix} = \begin{bmatrix} -2 & 2 \end{bmatrix} \begin{bmatrix} 4 \\ -3 \end{bmatrix} = -2 \times 4 + 2 \times -3 = (-8) + (-6) = -14$$

Dot products are so important in machine learning, that after a while they become second nature for practitioners.

To multiply two vectors with dimensions (rows=2, cols=1) in Numpy, we need to transpose the first vector at using the @ operator:

```
x, y = np.array([[-2],[2]]), np.array([[4],[-3]])
```

```
x.T @ y
```

```
array([[-14]])
```

Vector space, span, and subspace

Vector space

In its more general form, a **vector space**, also known as **linear space**, is a collection of objects that follow the rules defined for vectors in R^n. We mentioned those rules when we defined vectors: they can be added together and multiplied by scalars, and return vectors of the same type. More colloquially, a vector space is the set of proper vectors and all possible linear combinatios of the vector set. In addition, vector addition and multiplication must follow these eight rules:

1. commutativity: $x + y = y + x$
2. associativity: $x + (y + x) = (y + x) + z$
3. unique zero vector such that: $x + 0 = x \ \forall \ x$
4. $\forall \ x$ there is a unique vector x such that $x + -x = 0$
5. identity element of scalar multiplication: $1x = x$
6. distributivity of scalar multiplication w.r.t vector addition: $x(y + z) = xz + zy$
7. $x(yz) = (xy)z$

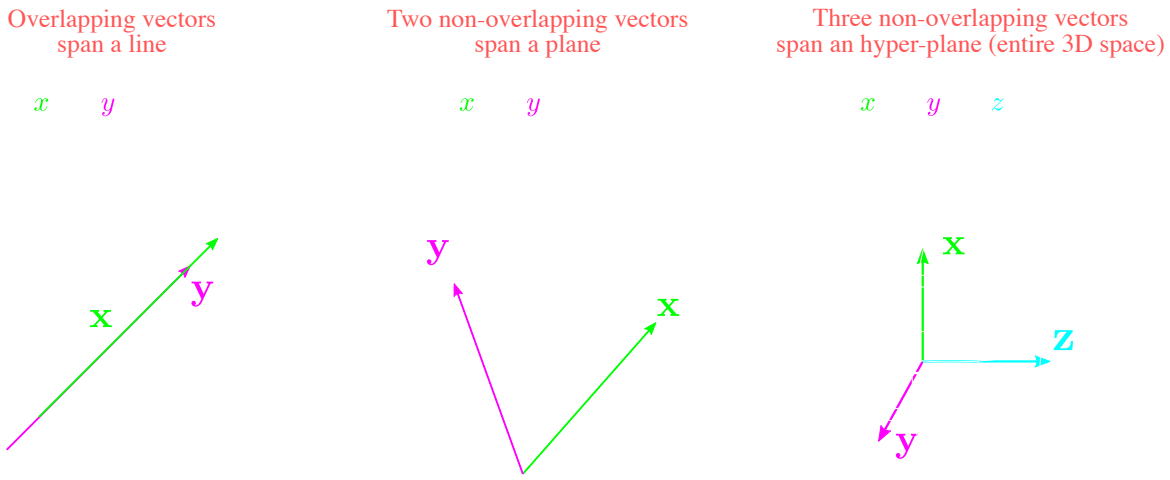
8. $(y + z)x = yx + zx$

In my experience remembering these properties is not really important, but it's good to know that such rules exist.

Vector span

Consider the vectors $\mathbf{x}$ and $\mathbf{y}$ and the scalars α and β. If we take *all* possible linear combinations of $\alpha\mathbf{x} + \beta\mathbf{y}$ we would obtain the **span** of such vectors. This is easier to grasp when you think about geometric vectors. If our vectors $\mathbf{x}$ and $\mathbf{y}$ point into **different directions** in the 2-dimensional space, we get that the $span(x, y)$ is equal to **the entire 2-dimensional plane**, as shown in the middle-pane in **Fig. 5**. Just imagine having an unlimited number of two types of sticks: one pointing vertically, and one pointing horizontally. Now, you can reach any point in the 2-dimensional space by simply combining the necessary number of vertical and horizontal sticks (including taking fractions of sticks).

Fig. 5: Vector Span



What would happen if the vectors point in the same direction? Now, if you combine them, you just can **span a line**, as shown in the left-pane in **Fig. 5**. If you have ever heard of the term "multicollinearity", it's closely related to this issue: when two variables are "colinear" they are pointing in the same direction, hence they provide redundant information, so can drop one without information loss.

With three vectors pointing into different directions, we can span the entire 3-dimensional space or a **hyper-plane**, as in the right-pane of **Fig. 5**. Note that the sphere is just meant as a 3-D reference, not as a limit.

Four vectors pointing into different directions will span the 4-dimensional space, and so on. From here our geometrical intuition can't help us. This is an example of how linear algebra can describe the behavior of vectors beyond our basics intuitions.

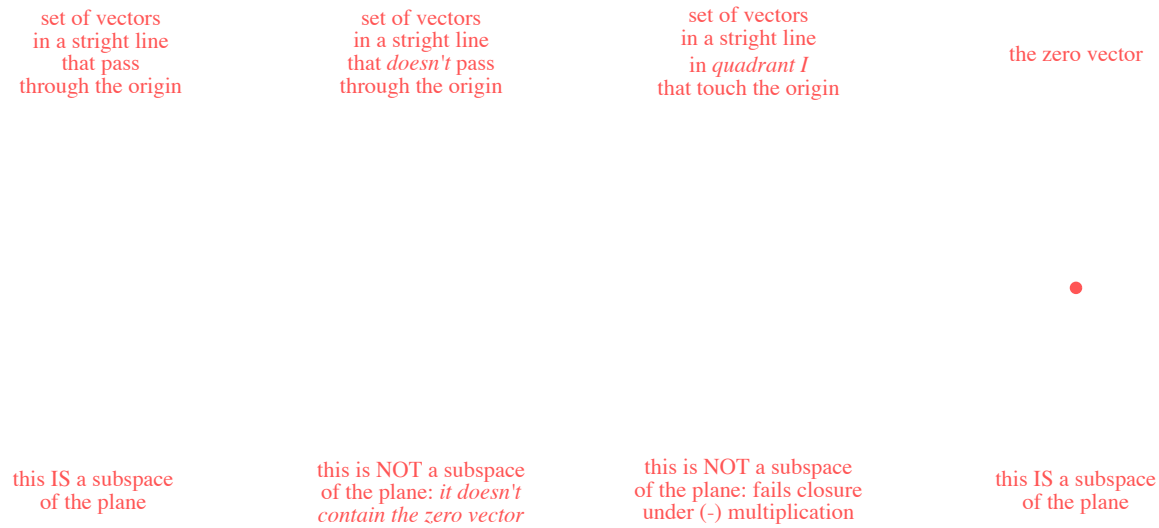
Vector subspaces

A vector subspace (or linear subspace) is a vector space that lies within a larger vector space. These are also known as linear subspaces. Consider a subspace S. For a vector to be a valid subspace it has to meet **three conditions**:

1. Contains the zero vector, $\mathbf{0} \in \mathbf{S}$
2. Closure under multiplication, $\forall \alpha \in \mathrm{R} \rightarrow \alpha \times s_i \in S$
3. Closure under addition, $\forall s_i \in S \rightarrow s_1 + s_2 \in S$

Intuitively, you can think in closure as being unable to "jump out" from space into another. A pair of vectors laying flat in the 2-dimensional space, can't, by either addition or multiplication, "jump out" into the 3-dimensional space.

Fig. 6: Vector subspaces



Consider the following questions: Is $\mathbf{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}$ a valid subspace of R^2

? Let's evaluate $\mathbf{x}$ on the three conditions:

Contains the zero vector: it does. Remember that the span of a vector are all linear combinations of such a vector. Therefore, we can simply multiply by 0 to get $\begin{bmatrix}0 & 0\end{bmatrix}$:

$$\mathbf{x} \times \mathbf{0} = \mathbf{0}\begin{bmatrix}1 \\ 1\end{bmatrix} = \begin{bmatrix}0 \\ 0\end{bmatrix}$$

Closure under multiplication implies that if take any vector belonging to $\mathbf{x}$ and multiply by any real scalar α, the resulting vector stays within the span of $\mathbf{x}$. Algebraically is easy to see that we can multiply $\begin{bmatrix}1 & 1\end{bmatrix}$ by any scalar α, and the resulting vector remains in the 2-dimensional plane (i.e., the span of R^2).

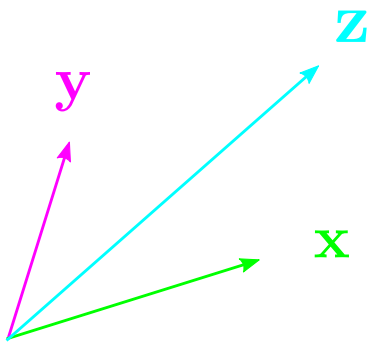
Closure under addition implies that if we add together any vectors belonging to $\mathbf{x}$, the resulting vector remains within the span of R^2. Again, algebraically is clear that if we add $\mathbf{x} + \mathbf{x}$, the resulting vector will remain in R^2. There is no way to get to R^3 or R^4 or any space outside the two-dimensional plane by adding $\mathbf{x}$ multiple times.

Linear dependence and independence

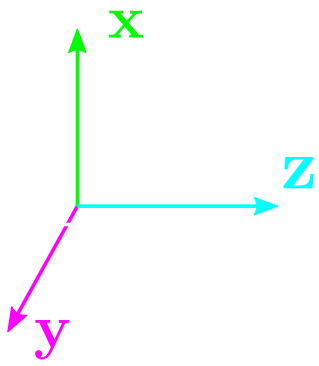
The left-pane shows a triplet of **linearly dependent** vectors, whereas the right-pane shows a triplet of **linearly independent** vectors.

Fig. 7: Linear dependence and independence

Linearly *dependent* vectors
(x + y = z)



Linearly *independent* vectors



A set of vectors is **linearly dependent** if at least one vector can be obtained as a linear combination of other vectors in the set. As you can see in the left pane, we can combine vectors x and y to obtain z.

There is more rigurous (but slightly harder to grasp) definition of linear dependence. Consider a set of vectors $x_1, \ldots, x_k$ and scalars $\beta \in \mathrm{R}$. If there is a way to get $0 = \sum_{i=1}^{k} \beta_i x_i$ with at least one $\beta \neq 0$, we have linearly dependent vectors. In other words, if we can get the zero vector as a linear combination of the vectors in the set, with weights that *are not* all zero, we have a linearly dependent set.

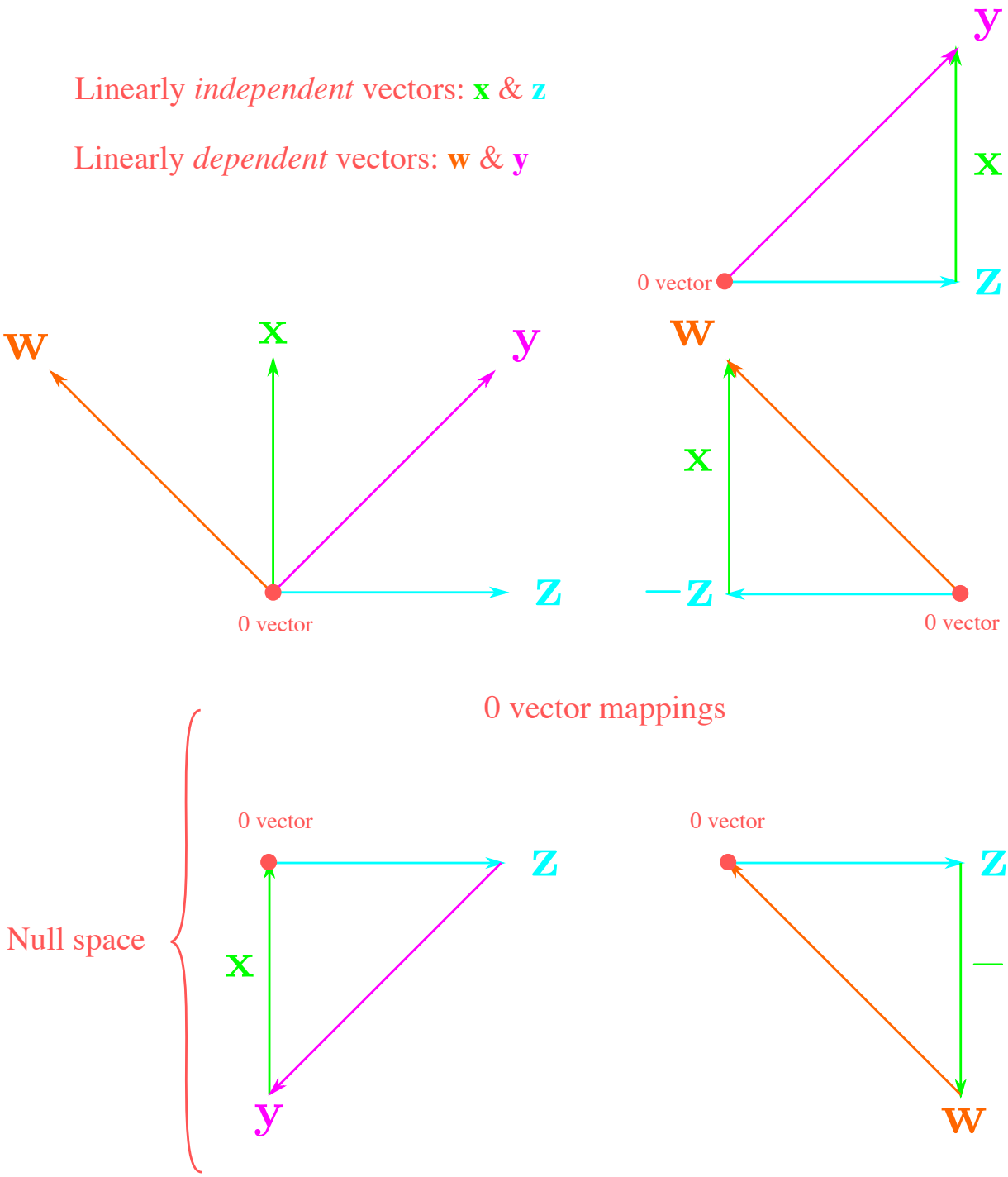
A set of vectors is **linearly independent** if none vector can be obtained as a linear combination of other vectors in the set. As you can see in the right pane, there is no way for us to combine vectors x and y to obtain z. Again, consider a set of vectors $x_1, \ldots, x_k$ and scalars $\beta \in \mathrm{R}$. If the only way to get $0 = \sum_{i=1}^{k} \beta_i x_i$ requires all $\beta_1, \ldots, \beta_k = 0$, the we have linearly independent vectors. In words, the only way to get the zero vectors in by multoplying each vector in the set by 0.

The importance of the concepts of linear dependence and independence will become clearer in more advanced topics. For now, the important points to remember are: linearly dependent vectors contain **redundant information**, whereas linearly independent vectors do not.

Vector null space

Now that we know what subspaces and linear dependent vectors are, we can introduce the idea of the **null space**. Intuitively, the null space of a set of vectors are **all linear combinations that "map" into the zero vector**. Consider a set of geometric vectors $\mathbf{w}$, $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ as in **Fig. 8**. By inspection, we can see that vectors $\mathbf{x}$ and $\mathbf{z}$ are parallel to each other, hence, independent. On the contrary, vectors $\mathbf{w}$ and $\mathbf{y}$ can be obtained as linear combinations of $\mathbf{x}$ and $\mathbf{z}$, therefore, dependent.

Fig. 8: Vector null space



As result, with this four vectors, we can form the following two combinations that will "map" into the origin of the coordinate system, this is, the zero vector $(0, 0)$:

$$z - y + x = 0$$
$$z - x + w = 0$$

We will see how this idea of the null space extends naturally in the context of matrices later.

Vector norms

Measuring vectors is another important operation in machine learning applications. Intuitively, we can think about the **norm** or the **length** of a vector as the distance between its "origin" and its "end".

Norms "map" vectors to non-negative values. In this sense are functions that assign length $\|\mathbf{x}\| \in \mathrm{R}^n$ to a vector $\mathbf{x}$. To be valid, a norm has to satisfy these properties (keep in mind these properties are a bit abstruse to understand):

1. **Absolutely homogeneous**: $\forall \alpha \in \mathrm{R}, \|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$. In words: for all real-valued scalars, the norm scales proportionally with the value of the scalar.
2. **Triangle inequality**: $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$. In words: in geometric terms, for any triangle the sum of any two sides must be greater or equal to the lenght of the third side. This is easy to see experimentally: grab a piece of rope, form triangles of different sizes, measure all the sides, and test this property.
3. **Positive definite**: $\|\mathbf{x}\| \geq \mathbf{0}$ and $\|\mathbf{x}\| = \mathbf{0} \iff \mathbf{x} = \mathbf{0}$. In words: the length of any $\mathbf{x}$ has to be a positive value (i.e., a vector can't have negative length), and a length of 0 occurs only of $\mathbf{x} = \mathbf{0}$

Grasping the meaning of these three properties may be difficult at this point, but they probably become clearer as you improve your understanding of linear algebra.

Fig. 9: Vector norms

Euclidean norm

The Euclidean norm is one of the most popular norms in machine learning. It is so widely used that sometimes is refered simply as "the norm" of a vector. Is defined as:

$$\|\mathbf{x}\|_2 := \sqrt{\sum_{i=1}^{n} \mathbf{x}_i^2} = \sqrt{\mathbf{x}^T\mathbf{x}}$$

Hence, in **two dimensions** the L_2 norm is:

$$\|\mathbf{x}\|_2 \in \mathrm{R}^2 = \sqrt{\mathbf{x}_1^2 \cdot \mathbf{x}_2^2}$$

Which is equivalent to the formula for the hypotenuse a triangle with sides x_1^2 and x_2^2.

The same pattern follows for higher dimensions of R^n

In NumPy, we can compute the L_2 norm as:

```
x = np.array([[3],[4]])

np.linalg.norm(x, 2)
```

```
5.0
```

If you remember the first "Pythagorean triple", you can confirm that the norm is correct.

Manhattan norm

The Manhattan or L_1 norm gets its name in analogy to measuring distances while moving in Manhattan, NYC. Since Manhattan has a grid-shape, the distance between any two points is measured by moving in vertical and horizontals lines (instead of diagonals as in the Euclidean norm). It is defined as:

$$\|\mathbf{x}\|_1 := \sum_{i=1}^{n} |\mathbf{x}_i|$$

Where $|x_i|$ is the absolute value. The L_1 norm is preferred when discriminating between elements that are exactly zero and elements that are small but not zero.

In NumPy we compute the L_1 norm as

```
x = np.array([[3],[-4]])

np.linalg.norm(x, 1)
```

```
7.0
```

Is easy to confirm that the sum of the absolute values of 3 and -4 is 7.

Max norm

The max norm or infinity norm is simply the absolute value of the largest element in the vector. It is defined as:

$$\|\mathbf{x}\|_\infty := \max_i |\mathbf{x_i}|$$

Where $|x_i|$ is the absolute value. For instance, for a vector with elements $\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$, the $\|\mathbf{x}\|_\infty = \mathbf{3}$

In NumPy we compute the L_∞ norm as:

```
x = np.array([[3],[-4]])

np.linalg.norm(x, np.inf)
```

```
4.0
```

Vector inner product, length, and distance.

For practical purposes, inner product and length are used as equivalent to dot product and norm, although technically are not the same.

Inner products are a more general concept that dot products, with a series of additional properties (see here). In other words, every dot product is an inner product, but not every inner product is a dot product. The notation for the inner product is usually a pair of angle brackets as $\langle .,. \rangle$ as. For instance, the scalar inner product is defined as:

$$\langle x, y \rangle := x \cdot y$$

In R^n the inner product is a dot product defined as:

$$\left\langle \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \right\rangle := x \cdot y = \sum_{i=1}^{n} x_i y_i$$

Length is a concept from geometry. We say that geometric vectors have length and that vectors in R^n have norm. In practice, many machine learning textbooks use these concepts interchangeably. I've found authors saying things like "we use the l_2 norm to compute the *length* of a vector". For instance, we can compute the length of a directed segment (i.e., geometrical vector) $\mathbf{x}$ by taking the square root of the inner product with itself as:

$$\|x\| = \sqrt{\langle x, x \rangle} = \sqrt{x \cdot y} = x^2 + y^2$$

Distance is a relational concept. It refers to the length (or norm) of the difference between two vectors. Hence, we use norms and lengths to measure the distance between vectors. Consider the vectors $\mathbf{x}$ and $\mathbf{y}$, we define the distance $d(x, y)$ as:

$$d(x, y) := \|x - y\| = \sqrt{\langle x - y, x - y \rangle}$$

When the inner product $\langle x - y, x - y \rangle$ is the dot product, the distance equals to the Euclidean distance.

In machine learning, unless made explicit, we can safely assume that an inner product refers to the dot product. We already reviewed how to compute the dot product in NumPy:

```
x, y = np.array([[-2],[2]]), np.array([[4],[-3]])
x.T @ y
```

```
array([[-14]])
```

As with the inner product, usually, we can safely assume that **distance** stands for the Euclidean distance or L_2 norm unless otherwise noted. To compute the L_2 distance between a pair of vectors:

```
distance = np.linalg.norm(x-y, 2)
print(f'L_2 distance : {distance}')
```

```
L_2 distance : 7.810249675906656
```

Vector angles and orthogonality

The concepts of angle and orthogonality are also related to geometrical vectors. We saw that inner products allow for the definition of length and distance. In the same manner, inner products are used to define **angles** and **orthogonality**.

In machine learning, the **angle** between a pair of vectors is used as a **measure of vector similarity**. To understand angles let's first look at the **Cauchy-Schwarz inequality**. Consider a pair of non-zero vectors $\mathbf{x}$ and $\mathbf{y} \in \mathrm{R}^n$. The Cauchy-Schwarz inequality states that:

$$|\langle x, y \rangle| \leq \|x\| \|y\|$$

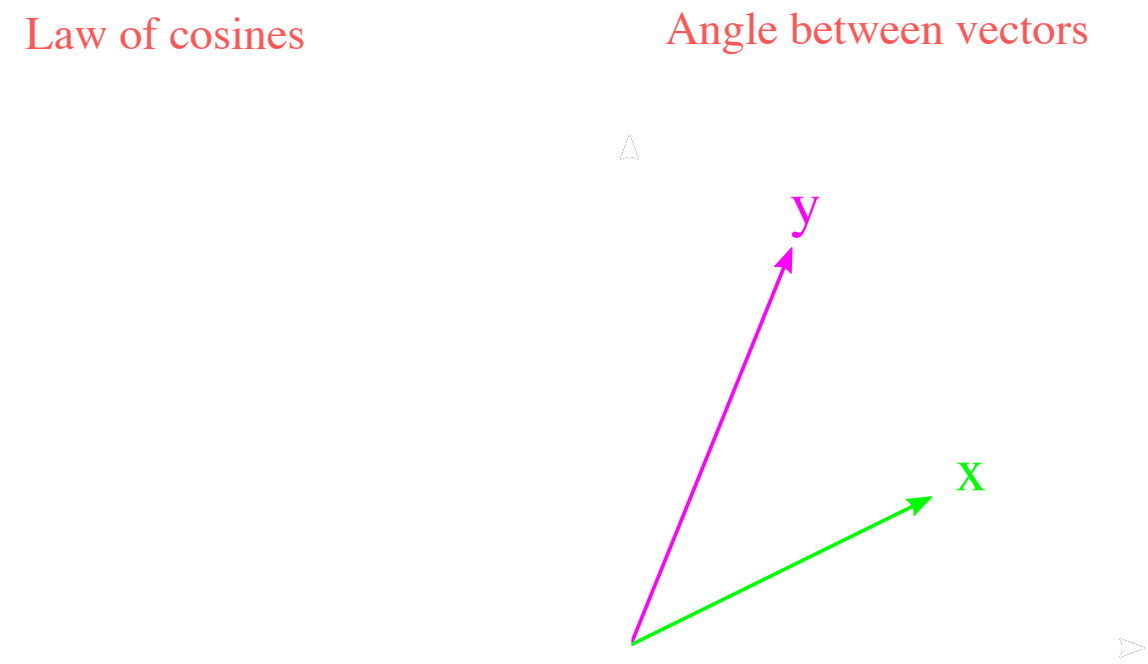
In words: *the absolute value of the inner product of a pair of vectors is less than or equal to the products of their length*. The only case where both sides of the expression are *equal* is when vectors are colinear, for instance, when $\mathbf{x}$ is a scaled version of $\mathbf{y}$. In the 2-dimensional case, such vectors would lie along the same line.

The definition of the angle between vectors can be thought as a generalization of the **law of cosines** in trigonometry, which defines for a triangle with sides a, b, and c, and an angle θ are related

as:

$$c^2 = a^2 + b^2 - 2ab\cos\theta$$

Fig. 10: Law of cosines and Angle between vectors



We can replace this expression with vectors lengths as:

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2(\|x\|\|y\|)\cos\theta$$

With a bit of algebraic manipulation, we can clear the previous equation to:

$$\cos\theta = \frac{\langle x, y\rangle}{\|x\|\|y\|}$$

And there we have a **definition for (cos) angle θ**. Further, from the Cauchy–Schwarz inequality we know that $\cos\theta$ must be:

$$-1 \leq \frac{\langle x, y\rangle}{\|x\|\|y\|} \leq 1$$

This is a necessary conclusion (range between $-1, 1$) since the numerator in the equation always is going to be smaller or equal to

the denominator.

In NumPy, we can compute the $\cos\theta$ between a pair of vectors as:

```
x, y = np.array([[1], [2]]), np.array([[5], [7]])

# here we translate the cos(theta) definition
cos_theta = (x.T @ y) / (np.linalg.norm(x,2) * np.linalg.norm(y,2)
print(f'cos of the angle = {np.round(cos_theta, 3)}')
```

```
cos of the angle = [[0.988]]
```

We get that $\cos\theta \approx 0.988$. Finally, to know the exact value of θ we need to take the trigonometric inverse of the cosine function as:

```
cos_inverse = np.arccos(cos_theta)
print(f'angle in radiants = {np.round(cos_inverse, 3)}')
```

```
angle in radiants = [[0.157]]
```

We obtain $\theta \approx 0.157$. To fo from radiants to degrees we can use the following formula:

```
degrees = cos_inverse * ((180)/np.pi)
print(f'angle in degrees = {np.round(degrees, 3)}')
```

```
angle in degrees = [[8.973]]
```

We obtain $\theta \approx 8.973^\circ$

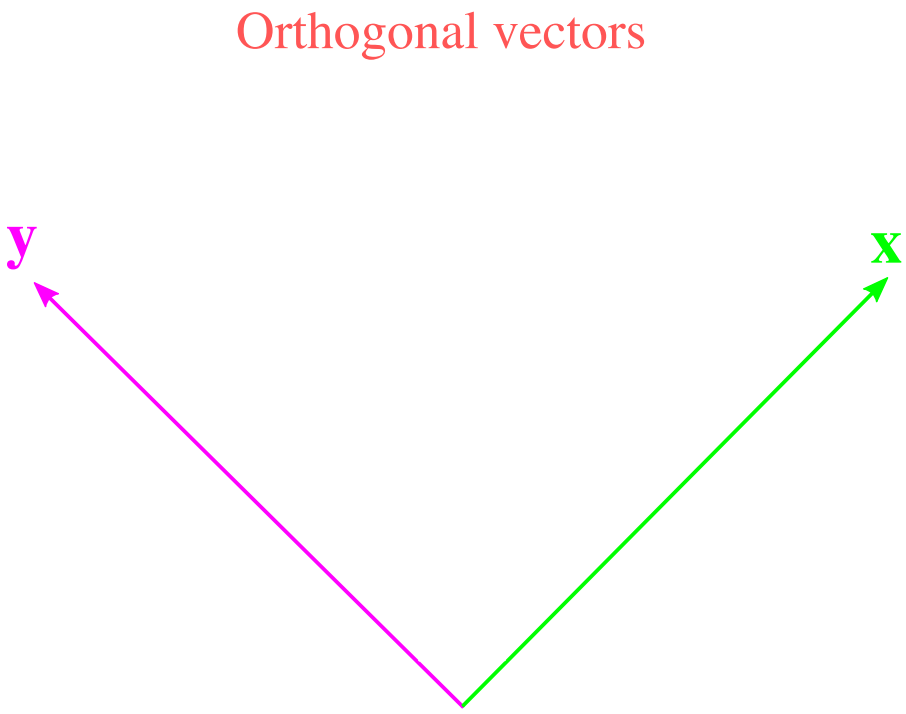
Orthogonality is often used interchangeably with "independence" although they are mathematically different concepts. Orthogonality can be seen as a generalization of perpendicularity to vectors in any number of dimensions.

We say that a pair of vectors $\mathbf{x}$ and $\mathbf{y}$ are **orthogonal** if their inner product is zero, $\langle x, y \rangle = 0$. The notation for a pair of orthogonal vectors is $\mathbf{x} \perp \mathbf{y}$. In the 2-dimensional plane, this equals to a pair

of vectors forming a $90°$ angle.

Here is an example of orthogonal vectors

Fig. 11: Orthogonal vectors




```
x = np.array([[2], [0]])
y = np.array([[0], [2]])

cos_theta = (x.T @ y) / (np.linalg.norm(x,2) * np.linalg.norm(y,2)
print(f'cos of the angle = {np.round(cos_theta, 3)}')
```

```
cos of the angle = [[0.]]
```

We see that this vectors are **orthogonal** as $\cos\theta = 0$. This is equal to ≈ 1.57 radiants and $\theta = 90°$

```
cos_inverse = np.arccos(cos_theta)
degrees = cos_inverse * ((180)/np.pi)
print(f'angle in radiants = {np.round(cos_inverse, 3)}\nangle in d
```

```
angle in radiants = [[1.571]]
angle in degrees =[[90.]]
```

Systems of linear equations

The purpose of linear algebra as a tool is to **solve systems of linear equations**. Informally, this means to figure out the right combination of linear segments to obtain an outcome. Even more informally, think about making pancakes: In what proportion ($w_i \in \mathrm{R}$) we have to mix ingredients to make pancakes? You can express this as a linear equation:

$$f_{\text{flour}} \times w_1 + b_{\text{baking powder}} \times w_2 + e_{\text{eggs}} \times w_3 + m_{\text{milk}} \times w_4 = P_{\text{pancakes}}$$

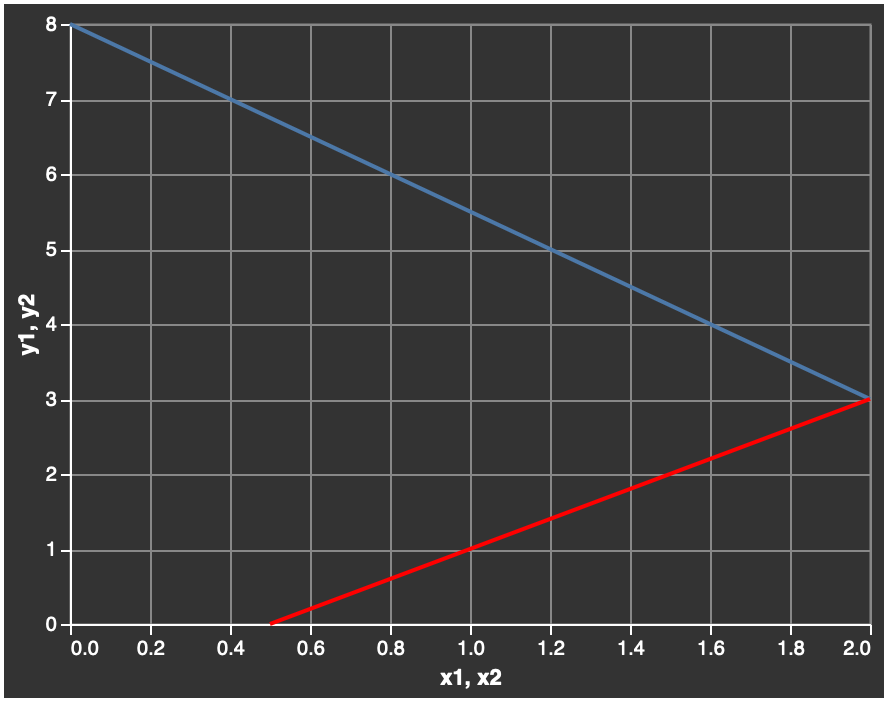
The above expression describe *a* linear equation. A *system* of linear equations involve multiple equations that have to be solved **simultaneously**. Consider:

$$x + 2y = 85x - 3y = 1$$

Now we have a system with two unknowns, x and y. We'll see general methods to solve systems of linear equations later. For now, I'll give you the answer: $x = 2$ and $y = 3$. Geometrically, we can see that both equations produce a straight line in the 2-dimensional plane. The point on where both lines encounter is the solution to the linear system.

```
df = pd.DataFrame({"x1": [0, 2], "y1":[8, 3], "x2": [0.5, 2], "y2"
```

```
equation1 = alt.Chart(df).mark_line().encode(x="x1", y="y1")
equation2 = alt.Chart(df).mark_line(color="red").encode(x="x2", y=
equation1 + equation2
```



Matrices

```
# Libraries for this section
import numpy as np
import pandas as pd
import altair as alt
```

Matrices are as fundamental as vectors in machine learning. With vectors, we can represent single variables as sets of numbers or instances. With matrices, we can represent sets of variables. In this sense, a matrix is simply an ordered **collection of vectors**. Conventionally, column vectors, but it's always wise to pay attention to the authors' notation when reading matrices. Since computer screens operate in two dimensions, matrices are the way in which we interact with data in practice.

More formally, we represent a matrix with a italicized upper-case letter like A. In two dimensions, we say the matrix A has m rows and n columns. Each entry of A is defined as a_{ij}, $i = 1, \ldots, m,$ and $j = 1, \ldots, n$. A matrix $A \in R^{m \times n}$ is defines as:

$$A := \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, a_{ij} \in \mathrm{R}$$

In Numpy, we construct matrices with the array method:

```
A = np.array([[0,2],  # 1st row
              [1,4]]) # 2nd row

print(f'a 2x2 Matrix:\n{A}')
```

```
a 2x2 Matrix:
[[0 2]
 [1 4]]
```

Basic Matrix operations

Matrix-matrix addition

We add matrices in a element-wise fashion. The sum of $\mathrm{A} \in \mathrm{R}^{m \times n}$ and $\mathrm{B} \in \mathrm{R}^{m \times n}$ is defined as:

$$\mathrm{A} + \mathrm{B} := \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{bmatrix} \in \mathrm{R}^{\mathrm{m} \times \mathrm{n}}$$

For instance: $\mathrm{A} = \begin{bmatrix} 0 & 2 \\ 1 & 4 \end{bmatrix} + \mathrm{B} = \begin{bmatrix} 3 & 1 \\ -3 & 2 \end{bmatrix} = \begin{bmatrix} 0+3 & 2+1 \\ 3+(-3) & 2+2 \end{bmatrix} = \begin{bmatrix} 3 & 3 \\ -2 & 6 \end{bmatrix}$

In Numpy, we add matrices with the + operator or add method:

```
A = np.array([[0,2],
              [1,4]])
B = np.array([[3,1],
```

```
              [-3,2]])
```

```
A + B
```

```
array([[ 3,  3],
       [-2,  6]])
```

```
np.add(A, B)
```

```
array([[ 3,  3],
       [-2,  6]])
```

Matrix-scalar multiplication

Matrix-scalar multiplication is an element-wise operation. Each element of the matrix A is multiplied by the scalar α. Is defined as:

$$a_{ij} \times \alpha, \text{ such that } (\alpha A)_{ij} = \alpha(A)_{ij}$$

Consider $\alpha = 2$ and $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, then:

$$\alpha A = 2\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \times 1 & 2 \times 2 \\ 2 \times 3 & 2 \times 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

In NumPy, we compute matrix-scalar multiplication with the $*$ operator or multiply method:

```
alpha = 2
A = np.array([[1,2],
              [3,4]])
```

```
alpha * A
```

```
array([[2, 4],
       [6, 8]])
```

```
np.multiply(alpha, A)
```

```
array([[2, 4],
       [6, 8]])
```

Matrix-vector multiplication: dot product

Matrix-vector multiplication equals to taking the dot product of each column n of a A with each element $\mathbf{x}$ resulting in a vector $\mathbf{y}$. Is defined as:

$$\mathrm{A} \cdot \mathbf{x} := \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \mathbf{x_1} \begin{bmatrix} a_{11} \\ \vdots \\ a_{m1} \end{bmatrix} + \mathbf{x_2} \begin{bmatrix} a_{12} \\ \vdots \\ a_{m2} \end{bmatrix} + \mathbf{x_n} \begin{bmatrix} a_{1n} \\ \vdots \\ a_{mn} \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_{mn} \end{bmatrix}$$

For instance:

$$\mathrm{A} \cdot \mathbf{x} = \begin{bmatrix} 0 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \mathbf{1} \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \mathbf{2} \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \times 0 + 2 \times 2 \\ 1 \times 1 + 2 \times 4 \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix}$$

In numpy, we compute the matrix-vector product with the `@` operator or `dot` method:

```
A = np.array([[0,2],
              [1,4]])
x = np.array([[1],
              [2]])
```

```
A @ x
```

```
array([[4],
       [9]])
```

```
np.dot(A, x)
```

```
array([[4],
       [9]])
```

Matrix-matrix multiplication

Matrix-matrix multiplication is a dot produt as well. To work, the number of columns in the first matrix A has to be equal to the number of rows in the second matrix B. Hence, $A \in R^{m \times n}$ times $B \in R^{n \times p}$ to be valid. One way to see matrix-matrix multiplication is by taking a series of dot products: the 1st column of A times the 1st row of B, the 2nd column of A times the 2nd row of B, until the n_{th} column of A times the n_{th} row of B.

We define $A \in R^{n \times p} \cdot B \in R^{n \times p} = C \in R^{m \times p}$:

$$A \cdot B := \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mp} \end{bmatrix}$$

A compact way to define the matrix-matrix product is:

$$c_{ij} := \sum_{l=1}^{n} a_{il} b_{lj}, \text{ with } i = 1, \ldots m, \text{ and}, j = 1, \ldots, p$$

For instance

$$A \cdot B = \begin{bmatrix} 0 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \times 0 + 2 \times 2 & 3 \times 0 + 1 \times 2 \\ 1 \times 1 + 2 \times 4 & 3 \times 1 + 1 \times 4 \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 9 & 7 \end{bmatrix}$$

Matrix-matrix multiplication has a series of important properties:

- Associativity: $(AB)C = A(BC)$
- Associativity with scalar multiplication: $\alpha(AB) = (\alpha A)B$
- Distributivity with addition: $A(B + C) = A + B + AC$
- Transpose of product: $(AB)^T = B^T A^T$

It's also important to remember that **matrix-matrix multiplication orders matter**, this is, it is **not commutative**. Hence, in general, $AB \neq BA$.

In NumPy, we obtan the matrix-matrix product with the @ operator or dot method:

```
A = np.array([[0,2],
              [1,4]])
B = np.array([[1,3],
              [2,1]])
```

```
A @ B
```

```
array([[4, 2],
       [9, 7]])
```

```
np.dot(A, B)
```

```
array([[4, 2],
       [9, 7]])
```

Matrix identity

An identity matrix is a square matrix with ones on the diagonal from the upper left to the bottom right, and zeros everywhere else. We denote the identity matrix as I_n. We define $\mathrm{I} \in \mathrm{R}^{n \times n}$ as:

$$\mathrm{I}_n := \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \in \mathrm{R}^{n \times n}$$

For example:

$$\mathrm{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

You can think in the inverse as playing the same role than 1 in operations with real numbers. The inverse matrix does not look very interesting in itself, but it plays an important role in some proofs and for the inverse matrix (which can be used to solve system of linear equations).

Matrix inverse

In the context of real numbers, the *multiplicative inverse (or reciprocal)* of a number x, is the number that when multiplied by x yields 1. We denote this by x^{-1} or $\frac{1}{x}$. Take the number 5. Its multiplicative inverse equals to $5 \times \frac{1}{5} = 1$.

If you recall the matrix identity section, we said that the identity plays a similar role than the number one but for matrices. Again, by analogy, we can see the *inverse* of a matrix as playing the same role than the multiplicative inverse for numbers but for matrices. Hence, the *inverse matrix* is a matrix than when multiplies another matrix *from either the right or the left side*, returns the identity matrix.

More formally, consider the square matrix $\mathrm{A} \in \mathrm{R}^{n \times n}$. We define A^{-1} as matrix with the property:

$$\mathrm{A}^{-1}\mathrm{A} = \mathrm{I}_n = \mathrm{A}\mathrm{A}^{-1}$$

The main reason we care about the inverse, is because it allows to **solve systems of linear equations** in certain situations. Consider a system of linear equations as:

$$\mathrm{A}\mathbf{x} = \mathbf{y}$$

Assuming A has an inverse, we can multiply by the inverse on both sides:

$$\mathrm{A}^{-1}\mathrm{A}\mathbf{x} = \mathrm{A}^{-\mathbf{1}}\mathbf{y}$$

And get:

$$\mathrm{I}\mathbf{x} = \mathrm{A}^{-\mathbf{1}}\mathbf{y}$$

Since the I does not affect $\mathbf{x}$ at all, our final expression becomes:

$$\mathbf{x} = \mathrm{A}^{-1}\mathbf{y}$$

This means that we just need to know the inverse of A, multiply by the target vector $\mathbf{y}$, and we obtain the solution for our system. I mentioned that this works only in *certain situations*. By this I meant: **if and only if $\mathbf{A}$ happens to have an inverse**. Not all matrices have an inverse. When A^{-1} exist, we say A is *nonsingular* or *invertible*, otherwise, we say it is *noninvertible* or *singular*.

The lingering question is how to find the inverse of a matrix. We can do it by reducing A to its *reduced row echelon form* by using Gauss-Jordan Elimination. If A has an inverse, we will obtain the identity matrix as the row echelon form of A. I haven't introduced either just yet. You can jump to the *Solving systems of linear equations with matrices* if you are eager to learn about it now. For now, we relie on NumPy.

In NumPy, we can compute the inverse of a matrix with the `.linalg.inv` method:

```
A = np.array([[1, 2, 1],
              [4, 4, 5],
              [6, 7, 7]])
```

```
A_i = np.linalg.inv(A)
print(f'A inverse:\n{A_i}')
```

```
A inverse:
[[-7. -7.  6.]
 [ 2.  1. -1.]
 [ 4.  5. -4.]]
```

We can check the A^{-1} is correct by multiplying. If so, we should obtain the identity I_3

```
I = np.round(A_i @ A)
print(f'A_i times A resulsts in I_3:\n{I}')
```

```
A_i times A resulsts in I_3:
[[ 1.  0.  0.]
 [ 0.  1. -0.]
 [ 0. -0.  1.]]
```

Matrix transpose

Consider a matrix $A \in R^{m \times n}$. The **transpose** of A is denoted as $A^T \in R^{m \times n}$. We obtain A^T as:

$$(A^T)_{ij} = A_j i$$

In other words, we get the A^T by switching the columns by the rows of A. For instance:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

In NumPy, we obtain the transpose with the `T` method:

```
A = np.array([[1, 2],
              [3, 4],
              [5, 6]])
```

```
A.T
```

```
array([[1, 3, 5],
       [2, 4, 6]])
```

Hadamard product

It is tempting to think in matrix-matrix multiplication as an element-wise operation, as multiplying each overlapping element of A and B. *It is not*. Such operation is called **Hadamard product**. I'm introducing this to avoid confusion. The Hadamard product is defined as

$$a_{ij} \cdot b_{ij} := c_{ij}$$

For instance:

$$A \odot B = \begin{bmatrix} 0 & 2 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 0 \times 1 & 2 \times 3 \\ 1 \times 2 & 4 \times 1 \end{bmatrix} = \begin{bmatrix} 0 & 6 \\ 2 & 4 \end{bmatrix}$$

In `numpy`, we compute the Hadamard product with the `*` operator or `multiply` method:

```
A = np.array([[0,2],
              [1,4]])
B = np.array([[1,3],
              [2,1]])


A * B
```

```
array([[0, 6],
       [2, 4]])
```

```
np.multiply(A, B)
```

```
array([[0, 6],
       [2, 4]])
```

Special matrices

There are several matrices with special names that are commonly found in machine learning theory and applications. Knowing these matrices beforehand can improve your linear algebra fluency, so we will briefly review a selection of 12 common matrices. For an extended list of special matrices see here and here.

Rectangular matrix

Matrices are said to be *rectangular* when the number of rows is $\neq$ to the number of columns, i.e., $A^{m \times n}$ with $m \neq n$. For instance:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Square matrix

Matrices are said to be **square** when the number of rows = the number of columns, i.e., $A^{n \times n}$. For instance:

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

Diagonal matrix

Square matrices are said to be **diagonal** when each of its non-diagonal elements is zero, i.e., For $D = (d_{i,j})$, we have $\forall i,j \in n, i \neq j \implies d_{i,j} = 0$. For instance:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{bmatrix}$$

Upper triangular matrix

Square matrices are said to be **upper triangular** when the elements below the main diagonal are zero, i.e., For $D = (d_{i,j})$, we have $d_{i,j} = 0, \text{for } i > j$. For instance:

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

Lower triangular matrix

Square matrices are said to be **lower triangular** when the elements above the main diagonal are zero, i.e., For $D = (d_{i,j})$, we have $d_{i,j} = 0, \text{for } i < j$. For instance:

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

Symmetric matrix

Square matrices are said to be symmetric its equal to its transpose, i.e., $\mathrm{A} = \mathrm{A}^T$. For instance:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 6 \\ 3 & 6 & 1 \end{bmatrix}$$

Identity matrix

A diagonal matrix is said to be the identity when the elements along its main diagonal are equal to one. For instance:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scalar matrix

Diagonal matrices are said to be scalar when all the elements along its main diaonal are equal, i.e., $\mathrm{D} = \alpha\mathrm{I}$. For instance:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Null or zero matrix

Matrices are said to be null or zero matrices when all its elements equal to zero, wich is denoted as $0_{m \times n}$. For instance:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Echelon matrix

Matrices are said to be on **echelon form** when it has undergone the process of Gaussian elimination. More specifically:

>> Zero rows are at the bottom of the matrix
>> The leading entry (pivot) of each nonzero row is to the right of the leading entry of the row above it
>> Each leading entry is the only nonzero entry in its column

For instance:

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 2 & -1 \\ 1 & 3 & 2 \end{bmatrix}$$

In echelon form after Gaussian Elimination becomes:

$$\begin{bmatrix} 1 & 3 & 5 \\ 0 & -4 & -11 \\ 0 & 0 & -3 \end{bmatrix}$$

Antidiagonal matrix

Matrices are said to be **antidiagonal** when all the entries are zero but the antidiagonal (i.e., the diagonal starting from the bottom left corner to the upper right corner). For instance:

$$\begin{bmatrix} 0 & 0 & 3 \\ 0 & 5 & 0 \\ 7 & 0 & 0 \end{bmatrix}$$

Design matrix

Design matrix is a special name for matrices containing explanatory

variables or features in the context of statistics and machine learning. Some authors favor this name to refer to the set of variables or features in a model.

Matrices as systems of linear equations

I introduced the idea of systems of linear equations as a way to figure out the right combination of linear segments to obtain an outcome. I did this in the context of vectors, now we can extend this to the context of matrices.

Matrices are ideal to represent systems of linear equations. Consider the matrix M and vectors w and y in $\in \mathrm{R}^3$. We can set up a system of linear equations as $\mathrm{M}w = y$ as:

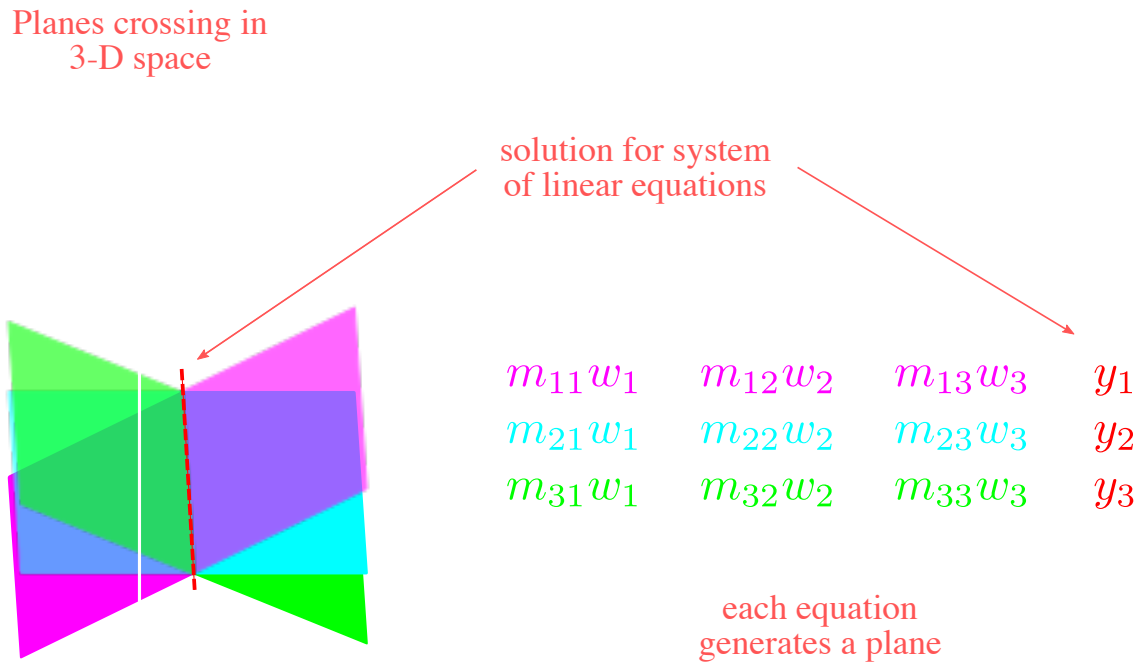
$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

This is equivalent to:

$$\begin{aligned} m_{11}w_1 + m_{12}w_2 + m_{13}w_3 &= y_1 \\ m_{21}w_1 + m_{22}w_2 + m_{23}w_3 &= y_2 \\ m_{31}w_1 + m_{32}w_2 + m_{33}w_3 &= y_3 \end{aligned}$$

Geometrically, the solution for this representation equals to plot a set of planes in 3-dimensional space, one for each equation, and to find the segment where the planes intersect.

Fig. 12: Visualiation system of equations as planes

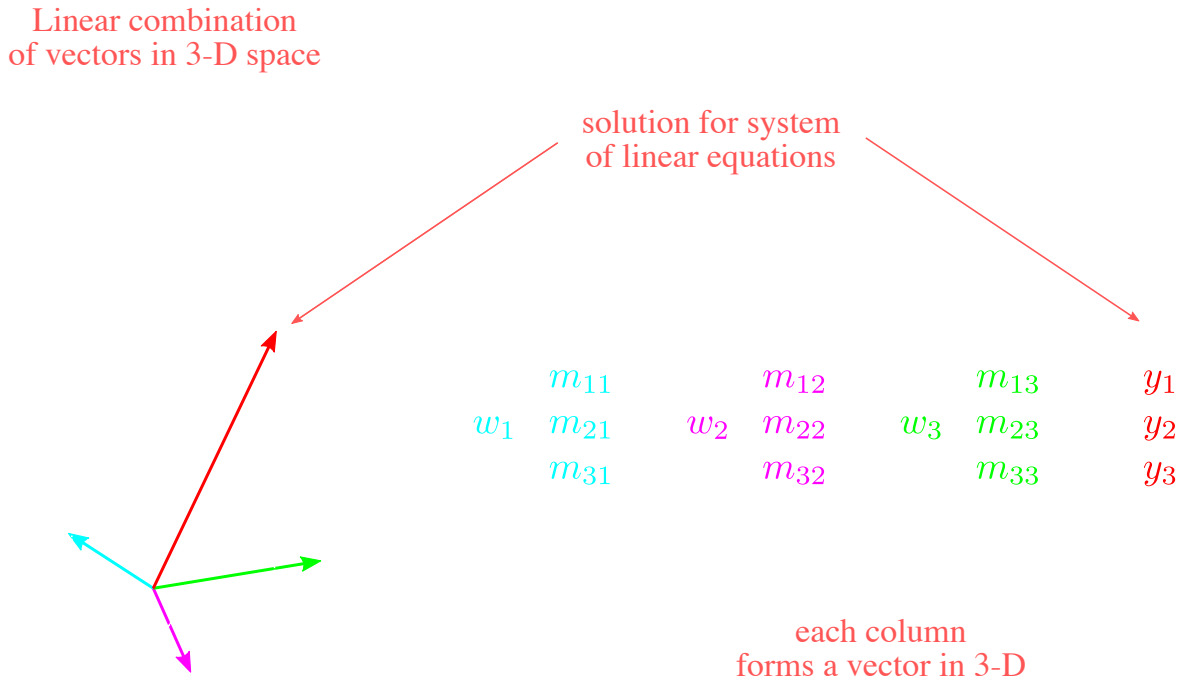


An alternative way, which I personally prefer to use, is to represent the system as a linear combination of the column vectors times a scaling term:

$$w_1 \begin{bmatrix} m_{11} \\ m_{21} \\ m_{31} \end{bmatrix} + w_2 \begin{bmatrix} m_{12} \\ m_{22} \\ m_{32} \end{bmatrix} + w_3 \begin{bmatrix} m_{13} \\ m_{23} \\ m_{33} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

Geometrically, the solution for this representation equals to plot a set of vectors in 3-dimensional space, one for each column vector, then scale them by w_i and add them up, tip to tail, to find the resulting vector y.

Fig. 13: System of equations as linear combination of vectors



The four fundamental matrix subsapces

Let's recall the definition of a subspace in the context of vectors:

1. Contains the zero vector, $\mathbf{0} \in \mathbf{S}$
2. Closure under multiplication, $\forall \alpha \in \mathrm{R} \rightarrow \alpha \times s_i \in S$
3. Closure under addition, $\forall s_i \in S \rightarrow s_1 + s_2 \in S$

These conditions carry on to matrices since matrices are simply collections of vectors. Thus, now we can ask what are all possible subspaces that can be "covered" by a collection of vectors in a matrix. Turns out, there are four fundamental subspaces that can be "covered" by a matrix of valid vectors: (1) the column space, (2) the row space, (3) the null space, and (4) the left null space or null space of the transpose.

These subspaces are considered fundamental because they express many important properties of matrices in linear algebra.

The column space

The column space of a matrix A is composed by **all linear combinations of the columns of $\mathbf{A}$**. We denote the column space as $C(\mathrm{A})$. In other words, $C(\mathrm{A})$ equals to the **span of the columns of $\mathbf{A}$**. This view of a matrix is what we represented in **Fig. 12**: vectors in R^n scaled by real numbers.

For a matrix $\mathrm{A} \in \mathrm{R}^{m \times n}$ and a vector $\mathbf{v} \in \mathrm{R}^{\mathbf{m}}$, the column space is defined as:

$$C(\mathrm{A}) := \{\mathbf{w} \in \mathrm{R}^{\mathbf{n}} \,|\, \mathbf{w} = \mathrm{A}\mathbf{v} \textbf{ for some } \mathbf{v} \in \mathrm{R}^{\mathbf{m}}\}$$

In words: all linear combinations of the column vectors of A and entries of an n dimensional vector $\mathbf{v}$.

The row space

The row space of a matrix A is composed of all linear combinations of the rows of a matrix. We denote the row space as $R(\mathrm{A})$. In other words, $R(\mathrm{A})$ equals to the **span of the rows** of A. Geometrically, this is the way we represented a matrix in **Fig. 11**: each row equation represented as planes. Now, a different way to see the row space, is by transposing A^T. Now, we can define the row space simply as $R(\mathrm{A}^T)$

For a matrix $\mathrm{A} \in \mathrm{R}^{m \times n}$ and a vector $\mathbf{w} \in \mathrm{R}^{\mathbf{m}}$, the row space is defined as:

$$R(\mathrm{A}) := \{\mathbf{v} \in \mathrm{R}^{\mathbf{m}} \,|\, \mathbf{v} = \mathrm{A}\mathbf{w}^{\mathbf{T}} \textbf{ for some } \mathbf{w} \in \mathrm{R}^{\mathbf{n}}\}$$

In words: all linear combinations of the row vectors of A and entries of an m dimensional vector $\mathbf{w}$.

The null space

The null space of a matrix A is composed of all vectors that are map into the zero vector when multiplied by A. We denote the null space as $N(\mathrm{A})$.

For a matrix $\mathrm{A} \in \mathrm{R}^{m \times n}$ and a vector $\mathbf{v} \in \mathrm{R}^{\mathbf{n}}$, the null space is defined as:

$$N(\mathrm{A}) := \{\mathbf{v} \in \mathrm{R}^{\mathbf{m}} \mid \mathrm{A}\mathbf{v} = \mathbf{0}\}$$

The null space of the transpose

The left null space of a matrix A is composed of all vectors that are map into the zero vector when multiplied by A from the left. By "from the left", the vectors on the left of A. We denote the left null space as $N(\mathrm{A}^T)$

For a matrix $\mathrm{A} \in \mathrm{R}^{m \times n}$ and a vector $\mathbf{w} \in \mathrm{R}^{\mathbf{m}}$, the null space is defined as:

$$N(\mathrm{A}^T) := \{\mathbf{w} \in \mathrm{R}^{\mathbf{n}} \mid \mathbf{v}^{\mathbf{T}}\mathrm{A} = \mathbf{0}^{\mathbf{T}}\}$$

Solving systems of linear equations with Matrices

Gaussian Elimination

When I was in high school, I learned to solve systems of two or three equations by the methods of elimination and substitution. Nevertheless, as systems of equations get larger and more complicated, such inspection-based methods become impractical. By inspection-based, I mean "just by looking at the equations and using common sense". Thus, to approach such kind of systems we can use the method of **Gaussian Elimination**.

Gaussian Elimination, is a robust algorithm to solve linear systems. We say is robust, because it works in general, it all possible circumstances. It works by *eliminating* terms from a system of equations, such that it is simplified to the point where we obtain the **row echelon form** of the matrix. A matrix is in row echelon form when all rows contain zeros at the bottom left of the matrix. For instance:

$$\begin{bmatrix} p_1 & a & b \\ 0 & p_2 & c \\ 0 & 0 & p_3 \end{bmatrix}$$

The p values along the diagonal are the **pivots** also known as basic variables of the matrix. An important remark about the pivots, is that they indicate which vectors are linearly independent in the matrix, once the matrix has been reduced to the row echelon form.

There are three *elementary transformations* in Gaussian Elimination that when combined, allow simplifying any system to its row echelon form:

1. Addition and subtraction of two equations (rows)
2. Multiplication of an equation (rows) by a number
3. Switching equations (rows)

Consider the following system $\mathrm{A}\mathbf{w} = \mathbf{y}$:

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 2 & -1 \\ 1 & 3 & 2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix}$$

We want to know what combination of columns of A will generate the target vector $\mathbf{y}$. Alternatively, we can see this as a decomposition problem, as how can we decompose $\mathbf{y}$ into columns of A. To aid the application of Gaussian Elimination, we can generate an **augmented matrix** $(\mathrm{A}|\mathbf{y})$, this is, appending $\mathbf{y}$ to A on this manner:

$$\left[\begin{array}{ccc|c} 1 & 3 & 5 & -1 \\ 2 & 2 & -1 & 1 \\ 1 & 3 & 2 & 2 \end{array}\right]$$

We start by multiplying row 1 by and substracting it from row 2 as $R_2 - 2R_1$ to obtain:

$$\left[\begin{array}{ccc|c} 1 & 3 & 5 & -1 \\ 0 & -4 & -11 & 3 \\ 1 & 3 & 2 & 2 \end{array}\right]$$

If we substract row 1 from row 3 as $R_3 - R_1$ we get:

$$\left[\begin{array}{ccc|c} 1 & 3 & 5 & -1 \\ 0 & -4 & -11 & 3 \\ 0 & 0 & -3 & 3 \end{array}\right]$$

At this point, we have found the row echelon form of A. If we divide row 3 by -3, We know that $w_3 = -1$. By **backsubsitution**, we can solve for w_2 as:

$$-4w_2 + -11(-1) = 3$$
$$-4w_2 = -8$$
$$w_2 = 2$$

Again, taking $w_2 = 2$ and $w_3 = -1$ we can solve for w_1 as:

$$w_1 + 3(2) + 5(-1) = -1w_1 + 6 - 5 = -1w_1 = -2$$

In this manner, we have found that the solution for our system is $w_1 = -2$, $w_2 = 2$, and $w_3 = -1$.

In NumPy, we can solve a system of equations with Gaussian Elimination with the `linalg.solve` method as:

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


np.linalg.solve(A, y)


array([[-2.],
       [ 2.],
       [-1.]])
```

Which confirms our solution is correct.

Gauss-Jordan Elimination

The only difference between **Gaussian Elimination** and **Gauss-Jordan Elimination**, is that this time we "keep going" with the elemental row operations until we obtain the **reduced row echelon form**. The *reduced* part means two additionak things: (1) the pivots must be 1, (2) and the entries above the pivots must be 0. This is simplest form a system of linear equations can take. For instance, for a 3x3 matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Let's retake from where we left Gaussian elimination in the above section. If we divide row 3 by -3 and row 2 by -4 as $\frac{R_3}{-3}$ and $\frac{R_2}{-4}$, we get:

$$\left[\begin{array}{ccc|c} 1 & 3 & 5 & -1 \\ 0 & 1 & 2.75 & -0.75 \\ 0 & 0 & 1 & -1 \end{array}\right]$$

Again, by this point we we know $w_3 = -1$. If we multiply row 2 by 3 and substract from row 1 as $R_1 - 3R_2$:

$$\left[\begin{array}{ccc|c} 1 & 0 & -3.25 & 1.25 \\ 0 & 1 & 2.75 & -0.75 \\ 0 & 0 & 1 & -1 \end{array}\right]$$

Finally, we can add 3.25 times row 3 to row 1, and substract 2.75 times row 3 to row 2, as $R_1 + 3.25R_3$ and $R_2 - 2.75R_3$ to get the **reduced row echelon form** as:

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & -1 \end{array}\right]$$

Now, by simply following the rules of matrix-vector multiplication, we get =

$$w_1\begin{bmatrix}1\\0\\0\end{bmatrix}+w_2\begin{bmatrix}0\\1\\0\end{bmatrix}+w_3\begin{bmatrix}0\\0\\1\end{bmatrix}=\begin{bmatrix}w_1\\w_2\\w_3\end{bmatrix}=\begin{bmatrix}-2\\2\\-1\end{bmatrix}$$

There you go, we obtained that $w_1 = -2$, $w_2 = 2$, and $w_3 = -1$.

Matrix basis and rank

A set of n linearly independent column vectors with n elements forms a **basis**. For instance, the column vectors of A are a basis:

$$\mathrm{A}=\begin{bmatrix}1&0\\0&1\end{bmatrix}$$

"A basis for what?" You may be wondering. In the case of A, for any vector $\mathbf{y} \in \mathrm{R}^2$. On the contrary, the column vectors for B *do not* form a basis for R^2:

$$\mathrm{B}=\begin{bmatrix}1&0&1\\0&1&1\end{bmatrix}$$

In the case of B, the third column vector is a linear combination of first and second column vectors.

The definition of a *basis* depends on the **independence-dimension inequality**, which states that *a linearly independent set of n vectors can have at most n elements*. Alternatively, we say that any set of n vectors with $n+1$ elements is, *necessarily*, linearly dependent. Given that each vector in a *basis* is linearly independent, we say that any vector $\mathbf{y}$ with n elements, can be generated in a unique linear combination of the *basis* vectors. Hence, any matrix more columns than rows (as in B) will have dependent vectors. *Basis* are sometimes referred to as the *minimal generating set*.

An important question is how to find the *basis* for a matrix. Another way to put the same question is to found out which vectors are linearly independent of each other. Hence, we need to solve:

$$\sum_{i=1}^{k} \beta_i a_i = 0$$

Where a_i are the column vectors of A. We can approach this by using **Gaussian Elimination** or **Gauss-Jordan Elimination** and reducing A to its **row echelon form** or **reduced row echelon form**. In either case, recall that the *pivots* of the echelon form indicate the set of linearly independent vectors in a matrix.

NumPy does not have a method to obtain the row echelon form of a matrix. But, we can use Sympy, a Python library for symbolic mathematics that counts with a module for Matrices operations.SymPy has a method to obtain the reduced row echelon form and the pivots, `rref`.

```
from sympy import Matrix


A = Matrix([[1, 0, 1],
            [0, 1, 1]])


B = Matrix([[1, 2, 3, -1],
            [2, -1, -4, 8],
            [-1, 1, 3, -5],
            [-1, 2, 5, -6],
            [-1, -2, -3, 1]])


A_rref, A_pivots = A.rref()


print('Reduced row echelon form of A:')
```

```
Reduced row echelon form of A:
```

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

```
print(f'Column pivots of A: {A_pivots}')
```

```
Column pivots of A: (0, 1)
```

```
B_rref, B_pivots = B.rref()
```

```
print('Reduced row echelon form of B:')
```

```
Reduced row echelon form of B:
```

$$\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

```
print(f'Column pivots of A: {B_pivots}')
```

```
Column pivots of A: (0, 1, 3)
```

For A, we found that the first and second column vectors are the *basis*, whereas for B is the first, second, and fourth.

Now that we know about a *basis* and how to find it, understanding the concept of *rank* is simpler. The **rank** of a matrix A is the dimensionality of the vector space generated by its number of linearly independent column vectors. This happens to be identical to the dimensionality of the vector space generated by its row vectors. We denote the *rank* of matrix as $rk(\mathrm{A})$ or $rank(\mathrm{A})$.

For an square matrix $\mathrm{R}^{m \times n}$ (i.e., $m = n$), we say is **full rank** when every column and/or row is linearly independent. For a non-square matrix with $m > n$ (i.e., more rows than columns), we say is **full rank** when every row is linearly independent. When $m < n$ (i.e., more columns than rows), we say is **full rank** when every column is

linearly independent.

From an applied machine learning perspective, the *rank* of a matrix is relevant as a measure of the information content of the matrix. Take matrix B from the example above. Although the original matrix has 5 columns, we know is rank 4, hence, it has less information than it appears at first glance.

Matrix norm

As with vectors, we can measure the size of a matrix by computing its **norm**. There are multiple ways to define the norm for a matrix, as long it satisfies the same properties defined for vectors norms: (1) absolutely homogeneous, (2) triangle inequality, (3) positive definite (see vector norms section). For our purposes, I'll cover two of the most commonly used norms in machine learning: (1) **Frobenius norm**, (2) **max norm**, (3) **spectral norm**.

Note: I won't cover the spectral norm just yet, because it depends on concepts that I have not introduced at this point.

Frobenius norm

The **Frobenius norm** is an element-wise norm named after the German mathematician Ferdinand Georg Frobenius. We denote this norm as $\|A\|_F$. You can thing about this norm as flattening out the matrix into a long vector. For instance, a 3×3 matrix would become a vector with $n = 9$ entries. We define the Frobenius norm as:

$$\|A\|_F := \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}$$

In words: square each entry of A, add them together, and then take the square root.

In NumPy, we can compute the Frobenius norm as with the `linal.norm` method ant `fro` as the argument:

```
A = np.array([[1, 2, 3],
              [4, 5, 6],
```

```
                [7, 8, 9]])
```

```
np.linalg.norm(A, 'fro')
```

```
16.881943016134134
```

Max norm

The **max norm** or **infinity norm** of a matrix equals to the largest sum of the absolute value of row vectors. We denote the max norm as $\|A\|_max$. Consider $A \in R^{m \times n}$. We define the max norm for A as:

$$\|A\|_{max} := \max_i \sum_{j=1}^{n} |a_{ij}|$$

This equals to go row by row, adding the absolute value of each entry, and then selecting the largest sum.

In Numpy, we compute the max norm as:

```
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

```
np.linalg.norm(A, np.inf)
```

```
24.0
```

In this case, is easy to see that the third row has the largest absolute value.

Spectral norm

To understand this norm, is necessary to first learn about eigenvectors and eigenvalues, which I cover later.

The **spectral norm** of a matrix equals to the largest singular value σ_1. We denote the spectral norm as $\|A\|_2$. Consider $A \in R^{m \times n}$. We define

the spectral for A as:

$$\|A\|_2 := \max_x \frac{\|Ax\|_2}{\|x\|_2}$$

In Numpy, we compute the max norm as:

```
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

```
np.linalg.norm(A, 2)
```

```
16.84810335261421
```

Linear and affine mappings

```
# Libraries for this section
import numpy as np
import pandas as pd
import altair as alt
alt.themes.enable('dark')
```

```
ThemeRegistry.enable('dark')
```

Linear mappings

Now we have covered the basics of vectors and matrices, we are ready to introduce the idea of a linear mapping. **Linear mappings**, also known as *linear transformations* and *linear functions*, indicate the correspondence between vectors in a vector space V and the same vectors in a different vector space W. This is an abstract idea. I like to think about this in the following manner: imagine there is a multiverse as in Marvel comics, but instead of humans, aliens, gods, stars, galaxies, and superheroes, we have *vectors*. In this context, a linear mapping would indicate the *correspondence* of

entities (i.e., planets, humans, superheroes, etc) *between universes*. Just imagine us, placidly existing in our own universe, and suddenly a *linear mapping* happens: our entire universe would be transformed into a different one, according to whatever rules the linear mapping has enforced. Now, switch *universes* for *vector spaces* and *us* by vectors, and you'll get the full picture.

So, linear mappings transform vector spaces into others. Yet, such transformations are constrained to a spefic kind: **linear ones**. Consider a linear mapping T and a pair of vectors $\mathbf{x}$ and $\mathbf{y}$. To be valid, a linear mapping must satisfies these rules:

$$T(\mathbf{x} + \mathbf{y}) = T(\mathbf{x}) + T(\mathbf{y})$$
$$T(\alpha\mathbf{x}) = \alpha T(\mathbf{x}), \forall\alpha$$

In words:

>> The transformation of the sum of the vectors must be equal to taking the transformation of each vector individually and then adding them up.
>> The transformation of a scaled version of a vector must be equal to taking the transformation of the vector first and then scaling the result.

The two properties above can be condenced into one, the **superposition property**:

$$T(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha T(\mathbf{x}) + \beta T(\mathbf{y})$$

As a result of satisfying those properties, linear mappings **preserve the structure of the original vector space**. Imagine a vector space $\in \mathbb{R}^2$, like a grid on lines in a cartesian plane. Visually, preserving the structure of the vector space after a mapping means to: (1) the origin of the coordinate space remains fixed, and (2) the lines remain lines and parallel to each other.

In linear algebra, linear mappings are represented as matrices and performed by matrix multiplication. Take a vector $\mathbf{x}$ and a matrix A. We say that when A multiplies $\mathbf{x}$, the matrix transform the vector into another one:

$$T(\mathbf{x}) = A\mathbf{x}$$

The typicall notation for a linear mapping is the same we used for

functions. For the vector spaces V and W, we indicate the linear mapping as $T: V \rightarrow W$

Examples of linear mappings

Let's examine a couple of examples of proper linear mappings. In general, *dot products are linear mappings*. This should come as no surprise since dot products are linear operations by definition. Dot products sometimes take special names, when they have a well-known effect on a linear space. I'll examine two simple cases: **negation** and **reversal**. Keep in mind that although we will test this for one vector, this mapping work on the entire vector space (i.e., the span) of a given dimensionality.

Negation matrix

A **negation matrix** returns the opposite sign of each element of a vector. It can be defined as:

$$T := A := -I$$

This is, the negative identity matrix. Consider a pair of vectors $\mathbf{x} \in R^3$ and $\mathbf{x} \in y^3$, and the negation matrix $-I \in R^{3 \times 3}$. Let's test the linear mapping properties with `NumPy`:

```
x = np.array([[-1],
              [0],
              [1]])

y = np.array([[-3],
              [0],
              [2]])

T = np.array([[-1,0,0],
              [0,-1,0],
              [0,0,-1]])
```

We first test $T(\mathbf{x} + \mathbf{y}) = T(\mathbf{x}) + T(\mathbf{y})$:

```
left_side_1 = T @ (x+y)
```

```
right_side_1 = (T @ x) + (T @ y)
print(f"Left side of the equation:\n{left_side_1}")
print(f"Right side of the equation:\n{right_side_1}")
```

```
Left side of the equation:
[[ 4]
 [ 0]
 [-3]]
Right side of the equation:
[[ 4]
 [ 0]
 [-3]]
```

Hence, we confirm we get the same results.

Let's check the second property $T(\alpha\mathbf{x}) = \alpha T(\mathbf{x}), \forall\alpha$

```
alpha = 2
left_side_2 = T @ (alpha * x)
right_side_2 = alpha * (T @ x)
print(f"Left side of the equation:\n{left_side_2}")
print(f"Right side of the equation:\n{right_side_2}")
```

```
Left side of the equation:
[[ 2]
 [ 0]
 [-2]]
Right side of the equation:
[[ 2]
 [ 0]
 [-2]]
```

Again, we confirm we get the same results for both sides of the equation

Reversal matrix

A **reversal matrix** returns reverses the order of the elements of a vector. This is, the last become the first, the second to last

becomes the second, and so on. For a matrix in $R^{3\times3}$ is defined as:

$$T := \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

In general, it is the *identity matrix but backwards*, with ones from the bottom left corner to the top right corern. Consider a pair of vectors $\mathbf{x} \in R^3$ and $\mathbf{x} \in y^3$, and the reversal matrix $T \in R^{3\times3}$. Let's test the linear mapping properties with NumPy:

```
x = np.array([[-1],
              [0],
              [1]])

y = np.array([[-3],
              [0],
              [2]])

T = np.array([[0,0,1],
              [0,1,0],
              [1,0,0]])
```

We first test $T(\mathbf{x} + \mathbf{y}) = T(\mathbf{x}) + T(\mathbf{y})$:

```
x_reversal = T @ x
y_reversal = T @ y
left_side_1 = T @ (x+y)
right_side_1 = (T @ x) + (T @ y)
print(f"x before reversal:\n{x}\nx after reversal \n{x_reversal}")
print(f"y before reversal:\n{y}\ny after reversal \n{y_reversal}")
print(f"Left side of the equation (add reversed vectors):\n{left_s
print(f"Right side of the equation (add reversed vectors):\n{right
```

```
x before reversal:
[[-1]
 [ 0]
 [ 1]]
```

```
x after reversal
[[ 1]
 [ 0]
 [-1]]
y before reversal:
[[-3]
 [ 0]
 [ 2]]
y after reversal
[[ 2]
 [ 0]
 [-3]]
Left side of the equation (add reversed vectors):
[[ 3]
 [ 0]
 [-4]]
Right side of the equation (add reversed vectors):
[[ 3]
 [ 0]
 [-4]]
```

This works fine. Let's check the second property $T(\alpha \mathbf{x}) = \alpha T(\mathbf{x}), \forall \alpha$

```
alpha = 2
left_side_2 = T @ (alpha * x)
right_side_2 = alpha * (T @ x)
print(f"Left side of the equation:\n{left_side_2}")
print(f"Right side of the equation:\n{right_side_2}")
```

```
Left side of the equation:
[[ 2]
 [ 0]
 [-2]]
Right side of the equation:
[[ 2]
 [ 0]
 [-2]]
```

Examples of nonlinear mappings

As with most subjects, examining examples of *what things are not* can be enlightening. Let's take a couple of non-linear mappings: **norms** and **translation**.

Norms

This may come as a surprise, but norms are not linear transformations. Not "some" norms, but all norms. This is because of the very definition of a norm, in particular, the **triangle inequality** and **positive definite** properties, colliding with the requirements of linear mappings.

First, the triangle inequality defines: $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$. Whereas the first requirement for linear mappings demands: $\mathrm{T}(\mathbf{x} + \mathbf{y}) = \mathrm{T}(\mathbf{x}) + \mathrm{T}(\mathbf{y})$. The problem here is in the $\leq$ condition, which means adding two vectors and then taking the norm *can* be less than the sum of the norms of the individual vectors. Such condition is, by defnition, not allowed for linear mappings.

Second, the positive definite defines: $\|\mathbf{x}\| \geq \mathbf{0}$ and $\|\mathbf{x}\| = \mathbf{0} \iff \mathbf{x} = \mathbf{0}$. Put simply, norms *have to* be a postive value. For instance, the norm of $\|-x\| = \|x\|$, instead of $\|-x\|$. But, the second property for linear mappings requires $\|-\alpha\mathbf{x}\| = -\alpha\|\mathbf{x}\|$. Hence, it fails when we multiply by a negative number (i.e., it can preserve the negative sign).

Translation

Translation is a geometric transformation that moves every vector in a vector space by the same distance in a given direction. Translation is an operation that matches our everyday life intuitions: move a cup of coffee from your left to your right, and you would have performed translation in R^3 space.

Contrary to what we have seen so far, the translation matrix is represented with **homogeneous coordinates** instead of cartesian coordinates. Put simply, the homogeneous coordinate system adds a extra 1 at the end of vectros. For instance, the vector in R^2 cartesian coordinates:

$$\mathbf{x} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

Becomes the following in R^2 homogeneous coordinates:

$$\mathbf{x} = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$$

In fact, the translation matrix for the general case can't be represented with cartesian coordinates. Homogeneous coordinates are the standard in fields like computer graphics since they allow us to better represent a series of transformations (or mappings) like scaling, translation, rotation, etc.

A translation matrix in R^3 can be denoted as:

$$\mathrm{T}_v = \begin{bmatrix} 1 & 0 & v_1 \\ 0 & 1 & v_2 \\ 0 & 0 & 1 \end{bmatrix}$$

Where v_1 and v_2 are the values added to each dimension for translation. For instance, consider $\mathbf{x} = \begin{bmatrix} 2 & 2 \end{bmatrix}^{\mathbf{T}} \in \mathrm{R}^{\mathbf{2}}$. If we want translate this 3 units in the first dimension, and 1 units in the second dimension, we first transfor the vector to homogeneous coordinates $\mathbf{x} = \begin{bmatrix} 2 & 2 & 1 \end{bmatrix}^{\mathbf{T}}$, and then perfom matrix-vector multiplication as usual:

$$\mathrm{T}_v = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ 1 \end{bmatrix}$$

The first two vectors in the translation matrix simple reproduce the original vector (i.e., the identity), and the third vector is the one actually "moving" the vectors.

Translation is **not a linear mapping** simply because $\mathrm{T}(\mathbf{x} + \mathbf{y}) = \mathrm{T}(\mathbf{x}) + \mathrm{T}(\mathbf{y})$ **does not hold**. In the case of translation $\mathrm{T}(\mathbf{x} + \mathbf{y}) = \mathrm{T}(\mathbf{x} + \mathbf{v_1}) + \mathrm{T}(\mathbf{y} + \ + \mathbf{v_1})$,

which invalidates the operation as a linear mapping. This type of mapping is known as an **affine mapping or transformation**, which is the topic I'll review next.

Affine mappings

The simplest way to describe affine mappings (or transformations) is as a *linear mapping* + *translation*. Hence, an affine mapping M takes the form of:

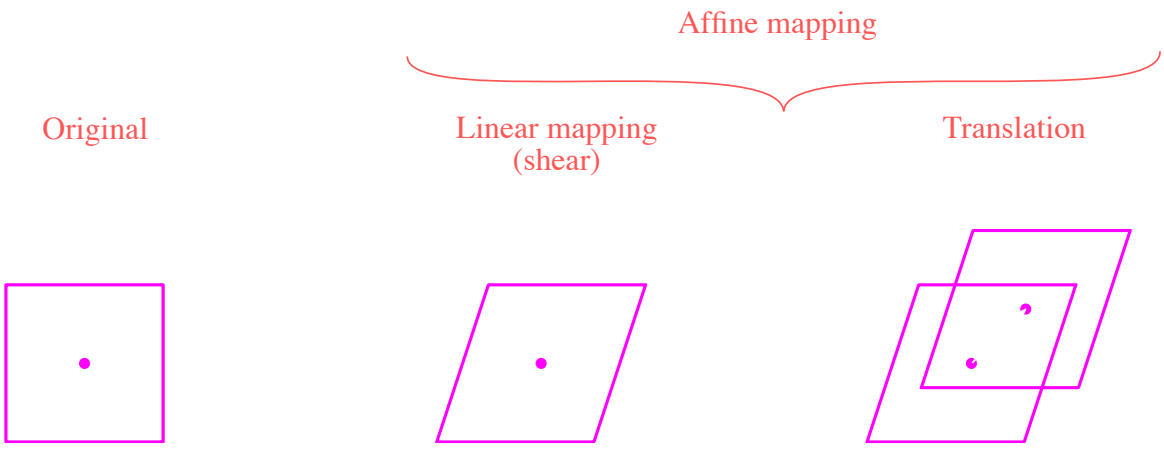
$$\mathrm{M}(\mathbf{x}) = \mathrm{A}\mathbf{x} + \mathbf{b}$$

Where A is a linear mapping or transformation and $\mathbf{b}$ is the translation vector.

If you are familiar with linear regression, you would notice that the above expression is its matrix form. Linear regression is usually analyzed as a linear mapping plus noise, but it can also be seen as an affine mapping. Alternative, we can say that $\mathrm{A}\mathbf{x} + \mathbf{b}$ is a linear mapping *if and only if* $\mathbf{b} = 0$.

From a geometrical perspective, affine mappings displace spaces (lines or hyperplanes) from the origin of the coordinate space. Consequently, affine mappings do not operate over *vector spaces* as the zero vector condition $\mathbf{0} \in \mathbf{S}$ does not hold anymore. Affine mappings act onto *affine subspaces*, that I'll define later in this section.

Fig. 14: Affine mapping



Affine combination of vectors

We can think in affine combinations of vectors, as linear combinations with an added constraint. Let's recall de definitoon for a linear combination. Consider a set of vectors $x_1, \ldots, x_k$ and scalars $\beta_1, \ldots, \beta_k \in \mathrm{R}$, then a linear combination is:

$$\sum_{j=1}^{k} \beta_j x_j := \beta_1 x_1 + \ldots + \beta_k x_k$$

For affine combinations, we add the condition:

$$\sum_{j=1}^{k} \beta_j = 1$$

In words, we constrain the sum of the weights β to 1. In practice, this defines a *weighted average of the vectors*. This restriction has a palpable effect which is easier to grasp from a geometric perspective.

Fig. 15: Affine combinations

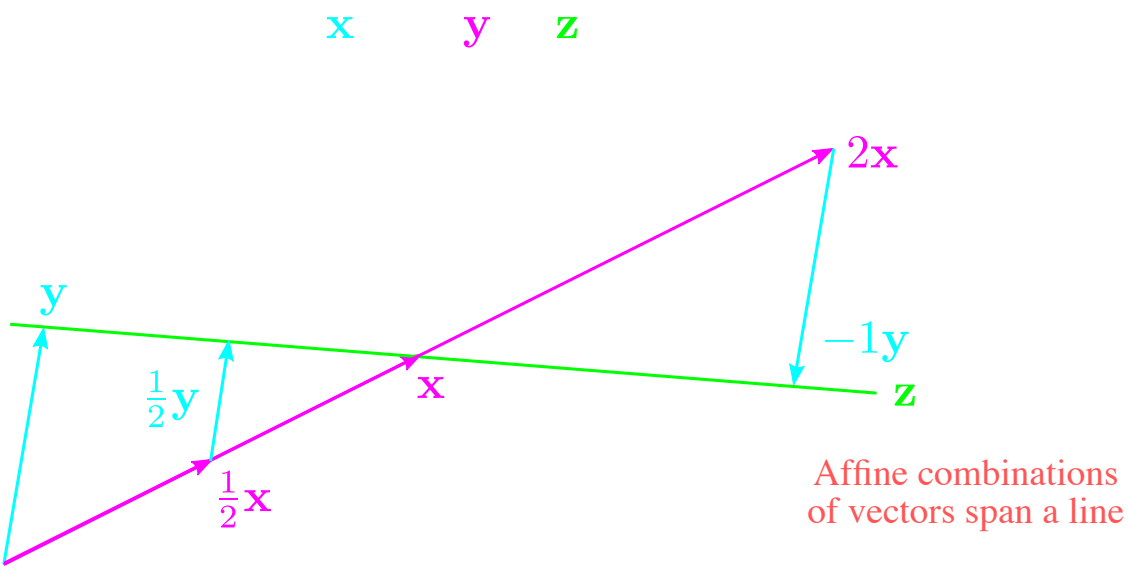


Fig. 15 shows two affine combinations. The first combination with weights $\beta_1 = \frac{1}{2}$ and $\beta_2 = \frac{1}{2}$, which yields the midpoint between vectors $\mathbf{x}$ and $\mathbf{y}$. The second combination with weights $\beta_1 = 3$ and $\beta_2 = -1$ (add up to 1), which yield a point over the vector $\mathbf{z}$. In both cases, we have that the resulting vector lies on the same line. This is a general consequence of constraining the sum of the weights to 1: *every affine combination of the same set of vectors will map onto the same space*.

Affine span

The set of all linear combinations, define the the vector span. Similarly, the set of all affine combinations determine the **affine span**. As we saw in Fig. 15, every affine of vectors $\mathbf{x}$ and $\mathbf{y}$ maps onto the line $\mathbf{z}$. More generally, we say that the **affine span** of vectors $\mathbf{x}_1, \cdots, \mathbf{x}_k$ is:

$$\mathbf{x}_1, \cdots, \mathbf{x}_k := \sum_{j=1}^{k} \beta_j \mathbf{x}_j, \mid \sum_{j=1} \beta_j = 1 \in \mathrm{R} \forall \beta$$

Again, in words: the affine span is the set of all linear combinations of the vector set, such that the weights add up to 1 and all weights are real numbers. Hence, the fundamental difference between vector spaces and affine spaces, is the former will span

the entire R^n space (assuming independent vectors), whereas the latter will span a line.

Let's consider three cases in R^3: (1) three linearly independent vectors; (2) two linearly independent vectors and one dependent vector; (3) three linearly dependent vectors. In case (1), the affine span is the 2-dimensional plane containing those vectors. In case (2), the affine space is a line. Finally, in case (3), the span a single point. This may not be entirely obvious, so I encourage you to draw and the three cases, take the affine combinations and see what happens.

Affine space and subspace

In simple terms, **affine spaces** are *translates* of vector spaces, this is, vector spaces that have been offset from the origin of the coordinate system. Such a notion makes sound affine spaces as a special case of vector spaces, but they are actually more general. Indeed, affine spaces provide a more general framework to do geometric manipulation, as they work independently of the choice of the coordinate system (i.e., it is not constrained to the origin). For instance, the set of solutions of the system of linear equations $\mathrm{A}\mathbf{x} = \mathbf{y}$ (i.e., linear regression), is an affine space, not a linear vector space.

Consider a vector space V, a vector $\mathbf{x}_0 \in \mathrm{V}$, and a subset $\mathrm{U} \subseteq \mathrm{V}$. We define an affine subspace L as:

$$\mathrm{L} = \mathbf{x}_0 + \mathrm{U} := \{\mathbf{x}_0 + \mathbf{u} : \mathbf{u} \in \mathrm{U}\}$$

Further, any point, line, plane, or hyperplane in R^n that does not go through the origin, is an affine subspace.

Affine mappings using the augmented matrix

Consider the matrix $\mathrm{A} \in \mathrm{R}^{m \times n}$, and vectors $\mathbf{x}, \mathbf{b}, \mathbf{y} \in \mathrm{R}^n$

We can represent the system of linear equations:

$$\mathrm{A}\mathbf{x} + \mathbf{b} = \mathbf{y}$$

As a single matrix vector multiplication, by using an **augmented**

matrix of the form:

$$\left[\begin{array}{ccc|c} & & & x_1 \\ & \mathrm{A} & & \vdots \\ & & & x_n \\ \hline 0 & \cdots & 1 & 1 \end{array}\right] = \begin{bmatrix} y_1 \\ \vdots \\ y_n \\ 1 \end{bmatrix}$$

This form is known as the **affine transformation matrix**. We made use of this form when we exemplified *translation*, which happens to be an affine mapping.

Special linear mappings

There are several important linear mappings (or transformations) that can be expressed as matrix-vector multiplications of the form $\mathbf{y} = \mathrm{A}\mathbf{x}$. Such mappings are common in image processing, computer vision, and other linear applications. Further, combinations of linear and nonlinear mappings are what complex models as neural networks do to learn mappings from inputs to outputs. Here we briefly review six of the most important linear mappings.

Scaling

Scaling is a mapping of the form $\mathbf{y} = \mathrm{A}\mathbf{x}$, with $\mathrm{A} = \alpha\mathrm{I}$. Scaling *stretches* $\mathbf{x}$ by a factor $|\alpha|$ when $\alpha < 1$, *shrinks* $\mathbf{x}$ when $\alpha < 1$, and *reverses* the direction of the vector when $\alpha < 0$. For geometrical objects in Euclidean space, scaling changes the size but not the shape of objects. An scaling matrix in R^2 takes the form:

$$\begin{bmatrix} s_1 & 0 \\ 0 & s_2 \end{bmatrix}$$

Where s_1, s_2 are the scaling factors.

Let's scale a vector using NumPy. We will define a scaling matrix A, a vector $\mathbf{x}$ to scale, and then plot the original and scaled vectors with Altair.

```
A = np.array([[2.0, 0],
              [0, 2.0]])

x = np.array([[0, 2.0,],
              [0, 4.0,]])
```

To scale $\mathbf{x}$, we perform matrix-vector multiplication as usual

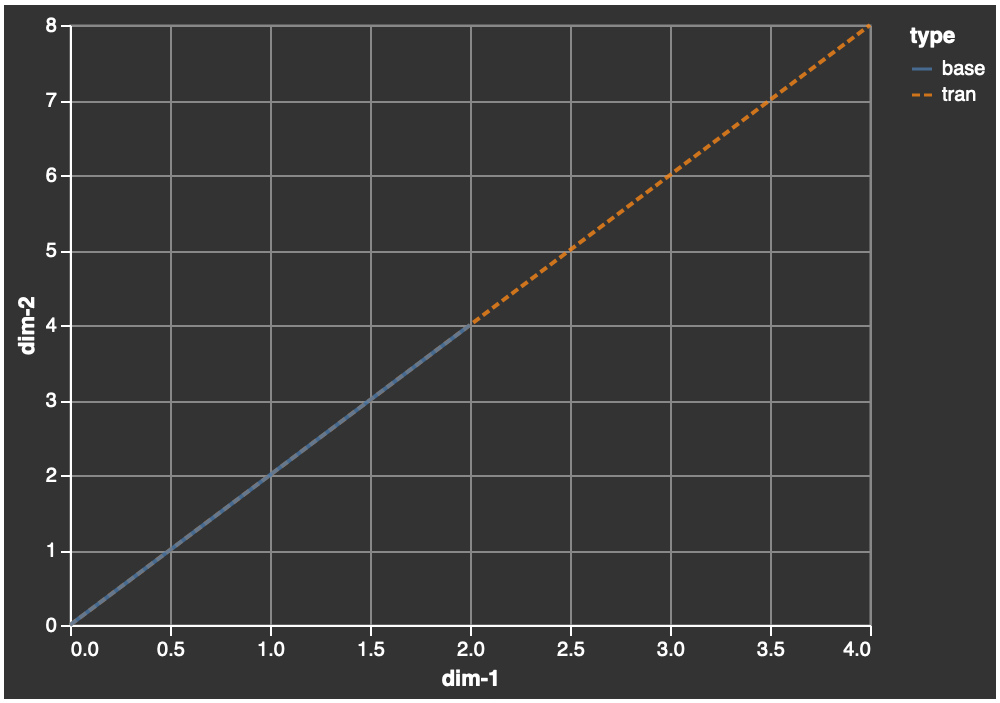
```
y = A @ x


z = np.column_stack((y,x))


df = pd.DataFrame({'dim-1': z[0], 'dim-2':z[1], 'type': ['tran', '
```

```
df
```

	dim-1	dim-2	type
0	0.0	0.0	tran
1	4.0	8.0	tran
2	0.0	0.0	base
3	2.0	4.0	base

We see that the resulting scaled vector ('tran') is indeed two times the original vector ('base'). Now let's plot. The light blue line solid line represents the original vector, whereas the dashed orange line represents the scaled vector.

```
chart = alt.Chart(df).mark_line(opacity=0.8).encode(
    x='dim-1',
    y='dim-2',
    color='type',
    strokeDash='type')

chart
```



Reflection

Reflection is the mirror image of an object in Euclidean space. For the general case, reflection of a vector $\mathbf{x}$ through a line that passes through the origin is obtained as:

$$\begin{bmatrix} \cos(2\theta) & \sin(2\theta) \\ \sin(2\theta) & -\cos(2\theta) \end{bmatrix} \mathbf{x}$$

where θ are radians of inclination with respect to the horizontal axis. I've been purposely avoiding trigonometric functions, so let's examine a couple of special cases for a vector $\mathbf{x}$ in R^2 (that can be extended to an arbitrary number of dimensions).

Reflection along the horizontal axis, or around the line at $0°$ from the origin:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Reflection along the vertical axis, or around the line at $90°$ from the origin:

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

Reflection along the line where the horizontal axis equals the vertical axis, or around the line at $45°$ from the origin:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Reflection along the line where the horizontal axis equals the negative of the vertical axis, or around the line at $-45°$ from the origin:

$$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$$

Let's reflect a vector using NumPy. We will define a reflection matrix A, a vector $\mathbf{x}$ to reflect, and then plot the original and reflected vectors with Altair.

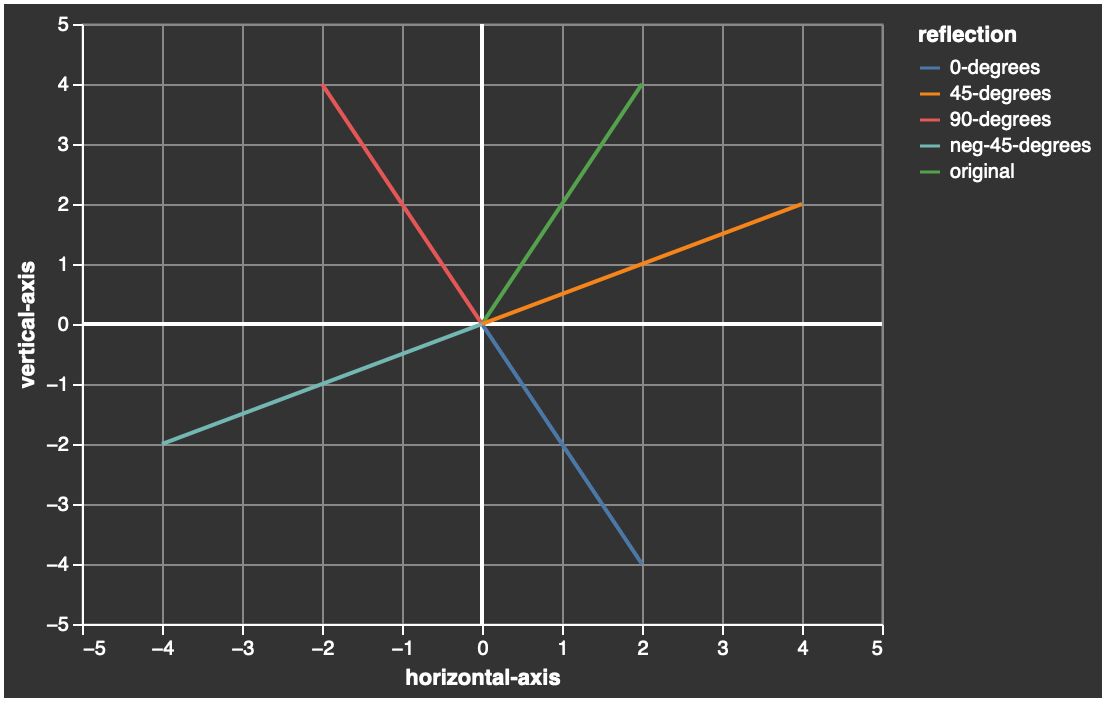
```
# rotation along the horiontal axis
A1 = np.array([[1.0, 0],
               [0, -1.0]])

# rotation along the vertical axis
A2 = np.array([[-1.0, 0],
               [0, 1.0]])

# rotation along the line at 45 degrees from the origin
A3 = np.array([[0, 1.0],
               [1.0, 0]])

# rotation along the line at -45 degrees from the origin
A4 = np.array([[0, -1.0],
               [-1.0, 0]])

x = np.array([[0, 2.0,],
              [0, 4.0,]])


y1 = A1 @ x
y2 = A2 @ x
y3 = A3 @ x
```

```
y4 = A4 @ x


z = np.column_stack((x, y1, y2, y3, y4))


df = pd.DataFrame({'dim-1': z[0], 'dim-2':z[1],
                   'reflection': ['original', 'original',
                                  '0-degrees', '0-degrees',
                                  '90-degrees', '90-degrees',
                                  '45-degrees', '45-degrees',
                                  'neg-45-degrees', 'neg-45-degree


df
```

	dim-1	dim-2	reflection
0	0.0	0.0	original
1	2.0	4.0	original
2	0.0	0.0	0-degrees
3	2.0	-4.0	0-degrees
4	0.0	0.0	90-degrees
5	-2.0	4.0	90-degrees
6	0.0	0.0	45-degrees
7	4.0	2.0	45-degrees
8	0.0	0.0	neg-45-degrees
9	-4.0	-2.0	neg-45-degrees

```
def base_coor(ran1: float, ran2: float):
    '''return base chart with coordinate space'''
    df_base = pd.DataFrame({'horizontal': np.linspace(ran1, ran2,

    h = alt.Chart(df_base).mark_line(color='white').encode(
        x='horizontal',
```

```
            y='vertical')
        v = alt.Chart(df_base).mark_line(color='white').encode(
            y='horizontal',
            x='vertical')
        base = h + v

        return base


chart = alt.Chart(df).mark_line().encode(
    x=alt.X('dim-1', axis=alt.Axis(title='horizontal-axis')),
    y=alt.Y('dim-2', axis=alt.Axis(title='vertical-axis')),
    color='reflection')

base_coor(-5.0, 5.0) + chart
```



Shear

Shear mappings are hard to describe in words but easy to understand with images. I recommend to look at the shear mapping below and then read this description: a shear mapping displaces points of an object in a given direction (e.g., all points to the right), in a proportion equal to their perpendicular distance from an axis (e.g., the line on the x axis) that remains fixed. A "proportion equal to their perpendicular distance" means that points further

away from the reference axis displace more than points near to the axis.

For an object in R^2, a **horizontal shear** matrix (i.e., paraller to the horizontal axis) takes the form:

$$\begin{bmatrix} 1 & m \\ 0 & 1 \end{bmatrix}$$

Where m is the *shear factor*, that essentially determines how pronounced is the shear.

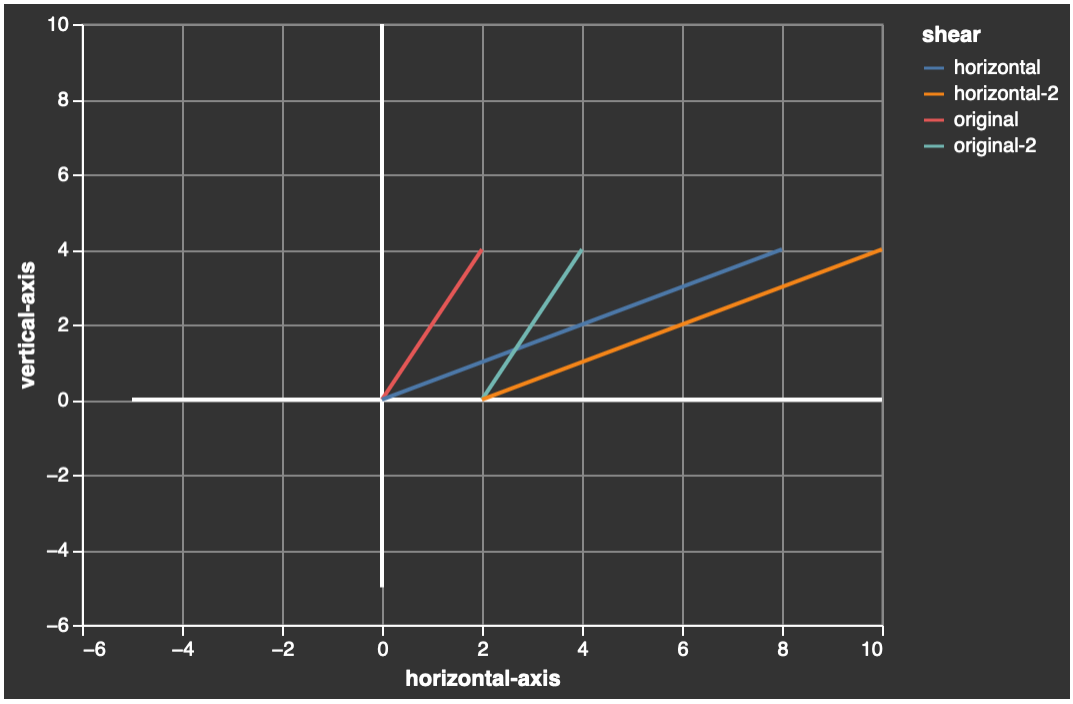
For an object in R^2, a **vertical shear** matrix (i.e., paraller to the vertical axis) takes the form:

$$\begin{bmatrix} 1 & 0 \\ m & 1 \end{bmatrix}$$

Let's shear a vector using NumPy. We will define a shear matrix A, a pair of vectors $\mathbf{x}$ and $\mathbf{u}$ to shear, and then plot the original and shear vectors with Altair. The reason we define two vectors, is that shear mappings are easier to appreciate with planes or multiple sides figures than single lines.

```
# shear along the horiontal axis
A1 = np.array([[1.0, 1.5],
               [0, 1.0]])

x = np.array([[0, 2.0,],
              [0, 4.0,]])

u = np.array([[2, 4.0,],
              [0, 4.0,]])


y1 = A1 @ x
v1 = A1 @ u

z = np.column_stack((x, y1, u, v1))
```

```
df = pd.DataFrame({'dim-1': z[0], 'dim-2':z[1],
                   'shear': ['original', 'original',
                             'horizontal', 'horizontal',
                             'original-2', 'original-2',
                             'horizontal-2', 'horizontal-2'
                            ]})


chart = alt.Chart(df).mark_line().encode(
    x=alt.X('dim-1', axis=alt.Axis(title='horizontal-axis')),
    y=alt.Y('dim-2', axis=alt.Axis(title='vertical-axis')),
    color='shear')

base_coor(-5.0, 10.0) + chart
```



Rotation

Rotation mappings do exactly what their name indicates: they move objects (by convection) counterclockwise in Euclidean space. For the general case in R^2, counterclockwise of vector $\mathbf{x}$ by θ radiants rotations is obtained as:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}\mathbf{x}$$

Again, let's examine a couple of special cases.

A $90°$ rotation matrix in $\mathbb{R}^2$:

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}\mathbf{x}$$

A $180°$ rotation matrix in $\mathbb{R}^2$:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}\mathbf{x}$$

A $270°$ rotation matrix in $\mathbb{R}^2$:

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}\mathbf{x}$$

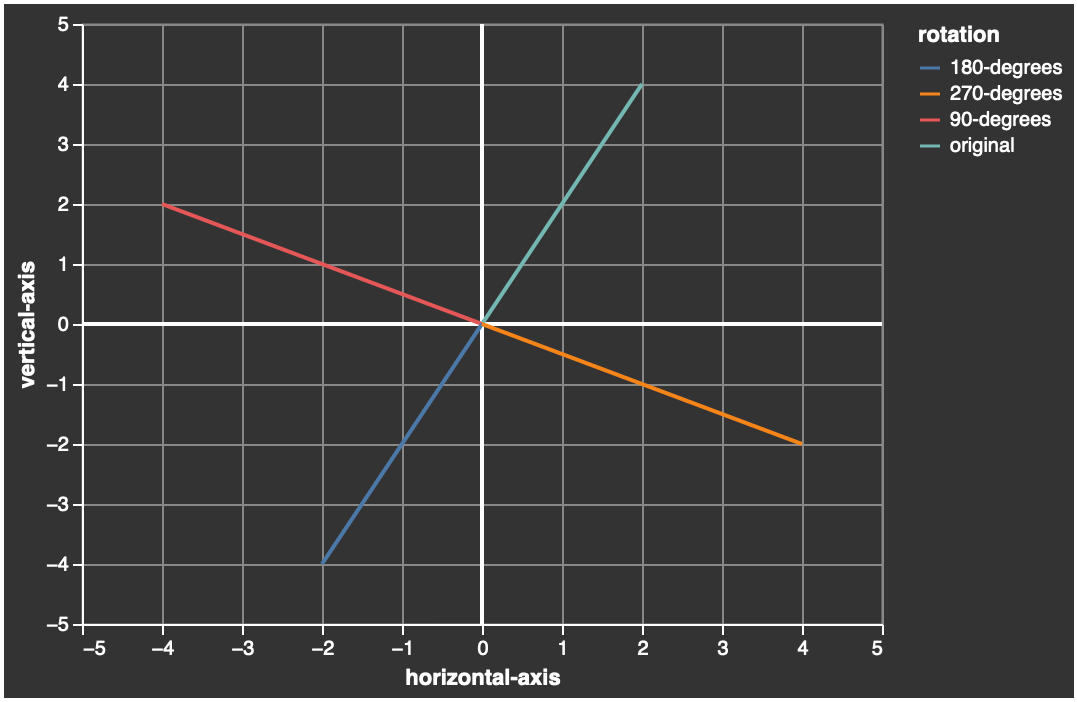
Let's rotate a vector using NumPy. We will define a rotation matrix A, a vector $\mathbf{x}$, and then plot the original and rotated vectors with Altair.

```
# 90-degrees roration
A1 = np.array([[0, -1.0],
               [1, 0]])

# 180-degrees roration
A2 = np.array([[-1.0, 0],
               [0, -1.0]])

# 270-degrees roration
A3 = np.array([[0, 1.0],
               [-1.0, 0]])

x = np.array([[0, 2.0,],
              [0, 4.0,]])


y1 = A1 @ x
y2 = A2 @ x
y3 = A3 @ x
```

```
z = np.column_stack((x, y1, y2, y3))

df = pd.DataFrame({'dim-1': z[0], 'dim-2':z[1],
                   'rotation': ['original', 'original',
                                '90-degrees', '90-degrees',
                                '180-degrees', '180-degrees',
                                '270-degrees', '270-degrees'
                               ]})

df
```

	dim-1	dim-2	rotation
0	0.0	0.0	original
1	2.0	4.0	original
2	0.0	0.0	90-degrees
3	-4.0	2.0	90-degrees
4	0.0	0.0	180-degrees
5	-2.0	-4.0	180-degrees
6	0.0	0.0	270-degrees
7	4.0	-2.0	270-degrees

```
chart = alt.Chart(df).mark_line().encode(
    x=alt.X('dim-1', axis=alt.Axis(title='horizontal-axis')),
    y=alt.Y('dim-2', axis=alt.Axis(title='vertical-axis')),
    color='rotation')

base_coor(-5.0, 5.0) + chart
```



Projections

Projections are a fundamental type of linear (and affine) mappings for machine learning. If you have ever heard concepts like "embeddings", "low-dimensional representation", or "dimensionality reduction", they all are examples of projections. Even linear regression and principal component analysis are exemplars of projections. Thus, projections allow working with high-dimensional spaces (i.e., problems with many features or variables) more efficiently, by projecting such spaces into lower-dimensional spaces. this works because is often the case that a few dimensions contain most of the information to understand the relation between inputs and outputs. Moreover, projections can be represented as *matrices acting on vectors*.

Put simply, projections are *mappings from a space onto a subpace*, or from a set of vectors onto a subset of vectors. Additionally, projections are "idempotent", this is, the projection has the property to be *equal to its composition with itself*. In other words, when you wrap a projection $\phi(x) = y$ into itself as $\phi(\phi(x))$, the result does not change, i.e., $\phi(\phi(x)) = y$. Formally, for a vector space V and a vector subset $\mathrm{U} \subset \mathrm{V}$, we define a projection ϕ as:

$$\phi : \mathrm{V} \rightarrow \mathrm{U}$$

with

$$\phi^2 : \phi \circ \phi = \phi$$

Here we are concerned with the matrix representation of projections, which receive the special name of **projection matrices**, denoted as P_ϕ. By extension, projection matrices are also "idempotent":

$$\mathrm{P}_\phi^2 = \mathrm{P}_\phi \circ \mathrm{P}_\phi = \mathrm{P}_\phi$$

Projections onto lines

In Freudian psychoanalysis, *projection* is a defense mechanism of the "ego" (i.e., the sense of self), where a person denies the possession of an undesired characteristic while attributing it to someone else, i.e., "projecting" what we don't like of us onto others.

It turns out, that the concept of projection in mathematics is not that different from the Freudian one. Just make the following analogy: imagine you and foe of you are represented as vectors in a 2-dimensional cartesian plane, as x and y respectively. The way on which you would project yourself onto your foe is by tracing a perpendicular line (z) from you onto him. Why perpendicular? Because this is the shortest distance between you and him, hence, the most efficient way to project yourself onto him. Now, the projection would be "how much" of yourself was "splattered" onto him, which is represented by the segment p from the origin until the point where the perpendicular line touched your foe.

Now, recall that lines crossing the origin form subspaces, hence vector $\mathbf{y}$ is a subspace, and that perpendicular lines form $90°$ angles, hence the **projection is orthogonal**. More formally, we can define the projection of $\mathbf{x} \in \mathrm{R}^2$ onto subspace $\mathrm{U} \in \mathrm{R}^2$ formed by $\mathbf{y}$ as:

$$\phi_\mathrm{U}(\mathbf{x}) \in \mathrm{U}$$

Where $\phi_\mathrm{U}(\mathbf{x})$ must be the minimal distance between $\mathbf{x}$ and $\mathbf{y}$ (i.e., $\mathbf{x}$ and U), where distance is:

$$\|\mathbf{x} - \phi_\mathrm{U}(\mathbf{x})\|$$

Further, the resulting projection $\phi_\mathrm{U}(\mathbf{x})$ must lie in the span of U.

Therefore, we can conclude that $\phi_U(\mathbf{x}) = \alpha\mathbf{y}$, where alpha is a scalar in $\mathbb{R}$.

The formula to find the orthogonal projection (I'm skipping the derivation on purpose) $\phi_U(\mathbf{x})$ is:

$$\phi_U(\mathbf{x}) = \alpha\mathbf{y} = \frac{\langle \mathbf{x},\mathbf{y} \rangle}{\|\mathbf{y}\|^2}\mathbf{y} = \frac{\mathbf{y}^T \cdot \mathbf{x}}{\|\mathbf{y}\|^2}\mathbf{y}$$

In words: we take the dot product between $\mathbf{x}$ and $\mathbf{y}$, divide by the norm of $\mathbf{y}$, and multiply by $\mathbf{y}$. In this case, $\mathbf{y}$ is also known as a basis vector, so we can say that $\mathbf{x}$ is projected onto the basis $\mathbf{y}$.

Now, we want to express projections as matrices, i.e., as the matrix vector product $\mathrm{P}_\phi\mathbf{x}$. For this, recall that matrix-scalar multiplication is *commutative*, hence we can perform a little of algeabric manipulation to find:

$$\phi_U(\mathbf{x}) = \mathbf{y}\alpha = \mathbf{y}\frac{\mathbf{y}^T \cdot \mathbf{x}}{\|\mathbf{y}\|^2} = \frac{\mathbf{y} \cdot \mathbf{y}^T}{\|\mathbf{y}\|^2}\mathbf{x}$$

In this form, we can indeed express the projection as a matrix-vector multiplication, because $\mathbf{y} \cdot \mathbf{y}^T$ results in a symmetrix matrix, and $\|\mathbf{y}\|^2$ is a scalar, which means that it can be expressed as a matrix:

$$\mathrm{P}_\phi = \frac{\mathbf{y} \cdot \mathbf{y}^T}{\|\mathbf{y}\|^2}$$

In sum, the matrix P_ϕ will project any vector onto $\mathbf{y}$.

Let's use NumPy to find the projection P_ϕ from $\mathbf{x}$ onto a basis vector $\mathbf{y}$.

```
# base vector
y = np.array([[3],
              [2]])

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

```
P = (y @ y.T)/(y.T @ y)

print(f'Projection matrix for y:\n{P}')
```

```
Projection matrix for y:
[[0.69230769 0.46153846]
 [0.46153846 0.30769231]]
```

```
z = P @ x

print(f'Projection from x onto y:\n{z}')
```

```
Projection from x onto y:
[[2.07692308]
 [1.38461538]]
```

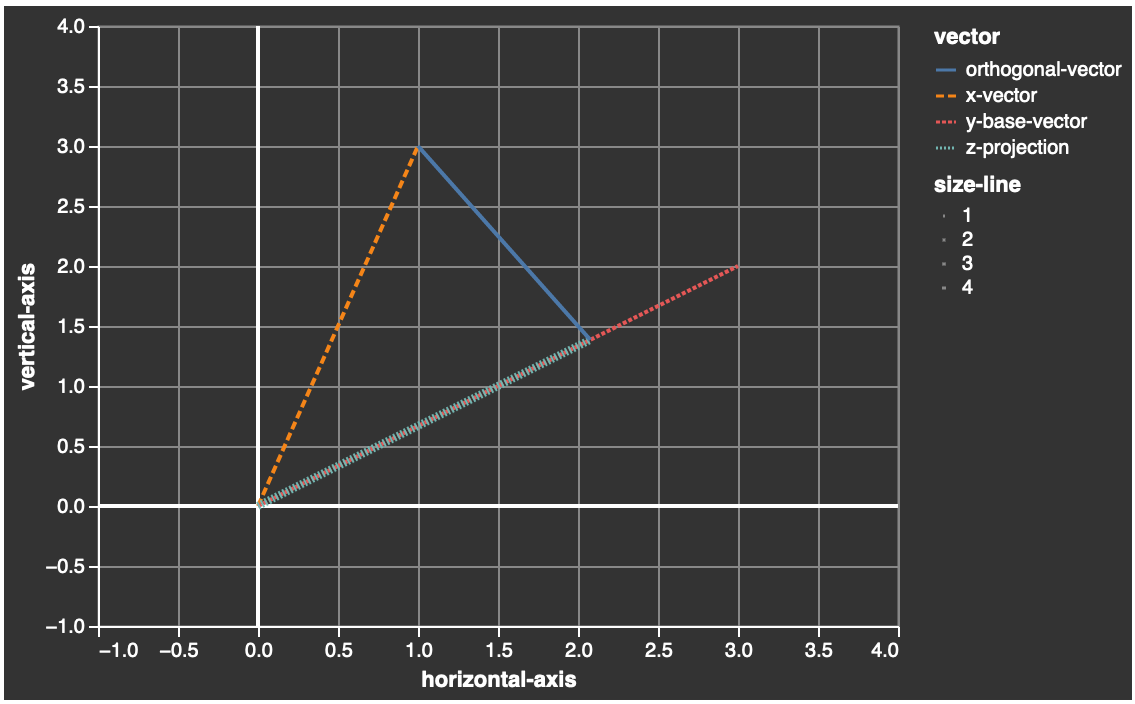
Let's plot the vectors to make things clearer

```
# origin coordinate space
o = np.array([[0],
              [0]])

v = np.column_stack((o, x, o, y, o, z, x, z))

df = pd.DataFrame({'dim-1': v[0], 'dim-2':v[1],
                   'vector': ['x-vector', 'x-vector',
                              'y-base-vector', 'y-base-vector',
                              'z-projection', 'z-projection',
                              'orthogonal-vector', 'orthogonal-vec
                  'size-line': [2, 2, 2, 2, 4, 4, 2, 2]})

df
```

	dim-1	dim-2	vector	size-line

	dim-1	dim-2	vector	size-line
0	0.000000	0.000000	x-vector	2
1	1.000000	3.000000	x-vector	2
2	0.000000	0.000000	y-base-vector	2
3	3.000000	2.000000	y-base-vector	2
4	0.000000	0.000000	z-projection	4
5	2.076923	1.384615	z-projection	4
6	1.000000	3.000000	orthogonal-vector	2
7	2.076923	1.384615	orthogonal-vector	2

```
chart = alt.Chart(df).mark_line().encode(
    x=alt.X('dim-1', axis=alt.Axis(title='horizontal-axis')),
    y=alt.Y('dim-2', axis=alt.Axis(title='vertical-axis')),
    color='vector',
    strokeDash='vector',
    size = 'size-line')

base_coor(-1.0, 4.0) + chart
```



Projections onto general subspaces

From the previous section, we learned that the projection matrix for the one-dimensional project of $\mathbf{x}$ onto U (i.e. $\mathbf{y}$) $\phi_{\mathrm{U}}(\mathbf{x})$ can be expressed as:

$$\mathrm{P}_{\phi} = \alpha\mathbf{y} = \frac{\mathbf{y} \cdot \mathbf{y}^T}{\|\mathbf{y}\|^2}$$

Which implies that we the projection is entirely defined in terms of the basis subspace. Now, we are interested in projections for the general case, this is, for **set of basis vectors** $\mathbf{y_1}, \cdots, \mathbf{y_m}$. By extension, we can define such projection as:

$$\phi_{\mathrm{U}}(\mathbf{x}) = \sum_{i=1}^{m} \mathbf{y}_i \alpha = \mathrm{Y}\alpha$$

Where Y is the matrix of basis vectors.

Nothing fancy going on here: we just need to take the sum for the product between each basis vector and α. As with the one-dimensional case, we want the projection to be the minimal distance from $\mathbf{x}$ onto Y, which we know implies orthogonal lines (or hyperplanes) connecting $\mathbf{x}$ with Y. The condition for orthogonality (again, I'm skipping the derivation on purpose) here equals:

$$\mathrm{Y}^T(\mathbf{x} - \mathrm{Y}\alpha) = 0$$

Now, recall what we really want is to find α (we know Y already). Therefore, with a bit of algeabric manipulation we can clear the expression above as:

$$\alpha = (\mathrm{Y}^T\mathrm{Y})^{-1}\mathrm{Y}^T\mathbf{x}$$

Such expression is known as the *pseudo-inverse* or *Moore–Penrose inverse* of Y. To work, it requires Y to be full rank (i.e., independent columns, which should be the case for basis). It can be used to solve linear regression problems, although you'll probably find the notation flipped as: $\alpha = (\mathrm{X}^T\mathrm{X})^{-1}\mathrm{X}^T\mathbf{y}$ (my bad choice of notation!).

Going back to our Freudian projection analogy, this is like a group

of people projecting themselves onto someone else, with that person representing a rough approximation of the character of the group.

Projections as approximate solutions to systems of linear equations

Machine learning prediction problems usually require to find a solution to systems of linear equations of the form:

$$\mathbf{Ax} = \mathbf{y}$$

In other words, to represent $\mathbf{y}$ as linear combinations of the columns of $\mathbf{A}$. Unfortunately, in most cases $\mathbf{y}$ is not in the column space of $\mathbf{A}$, i.e., *there is no way to find a linear combination of its columns to obtain the target* $\mathbf{y}$. In such cases, we can use orthogonal projections to find **approximate solutions** to the system. We usually denote approximated solutions for systems of linear equations as $\hat{\mathbf{y}}$. Now, $\hat{\mathbf{y}}$ will be in the span of the columns of $\mathbf{A}$ and will be the result of projecting $\mathbf{y}$ onto the subspace of the columns of $\mathbf{A}$. That solution will be the best (closest) approximation of $\mathbf{y}$ given the span of the columns of $\mathbf{A}$. In sum: **the approximated solution $\hat{\mathbf{y}}$ is the orthogonal projection of $\mathbf{y}$ onto $\mathbf{A}$.**

Matrix decompositions

In the Japanese manga/anime series *Fullmetal Alchemist*, Alchemy is understood as the metaphysical science of altering objects by manipulating its natural components, act known as *Transmutation* (Rensei). There are three steps to Transmutation: (1) *Comprehension*, to understand the atomic structure and properties of the object, (2) *Deconstruction*, to break down the structure of the object into its fundamental elements (3) *Reconstruction*, to use the natural flow of energy to reform the object into a new shape.

Metaphorically speaking, we can understand linear combinations and matrix decompositions in analogy to *Transmutation*. **Matrix decomposition** is essentially about to break down a matrix into simpler "elements" or matrices (deconstruction), which allows us to better understand its fundamental structure (comprehension). Linear combinations are essentially about taking the fundamental elements of a matrix (i.e., set of vectors) to generate a new object.

Matrix decomposition is also known as **matrix factorization**, in reference the fact that matrices can be broken down into simpler matrices, more on less in the same way that Prime factorization breaks down large numbers into simpler primes (e.g., $112 = 2 \times 2 \times 2 \times 2 \times 7$).

There are several important applications of matrix factorization in machine learning: clustering, recommender systems, dimensionality reduction, topic modeling, and others. In what follows I'll cover a selection of several basic and common matrix decomposition techniques.

LU decomposition

There are multiple ways to decompose or factorize matrices. One of the simplest ways is by decomposition a matrix into a **lower triangular matrix** and an **upper triangular matrix**, the so-called **LU or Lower-Upper decomposition**.

LU decomposition is of great interest to us since it's one of the methods computers use to solve linear algebra problems. In particular, LU decomposition is a way to represent **Gaussian Elimination in numerical linear algebra**. LU decomposition is flexible as it can be obtained from noninvertible or singular matrices, and from non-square matrices.

The general expression for LU decomposition is:

$$A = LU$$

Meaning that A can be represented as the product of the lower triangular matrix L and upper triangular matrix U. In the next sections, we explain the mechanics of the LU decomposition.

Elementary matrices

Our first step to approach LU decomposition is to introduce **elementary matrices**. When considering matrices as functions or mappings, we can associate special meaning to a couple of basic or "elementary" operations performed by matrices. Our starting point is the **identity matrix**, for instance:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

As we saw before, the identity matrix does not change the values of another matrix under multiplication:

$$AI = IA = A$$

Because it is essentially saying: *give me 1 of each column of the matrix*, i.e., return the original matrix. Now, consider the following matrix:

$$I_2 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The only thing we did to the I to obtain I_2 was to multiply the first row by 2. This can be considered an elementary operation. Here is another example:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Clearly, we can't obtain I_3 by multiplication only. From a column perspective, what we did was to add 2 times the second column to the first column. Alternatively, from a row perspective, we can say we added 2 times the first row to the second row.

One last example:

$$I_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{bmatrix}$$

From the column perspective, we added -3 times the third column to the second column. From the row perspective, we added -3 times the second row to the third row.

You can probably see the pattern by now: by performing simple or "elementary" column or row operations, this is, *multiplication* and *addition*, we can obtain any lower triangular matrix. This type of matrices are what we call **elementary matrices**. In a way, we can say elementary matrices "encode" fundamental column and row operations. To see this, consider the following generic matrix:

$$\mathrm{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Let's what happens when we multiply AI_3:

$$\mathrm{AI}_3 = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a+2b & b & c \\ d+2e & e & f \\ g+2h & h & i \end{bmatrix}$$

The result of AI_3 reflects the same elementary operations we performed on I to obtain I_3 from the **column perspective**: to add 2 times the second column to the first one.

Now consider what happens when we multiply from the left:

$$\mathrm{I}_3\mathrm{A} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} a & b & c \\ d+2a & e+2b & f+2c \\ g & h & i \end{bmatrix}$$

Now we obtain the same elementary operations we performed on I to obtain I_3 from the **row perspective**: to add 2 times the first row to the second one.

The inverse of elementary matrices

A nice property of elementary matrices, is that the inverse is simply the opposite operation. For instance, the inverse of I_2 is:

$$\begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This is because instead of multiplying the first row of I by 2, we divide it by 2. Similarly, the inverse of I_3 is:

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Again, instead of adding 2, we add -2 (or substract 2). Finally, the inverse of I_4 is:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{bmatrix}$$

The reason we care about elementary matrices and its inverse is that it will be fundamental to understand LU decomposition.

LU decomposition as Gaussian Elimination

Let's briefly recall Gaussian Elimination: it's an robust algorithm to solve systems of linear equations, by sequentially applying three elementary transformations:

1. Addition and subtraction of two equations (rows)
2. Multiplication of an equation (rows) by a number
3. Switching equations (rows)

Gaussian Elimination will reduce matrices to its **row echelon form**, which is an upper triangular matrix, with zero rows at the bottom, and zeros below the pivot for each column.

It turns out, there is a clever way to organize the steps from Gaussian Elimination: with **elementary matrices**.

Consider the following matrix A:

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 2 & -1 \\ 1 & 3 & 2 \end{bmatrix}$$

The first step consist of substracting two times row 1 from row 1. Before, we represented this operation as $R_2 - 2R_1$, and write down the result, which is:

$$\begin{bmatrix} 1 & 3 & 5 \\ 0 & -4 & -11 \\ 1 & 3 & 2 \end{bmatrix}$$

Alternatively, as we learned in the previous section, *we can represent row operations as multiplication by elementary matrices*, to obtain the same result. Since we want to substract 2 times the first row from the second, we need to (1) multiply from the left, and (2) add a -2 to the first element of the second row:

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 5 \\ 2 & 2 & -1 \\ 1 & 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 0 & -4 & -11 \\ 1 & 3 & 2 \end{bmatrix}$$

You don't have to believe me. Let's confirm this is correct with NumPy:

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

l1 = np.array([[1, 0, 0],
               [-2, 1, 0],
               [0, 0, 1]])


l1 @ A
```

```
array([[  1,   3,   5],
       [  0,  -4, -11],
```

```
       [  1,   3,   2]])
```

As you can see, the result is exactly what we obtained before by $R_2 - 2R_1$. But, we are not done. We still need to get rid of the 1 and 3 in the third row. For this, we would normally do $R_3 - R_1$ to obtain:

$$\begin{bmatrix} 1 & 3 & 5 \\ 0 & -4 & -11 \\ 0 & 0 & -3 \end{bmatrix}$$

Again, we can encode this using elementary matrices as:

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 5 \\ 2 & 2 & -1 \\ 1 & 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 0 & -4 & -11 \\ 0 & 0 & -3 \end{bmatrix}$$

Once again, let's confirm this with NumPy:

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

l2 = np.array([[1, 0, 0],
               [-2, 1, 0],
               [-1, 0, 1]])


l2 @ A
```

```
array([[  1,   3,   5],
       [  0,  -4, -11],
       [  0,   0,  -3]])
```

Indeed, the result is correct. At this point, we have reduced A to its **row echelon form**. We will call U to the resulting matrix from lA, as it is an *upper triangular matrix*. Hence, we arrived to the identity:

$$lA = U$$

This is not quite LU decomposition. To get there, we just need to multiply both sides of the equality by the inverse of l, that we will call L, which yields:

$$A = LU$$

There you go: we arrived to the LU decomposition expression. As a final note, recall that the inverse of l is:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Let's confirm with NumPy this works, by multiplying L by U:

```
# inverse of l
L = np.array([[1, 0, 0],
              [2, 1, 0],
              [1, 0, 1]])

# upper triangular resulting from Gaussian Elimination
U = np.array([[1, 3, 5],
             [0, -4, -11],
             [0, 0, -3]])


L @ U
```

```
array([[ 1,  3,  5],
       [ 2,  2, -1],
       [ 1,  3,  2]])
```

Indeed, we recover A by multiplying LU.

LU decomposition with pivoting

If you recall the three elementary operations allowed in Gaussian Elimination, we had: (1) multiplication, (2) addition, (3) switching. At this point, we haven't seen switching with LU decomposition. It turns out, that LU decomposition does not work

when switching or permutations of rows are required to solve a system of linear equations. Further, even when pivoting is not required to solve a system, the numerical stability of Gaussian Elimination when implemented in computers is problematic, and pivoting helps to tackle that issue as well.

Let's see a simple example of pivoting. Consider the following matrix A:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

In this case, we can't get rid of the first 1 in the second column by substraction. If we do that, we obtain:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Which is the opposite of what we want. A simple way to fix this is by switching rows 1 and 2 as:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

And then substracting row 1 from row 2 to obtain:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Bam! Problem fixed. Now, as with multiplication and addition, we can **represent permutations with matrices** as well. In particular, by using **permutation matrices**. For our previous example, we can do:

$$PA = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Let's confirm this is correct with NumPy

```
P = np.array([[0, 1],
              [1, 0]])
A = np.array([[0, 1],
```

```
                [1, 1]])
```

```
P @ A
```

```
array([[1, 1],
       [0, 1]])
```

It works. Now we can put all the pieces together and decompose A by using the following expression:

$$PA = LU$$

This is known as LU decomposition with pivoting. An alternative expression of the same decomposition is:

$$A = LUP$$

In Python, we can use SciPy to perform LUP decomposition by using the `linalg.lu` method. Let's decompose a larger matrix to make things more interesting.

```
from scipy.linalg import lu

A = np.array([[2, 1, 1, 0],
              [4, 3, 3, 1],
              [8, 7, 9, 5],
              [6, 7, 9, 8]])
```

```
P, L, U = lu(A)
```

```
print(f'Pivot matrix:\n{P}')
```

```
Pivot matrix:
[[0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]]
```

```
print(f'Lower triangular matrix:\n{np.round(L, 2)}')
```

```
Lower triangular matrix:
[[ 1.    0.    0.    0.  ]
 [ 0.75  1.    0.    0.  ]
 [ 0.5  -0.29  1.    0.  ]
 [ 0.25 -0.43  0.33  1.  ]]
```

```
print(f'Upper triangular matrix:\n{np.round(U, 2)}')
```

```
Upper triangular matrix:
[[ 8.    7.    9.    5.  ]
 [ 0.    1.75  2.25  4.25]
 [ 0.    0.   -0.86 -0.29]
 [ 0.    0.    0.    0.67]]
```

We can confirm the decomposition is correct by multiplying the obtained matrices

```
A_recover = np.round(P @ L @ U, 1)
print(f'PLU multiplicatin:\n{A_recover.astype(int)}')
```

```
PLU multiplicatin:
[[2 1 1 0]
 [4 3 3 1]
 [8 7 9 5]
 [6 7 9 8]]
```

We recover A perfectly.

QR decomposition

QR decomposition or **QR factorization**, is another very relevant decomposition in the context of numerical linear algebra. As with LU decomposition, It can be used to solve systems of linear equations like least square problems and to find eigenvalues of a general matrix.

QR decomposition works by decomposing A into an orthogonal matrix Q, and a upper traingular matrix R as:

$$A = QR$$

Next, we will review a few concepts to properly explain QR decomposition.

Orthonormal basis

In previous sections we learned about *basis* and *orthogonal basis*. Specifically, we said that a set of n linearly independent column vectors with n elements forms a **basis**. We also said that a pair of vectors $\mathbf{x}$ and $\mathbf{y}$ are **orthogonal** if their inner product is zero, $\langle x, y \rangle = 0$ or $\mathbf{x}^T\mathbf{y} = 0$. Consequently, *a set of orthogonal vectors form an orthogonal basis for a matrix A and for the vector space spanned by such matrix*.

To go from orthogonal basis vectors to **orthonomal basis vectors**, we just need to divide each vector by its lenght or norm. When we divide a basis vector by its norm we obtain a **unit basis vector**. More formally, a set of vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n$ is orthonormal if:

$$\mathbf{x}_i^T\mathbf{x}_j = \begin{cases} 0, & \text{when} \quad i \neq j \quad \text{orthogonal vectors} \\ 1, & \text{when} \quad i = j \quad \text{unit vectors} \end{cases}$$

In words: when we take the inner product of a pair of orthogonal vectors, it results in 0; when we take the inner product of a vector with itself, it results in 1.

For instance, consider $\mathbf{x}$ and $\mathbf{y}$:

$$\mathbf{x} = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix} \mathbf{y} = \begin{bmatrix} -4 \\ 3 \\ 2 \end{bmatrix}$$

To obtain the normalized version of $\mathbf{x}$ or $\mathbf{y}$, we divide by its Euclidean norm as:

$$\hat{\mathbf{x}} = \frac{\mathbf{x}}{\|\mathbf{x}\|}$$

We add a "hat" to the normalized vector to distinguish it from the un-normalized version.

Let's try an example with NumPy. I'll define vectors $\mathbf{x}, \mathbf{y} \in \mathrm{R}^3$, compute its Euclidean norm, and then perform element-wise division $\frac{\mathbf{x}}{\|\mathbf{x}\|}$:

```
x, y = np.array([[3],[4],[0]]), np.array([[-4],[3],[2]])

# euclidean norm of x and y
x_norm = np.linalg.norm(x, 2)
y_norm = np.linalg.norm(y, 2)

# normalized x or unit vector
x_unit = x * (1/x_norm)
y_unit = y * (1/y_norm)


print(f'Euclidean norm of x:\n{x_norm}\n')
print(f'Euclidean norm of y:\n{y_norm}\n')

print(f'Normalized x:\n{x_unit}\n')
print(f'Normalized y:\n{y_unit}')
```

```
Euclidean norm of x:
5.0

Euclidean norm of y:
5.385164807134504

Normalized x:
[[0.6]
 [0.8]
 [0. ]]

Normalized y:
[[-0.74278135]
 [ 0.55708601]
 [ 0.37139068]]
```

We can confirm that the Euclidean norm of the normalized versions of $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ equals 1 by:

```
print(f'Euclidean norm of normalized x:\n{np.round(np.linalg.norm(
print(f'Euclidean norm of normalized y:\n{np.round(np.linalg.norm(
```

```
Euclidean norm of normalized x:
1.0

Euclidean norm of normalized y:
1.0
```

Taking $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ as a set, we can confirm the conditions for the definition of orthonormal vectors are correct.

```
print(f'Inner product normalized vectors:\n{np.round(x_unit.T @ y_
print(f'Inner product normalized x with itself:\n{np.round(x_unit.
print(f'Inner product normalized y with itself:\n{np.round(y_unit.
```

```
Inner product normalized vectors:
[[-0.]]

Inner product normalized x with itself:
[[1.]]

Inner product normalized y with itself:
[[1.]]
```

Sets of vectors can be represented as matrices. **We denote as $\mathbf{Q}$ the special case of a matrix composed of orthonormal vectors**. The same properties we defined for sets of vectors hold when represented in matrix form.

Orthonormal basis transpose

A nice property of $\mathbf{Q}$ is that *the matrix product with its transpose equals the identity*:

$$\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$$

This is true even when Q is not square. Let's see this with the $Q \in R^{3 \times 3}$ orthonormal matrix resulting from stacking $\hat{x}$ and $\hat{y}$.

```
Q = np.column_stack((x_unit, y_unit))
print(f'Orthonormal matrix Q:\n{Q}')
```

```
Orthonormal matrix Q:
[[ 0.6        -0.74278135]
 [ 0.8         0.55708601]
 [ 0.          0.37139068]]
```

Now we confirm $Q^TQ = I$

```
np.round(Q.T @ Q,1)
```

```
array([[ 1., -0.],
       [-0.,  1.]])
```

This property will be useful for several applications. For instance, the *coupling matrix* or *correlation matrix* of a matrix A equals A^TA. If we are able to transform the vectors of A into orthonormal vectors, such expressions reduces to $Q^TQ = I$. Other applications are the Fourier series and Least Square problems (as we will see later).

Gram-Schmidt Orthogonalization

In the previous section, I selected orthogonal vectors to illustrate the idea of an orthonormal basis. Unfortunately, in most cases, matrices are not full rank, i.e., not composed of a set of orthogonal vectors. Fortunately, there are ways to *transform a set of non-orthogonal vectors into orthogonal vectors*. This is the so-called **Gram-Schmidt orthogonalization procedure**.

The Gram-Schmidt orthogonalization consist of *taking the vectors of a matrix, one by one, and making each subsequent vector orthonormal to the previous one*. This is easier to grasp with an example. Consider the matrix A:

$$\begin{bmatrix} 2 & 1 & -2 \\ 7 & -3 & 1 \\ -3 & 5 & -1 \end{bmatrix}$$

What we want to do, is to find the set of orthonormal vectors $\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$, starting from the columns of A, i.e., $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3$. We can select any vector to begin with. Recall that we normalize vectors by dividing by its norm as:

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{\|\mathbf{a}\|}$$

Let's approach this with NumPy:

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

A simple way to check the columns of A are not orthonormal is to compute $\mathrm{A}^T\mathrm{A}$, which should be equal to the identity in the orthonormal case.

```
A.T @ A
```

```
array([[ 62, -34,   6],
       [-34,  35, -10],
       [  6, -10,   6]])
```

To build our orthogonal set, we begin by denoting $\mathbf{a}_1$ as $\mathbf{q}_1$.

Our **first step** is to generate the vector $\mathbf{q}_2$ from $\mathbf{a}_2$ such that is orthogonal to $\mathbf{q}_1$ (i.e., $\mathbf{a}_1$). To do this, we start with $\mathbf{a}_2$ and subtract its projection along $\mathbf{q}_1$, which yields the following expression:

$$\mathbf{q}_2 = \mathbf{a}_2 - \frac{\mathbf{q}_1^T\mathbf{a}_2}{\mathbf{q}_1^T\mathbf{q}_1}\mathbf{q}_1$$

Think in this expression carefully. What are we doing, is to

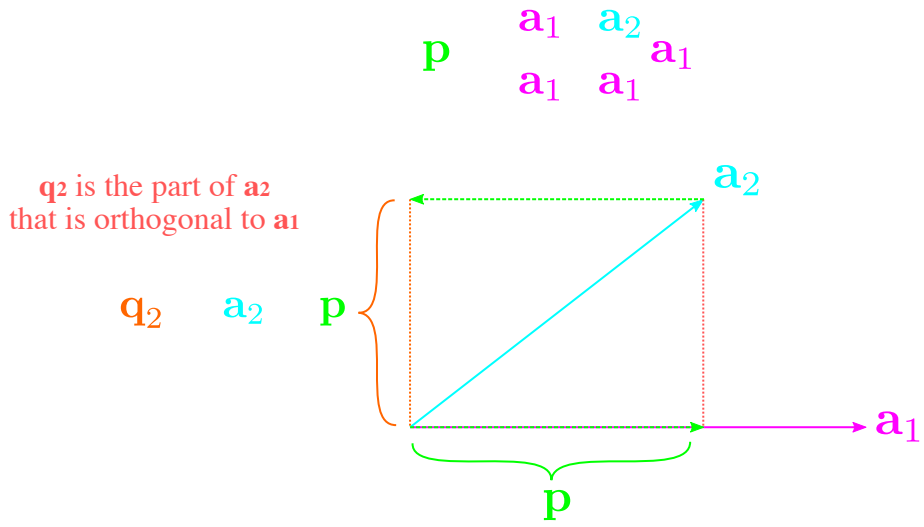
subtract $\frac{\mathbf{q}_1^T\mathbf{a}_2}{\mathbf{q}_1^T\mathbf{q}_1}$ *times* the first column from the second column. Let's denote $\frac{\mathbf{q}_1^T\mathbf{a}_2}{\mathbf{q}_1^T\mathbf{q}_1}$ as α, then, we can rewrite our expression as:

$$\mathbf{q}_2 = \mathbf{a}_2 - \alpha\mathbf{q}_1$$

As we will see, α is a scalar, so effectively we are substracting an scaled version of column one from column two. The figure below express geometrically, what I have been saying: the *non-orthogonal* $\mathbf{a}_2$ is projected onto $\mathbf{q}_1$. Then, we subtract the projection $\mathbf{p}$ from $\mathbf{a}_2$ to obtain $\mathbf{q}_2$ which is orthogonal to $\mathbf{q}_1$ as you can appreciate visually (recall $\mathbf{a}_1 = \mathbf{q}_1$).

Keep these ideas in mind as it will be important later for QR decomposition.

Fig. 16: Orthogonalization



Here we ignore normalization for simplicity

Let's compute $\mathbf{q}_2$ now:

```
q1 = A[:, 0]
a2 = A[:, 1]
q2 = a2 - ((q1.T @ a2)/(q1.T @ q1)) * q1
```

Let's check that $\mathbf{q}_1$ and $\mathbf{q}_2$ are actually orthogonal. If so, their dot product should be 0.

```
np.round(q1 @ q2, 2)
```

```
-0.0
```

Next, we need to generate $\mathbf{q}_3$ from $\mathbf{a}_3$. This time, we want $\mathbf{q}_3$ to be orthogonal to both $\mathbf{q}_1$ and $\mathbf{q}_2$. Therefore, we need to subtract its projection along $\mathbf{q}_1$ and $\mathbf{q}_2$, which yields:

$$\mathbf{q}_3 = \mathbf{a}_3 - \frac{\mathbf{q}_1^T\mathbf{a}_3}{\mathbf{q}_1^T\mathbf{q}_1}\mathbf{q}_1 - \frac{\mathbf{q}_2^T\mathbf{a}_3}{\mathbf{q}_2^T\mathbf{q}_2}\mathbf{q}_2$$

```
a3 = A[:, 2]
q3 = a3 - (((q1.T @ a3)/(q1.T @ q1)) * q1) - (((q2.T @ a3)/(q2.T @
```

Verify orthogonality

```
print(f'Dot product q1 and q3:\n{np.round(q1 @ q3, 1)}\n')
print(f'Dot product q2 and q3:\n{np.round(q2 @ q3, 1)}')
```

```
Dot product q1 and q3:
-0.0

Dot product q2 and q3:
0.0
```

We can put $\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$ into Q':

```
Q_prime = np.column_stack((q1, q2, q3))
print(f'Orthogonal matrix Q:\n{Q_prime}')
```

```
Orthogonal matrix Q:
[[ 2.          2.09677419 -1.33333333]
 [ 7.          0.83870968  0.66666667]
```

```
  [-3.          3.35483871  0.66666667]]
```

The reason we call this matrix Q' is that although vectors are orthogonal, they are not normal.

```
Q_norms = np.linalg.norm(Q_prime, 2, axis=0)
print(f'Norms of vectors in Q-prime:\n{Q_norms}')
```

```
Norms of vectors in Q-prime:
[7.87400787 4.04411161 1.63299316]
```

We rename Q' to Q by normalizing its vectors.

```
Q = Q_prime / Q_norms
np.linalg.norm(Q , 2, axis=0)
```

```
array([1., 1., 1.])
```

To confirm we did this right, let's evaluate Q^TQ, that should return the identity:

```
np.round(Q.T @ Q, 1)
```

```
array([[ 1., -0., -0.],
       [-0.,  1.,  0.],
       [-0.,  0.,  1.]])
```

There you go: we performed Gram-Schmidt orthogonalization of A

QR decomposition as Gram-Schmidt Orthogonalization

Gaussian Elimination can be represented as LU decomposition. Similarly, **Gram-Schmidt Orthogonalization can be represented as QR decomposition**.

We learned Q is an orthonormal matrix. Now let's examine R, which is an upper triangular matrix. In LU decomposition, we used **elementary matrices** to perform *row operations*. Similarly, in the

case of QR decomposition, we will use **elementary matrices** to perform *column operations*. We used a lower triangular matrix to perform row operations in LU decomposition by multiplying A from the *left side*. This time, we will use an upper triangular matrix to perform column operations in QR decomposition by multiplying A from the *right side*.

Once again, our starting point is the identity matrix. The idea is to alter the identity with the operations we want to perform over A. Consider the matrix from our previous example:

$$A = \begin{bmatrix} 2 & 1 & -2 \\ 7 & -3 & 1 \\ -3 & 5 & -1 \end{bmatrix}$$

What we did in out first step, wast to subtract $\alpha = \frac{\mathbf{a}_1 \cdot \mathbf{a}_2}{\mathbf{a}_1 \cdot \mathbf{a}_1}$ of column $\mathbf{a}_1$ from column $\mathbf{a}_2$. Let's compute α first:

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

a1 = A[:, 0]
a2 = A[:, 1]

alpha = (a1.T @ a2)/(a1.T @ a1)

print(f'alpha factor:{np.round(alpha, 2)}')
```

```
alpha factor:-0.55
```

Now we need to subtract $\alpha = -0.55$ times $\mathbf{a}_1$ from $\mathbf{a}_2$. We can represent this operation with an **elementary matrix**, by doing applying the same operations the identity:

$$l = \begin{bmatrix} 1 & -0.55 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Next, we have to subtract $\beta = \frac{\mathbf{a}_1 \cdot \mathbf{a}_3}{\mathbf{a}_1 \cdot \mathbf{a}_1}$ times column $\mathbf{a}_1$ and $\gamma = \frac{\mathbf{a}_2 \cdot \mathbf{a}_3}{\mathbf{a}_2 \cdot \mathbf{a}_2}$ times $\mathbf{a}_2$ from $\mathbf{a}_3$. Let's compute the new β and γ:

```
a3 = A[:, 2]

beta = (a1.T @ a3)/(a1.T @ a1)
gamma = (a2.T @ a3)/(a2.T @ a2)

print(f'beta factor:{np.round(beta, 2)}')
print(f'gamma factor:{np.round(gamma, 2)}')
```

```
beta factor:0.1
gamma factor:-0.29
```

We can add this operations to our elementary matrix by subtracting 0.1 times the first column from the third, and -0.29 times the second from the third:

$$l = \begin{bmatrix} 1 & -0.55 & 0.1 \\ 0 & 1 & -0.29 \\ 0 & 0 & 1 \end{bmatrix}$$

The last step is to normalize each vector of l

```
l = np.array([[1, alpha, beta],
              [0, 1, gamma],
              [0, 0, 1]])
```

At this point, we should be able to recover A:

$$\begin{bmatrix} 2 & 1 & -2 \\ 7 & -3 & 1 \\ -3 & 5 & -1 \end{bmatrix}$$

As the matrix product of Q' and l

```
print(f'Q-prime and l product:\n{np.round(Q_prime @ l)}')
```

```
Q-prime and l product:
[[ 2.  1. -2.]
 [ 7. -3.  1.]
 [-3.  5. -1.]]
```

It works! Now, to recover Q will be difficult because of numerical stability and approximation issues in how we have computed things. Actually, if you remove the rounding from `np.round(Q_prime @ l)` you will obtain different numbers. Fortunately, there is no need to compute Q and R by hand. We follow the previous steps merely for pedagogical purposes. In NumPy, we can compute the QR decomposition as:

```
Q_1, R = np.linalg.qr(A)
```

Let's compare our Q with Q_1

```
print(f'Q:\n{Q}\n')
print(f'Q:\n{Q_1}')
```

```
Q:
[[ 0.25400025  0.51847585 -0.81649658]
 [ 0.88900089  0.20739034  0.40824829]
 [-0.38100038  0.82956136  0.40824829]]

Q:
[[-0.25400025  0.51847585 -0.81649658]
 [-0.88900089  0.20739034  0.40824829]
 [ 0.38100038  0.82956136  0.40824829]]
```

The numbers are the same, but some signs are flipped. This stability and approximation issues is why you probably always want to use NumPy functions (when available).

Determinant

If matrices had personality, the **determinant** would be the personality trait that reveals most information about the matrix character. The determinant of a matrix is a single number that tells **whether a matrix is invertible or singular**, this is, whether its columns are linearly independent or not, which is one of the most important things you can learn about a matrix. Actually, the name "determinant" refers to the property of "determining" if the matrix is singular or not. Specifically, for an square matrix $A \in R^{n \times n}$, a determinant equal to 0, denoted as $\det(A = 0)$, implies *the matrix is singular* (i.e., noninvertible), whereas a determinant equal to 1, denoted as $\det(A) = 1$, implies the *matrix is not singular* (i.e., invertible). Although determinants can reveal if matrices are singular with a single number, it's not used for large matrices as Gaussian Elimination is faster.

Recall that matrices can be thought of as function action on vectors or other matrices. Thus, the determinant can also be considered a linear mapping of a matrix A onto a single number. But, what does that number mean? So far, we have defined determinants based on their utility of determining matrix invertibility. Before going into the calculation of determinants, let's examine determinants from a geometrical perspective to gain insight into the meaning of determinants.

Determinant as measures of volume

From a geometric perspective, determinants indicate the sign **area of a parallelogram** (e.g., a rectangular area) and the sign **volume of the parallelepiped**, for a matrix whose columns consist of the basis vectors in Euclidean space.

Let's parse out the above phrase: the sign area indicates the absolute value of the area, and the sign volume equals the absolute value of the volume. You may be wondering why we need to take the absolute value since real-life objects can't have negative area or

volume. In linear algebra, we say the area of a parallelogram is **negative** when the vectors forming the figure are *clockwise oriented* (i.e., negatively oriented), and **positive** when the vectors forming the figure are *counterclockwise oriented* (i.e., positively oriented).

Here is an example of a matrix A with vectors *clockwise* or *negatively* oriented:

$$A = \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix}$$

The elements of the first column, indicate the first vector of the matrix, while the elements of the second column, the second vector of the matrix. Therefore, when we measure the area of the parallelogram formed by the pair of vectors, we move from left to right, i.e., *clockwise*, meaning that the vectors are **negatively oriented**, and the **area of the matrix will be negative**.

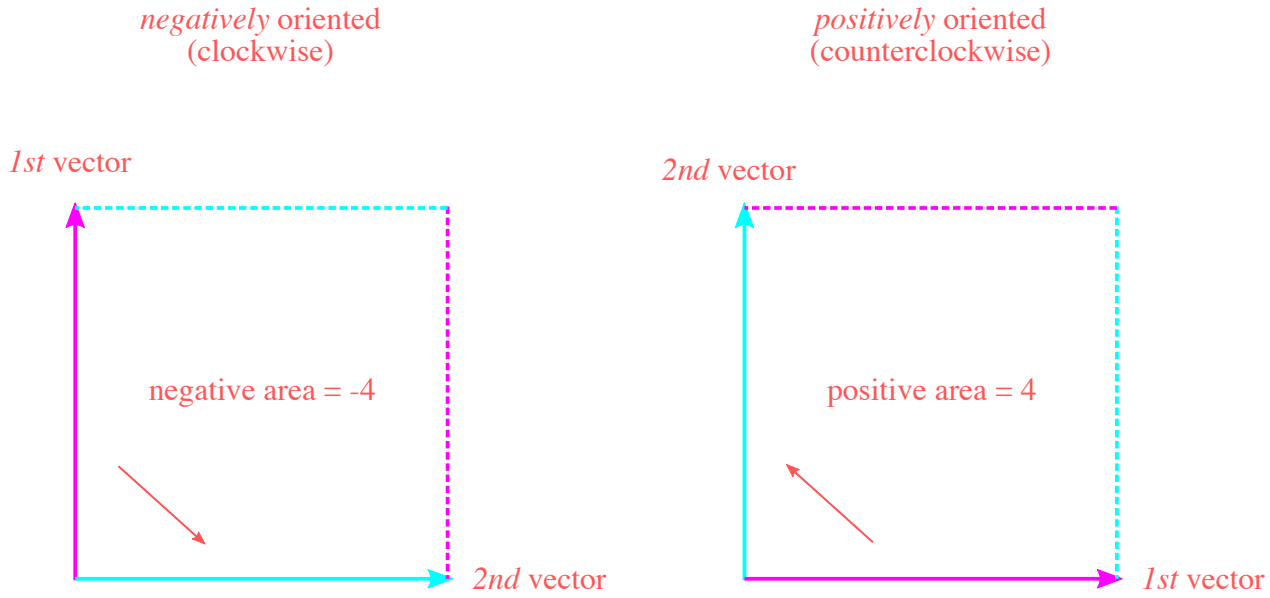
Here is the same matrix A with vectors *counterclockwise* or *positively* oriented:

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

Again, the elements of the *first column*, indicate the *first vector* of the matrix, while the elements of the *second column*, the *second vector* of the matrix. Therefore, when we measure the area of the parallelogram formed by the pair of vectors, we move from *right to left*, i.e., *counterclockwise*, meaning that the vectors are **positively oriented**, and the **area of the matrix will be positive**.

The figure below exemplifies what I just said.

Fig. 17: Vector orientation



The situation for the sign volume of the parallelepiped is no different: when the vectors are *counterclockwise* oriented, we say the vectors are *positively oriented* (i.e., positive volume); when the vectors are *clockwise* oriented, we say the vectors are *negatively oriented* (i.e., negative volume).

The 2 X 2 determinant

Recall that matrices are invertible or nonsingular when their columns are linearly independent. By extension, the determinant allow us to whether the columns of a matrix a linearly independent. To understand this method, let's examine the 2×2 special case first.

Consider a square matrix as:

$$A = \begin{bmatrix} 1 & 4 \\ 2 & 8 \end{bmatrix}$$

How can we decide whether the columns are linearly independent? A strategy that I often use in simple cases like this, is just to examine whether the second column equals the first column times some factor. In the case of A is easy to see that the second column equals four times the first column, so the columns are linearly *dependent*. We can express such criteria by comparing the *elementwise division* between each element of the second column by

each element of the first column as:

$$\begin{bmatrix} \frac{4}{1} = \frac{8}{2} \end{bmatrix} = \begin{bmatrix} 4 = 4 \end{bmatrix}$$

We obtain that both entries equal 4, meaning that the second column can be divided exactly by the first column (i.e., linearly *dependent*).

Consider this matrix now:

$$\mathrm{B} = \begin{bmatrix} 0 & 4 \\ 0 & 8 \end{bmatrix}$$

Let's try again our method for B:

$$\begin{bmatrix} \frac{4}{0} = \frac{8}{0} \end{bmatrix} = \begin{bmatrix} \text{undef} = \text{undef} \end{bmatrix}$$

Now we got into a problem because division by 0 is undefined, so we can determine the relationship between columns of B. Yet, by inspection, we can see the first column is simply 0 times the second column, therefore linearly dependent. Here is when **determinants** come to the rescue.

Consider the generic matrix:

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

According to our previous strategy, we had:

$$\frac{b}{a} = \frac{d}{c}$$

This is, we tested the elementwise division of the second column by the first column. Before, we failed because of division, so we probably want a method that does not involve it. Notice that we can rearrange our expression as:

$$ad = bc$$

Let's try again with this method for A:

$$\left[1 \times 8 = 4 \times 2\right]\left[8 = 8\right]$$

And for B:

$$\left[0 \times 8 = 4 \times 0\right]\left[0 = 0\right]$$

It works. Indeed, $ad = bc$ are equal for both matrices, A and B, meaning their columns are linearly dependent. Finally, notice that we can rearange all the terms on one side of the equation as:

$$(ad) - (bc) = 0$$

There you go: the above expression is what is known as the **determinant of a matrix**. We denote the determinant as:

$$|\mathrm{A}| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = (ad) - (bc)$$

Or

$$\det(\mathrm{A}) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = (ad) - (bc)$$

The N X N determinant

As matrices larger, computing the determinant gets more complicated. Consider the 3×3 case as:

$$|\mathrm{A}| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

The problem now is that linearly independent columns can be either: (1) multiples of another column, and (2) linear combinations of pairs of columns. The determinant for a 3×3 is:

$$|\mathrm{A}| = aei - afh + bfg - bdi + cdh - ceg$$

Such expression is hard to memorize, and it will get even more complicated for larger matrices. For instance, the 4×4 entails 24 terms. As with most things in mathematics, there is a general formula to express the determinant compactly, which is known as the

Leibniz's formula:

$$|\mathrm{A}| = \sum_{\sigma} \operatorname{sign}(\sigma) \prod_{i=1}^{n} a_{\sigma(i),i}$$

Where σ computes the permutation for the rows and columns vectors of the matrix. Is of little importance for us to break down the meaning of this formula since we are interested in its applicability and conceptual value. What is important to notice, is that for an arbitrary square $n \times n$ matrix, we will have $n!$ terms to sum over. For instance, for a 10×10 matrix, $10! = 3,628,800$, which is a gigantic number considering the size of the matrix. In machine learning, we care about matrices with thousands or even millions of columns, so there is no use for such formula. Nonetheless, this does not mean that the determinant is useless, but the direct calculation with the above algebraic expression is not used.

Determinants as scaling factors

When we think in matrices as linear mappings, this is, as functions applied to vectors (or vectors spaces), the determinant acquires an intuitive geometrical interpretation: **as the factor by which areas are scaled under a mapping**. Plainly, if you do a mapping or transformation, and the area increases by a factor of 3, then the determinant of the transformation matrix equals 3. Consider the matrix A and the basis vector $\mathbf{x}$:

$$\mathrm{A} = \begin{bmatrix} 4 & 0 \\ 0 & 3 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

Is easy to see that the parallelogram formed by the basis vectors of $\mathbf{x}$ is $1 \times 1 = 1$. When we apply $\mathrm{A}\mathbf{x}$, we get:

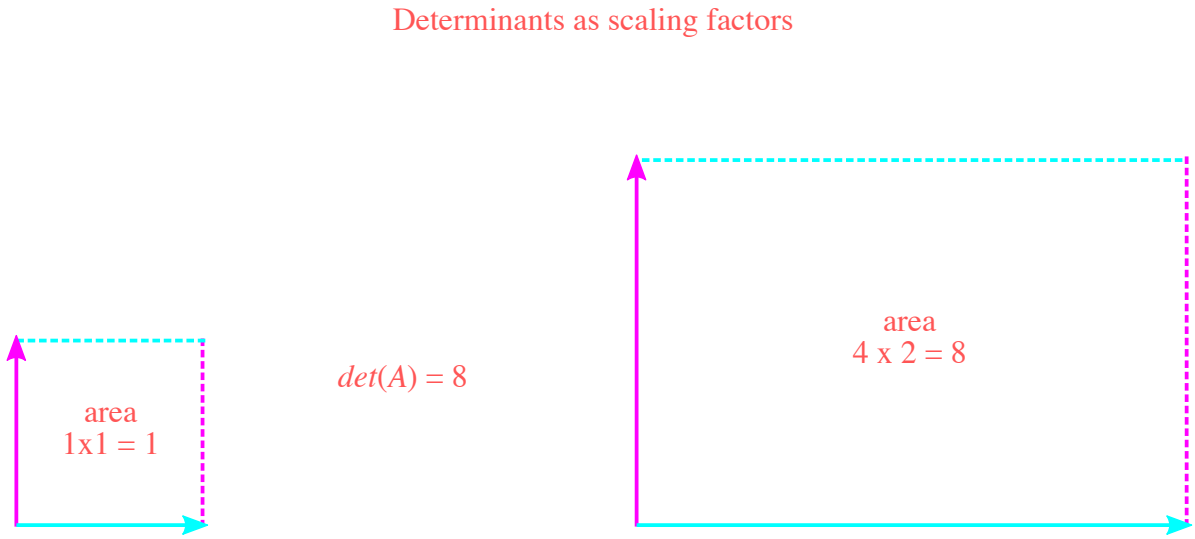
```
A = np.array([[4, 0],
              [0, 3]])

x = np.array([1,1])
```

```
A @ x.T
```

```
array([4, 3])
```

Meaning that the vertical axis was scaled by 4 and the horizontal axis by 3, hence, the new parallelogram has area $4 \times 3 = 12$. Since the new area has increased by a factor of 12, the determinant $|A| = 12$. Although we exemplified this with the basis vectors in x, the determinant of A for mappings of the entire vector space. The figure below visually illustrates this idea.

Fig. 18: Determinants



The importance of determinants

Considering that calculating the determinant is not computationally feasible for large matrices and that we can determine linear independence via Gaussian Elimination, you may be wondering what's the point of learning about determinants in the first place. I also asked myself more than once. It turns out that determinants play a crucial conceptual role in other topics in matrix decomposition, particularly eigenvalues and eigenvectors. Some books I reviewed devote a ton of space to determinants, whereas others (like Strang's Intro to Linear Algebra) do not. In any case, we study determinants mostly because of its conceptual value to better understand linear algebra and matrix decomposition.

Eigenthings

Eigenvectors, eigenvalues, and their associated mathematical objects and properties (which I call "Eigen-things") have important applications in machine learning like Principal Component Analysis (PCA), Spectral Clustering (K-means), Google's PageRank algorithm, Markov processes, and others. Next, we will review several of these "eigen-things".

Change of basis

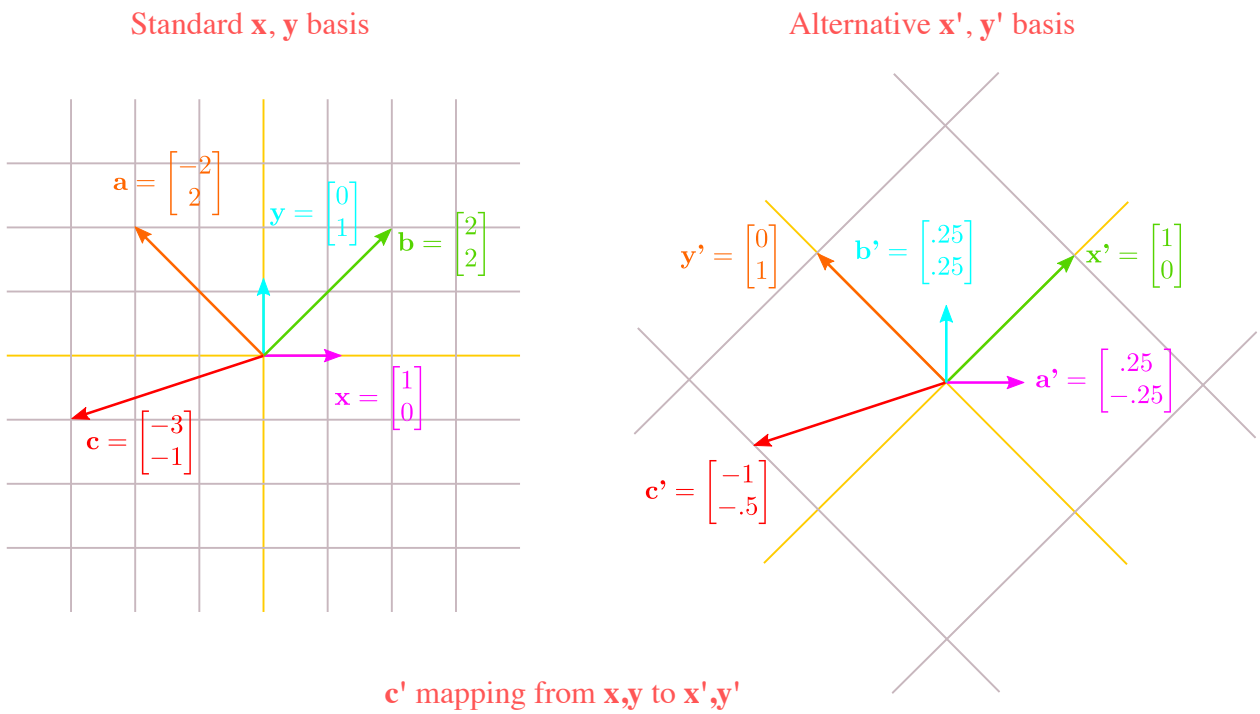
Previously, we said that a set of n linearly independent vectors with n elements forms a **basis** for a vector space. For instance, we say that the *orthogonal* pair of vectors $\mathbf{x}$ and $\mathbf{y}$ (or horizontal and vertical axes), describe the Cartesian plane or R^2 space. Further, if we think in the $\mathbf{x}$ and $\mathbf{y}$ pair as unit vectors, then we can describe any vector in R^2 as a linear combination of $\mathbf{x}$ and $\mathbf{y}$. For example, the vector:

$$\mathbf{c} = \begin{bmatrix} -3 \\ -1 \end{bmatrix}$$

Can be described as scaling the unit vector $\mathbf{x} = \begin{bmatrix} 1 & 0 \end{bmatrix}$ by -3, and scaling the unit vector $\mathbf{y} = \begin{bmatrix} 0 & 1 \end{bmatrix}$ by -1.

If you are like me, you have probably gotten use to the idea of describing any 2-dimensional space as $\mathbf{x}$ and $\mathbf{y}$ coordinates, with $\mathbf{x}$ lying perfectly horizontal and $\mathbf{y}$ perpendicular to it, as if this were the only natural way of thinking on coordinates in space. It turns out, there is nothing "natural" about it. You could literally draw a pair of orthogonal vectors on any orientation in space, define the first one as $\mathbf{x}' = \begin{bmatrix} 1 & 0 \end{bmatrix}$, and the second one as $\mathbf{y}' = \begin{bmatrix} 0 & 1 \end{bmatrix}$, and that would be perfectly fine. It may look different, but every single mathematical property we have studied so far about vectors would hold. For instance, in Fig 19. the **alternative coordinates** $\mathbf{x}'$ and $\mathbf{y}'$ are equivalent to the vectors $\mathbf{a} = \begin{bmatrix} -2 & 2 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} 2 & 2 \end{bmatrix}$, in the standard $\mathbf{x}$ and $\mathbf{y}$ coordinates.

Fig. 19: Change of basis



The question now is how to "move" from one set of basis vectors to the other. The answer is with **linear mappings**. We know already that $\mathbf{x'}, \mathbf{y'}$ equals to $\mathbf{a} = \begin{bmatrix} -2 & 2 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} 2 & 2 \end{bmatrix}$ in $\mathbf{x}, \mathbf{y}$ coordinates. To find the values of $\mathbf{x}, \mathbf{y}$ in $\mathbf{x'}, \mathbf{y'}$, we need to take the **inverse of T**. Think about it in this way: we represented $\mathbf{x'}=\mathbf{a}, \mathbf{y'}=\mathbf{a}$ in $\mathbf{x}, \mathbf{y}$ by scaling its unit vectors by the transformation matrix T as:

$$TA = \begin{bmatrix} -2 & 2 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 2 \\ 2 & 2 \end{bmatrix}$$

Now, to do the *opposite*, i.e., to "translate" the values of the coordinates $\mathbf{x}, \mathbf{y}$ to values in $\mathbf{x'}, \mathbf{y'}$, we scale $\mathbf{x}, \mathbf{y}$ by the inverse of T as:

```
# Transformation or mapping matrix
T = np.array([[2, -2],
              [2,  2]])

# Inverse of T
T_i = np.linalg.inv(T)
```

```
# x, y vectors
A = np.array([[1, 0],
              [0,1]])

print(f"x, y vectors in x', y' coordinates:\n{T_i @ A}")
```

```
x, y vectors in x', y' coordinates:
[[ 0.25  0.25]
 [-0.25  0.25]]
```

That is our answer: $\mathbf{x}, \mathbf{y}$ equals to $\begin{bmatrix} 0.25 & -0.25 \end{bmatrix}$ and $\begin{bmatrix} 0.25 & 0.25 \end{bmatrix}$ in $\mathbf{x'}, \mathbf{y'}$ coordinate space. **Fig. 19** illustrate this as well.

This may become clearer by mapping $\mathbf{c} = \begin{bmatrix} -3 & -1 \end{bmatrix}$ in $\mathbf{x}, \mathbf{y}$, onto $\mathbf{c'}$ in $\mathbf{x'}, \mathbf{y'}$ alternative coordinates. To do the mapping, again, we need to multiply $\mathbf{c}$ by T^{-1}. Let's try this out with NumPy:

```
# vector to map
a = np.array([[-3],[-1]])
print(f'Vector a=[1,3] in x\' and y\' basis:\n{T_i@a}')
```

```
Vector a=[1,3] in x' and y' basis:
[[-1. ]
 [ 0.5]]
```

In **Fig. 19**, we can confirm the mapping by simply visual inspection.

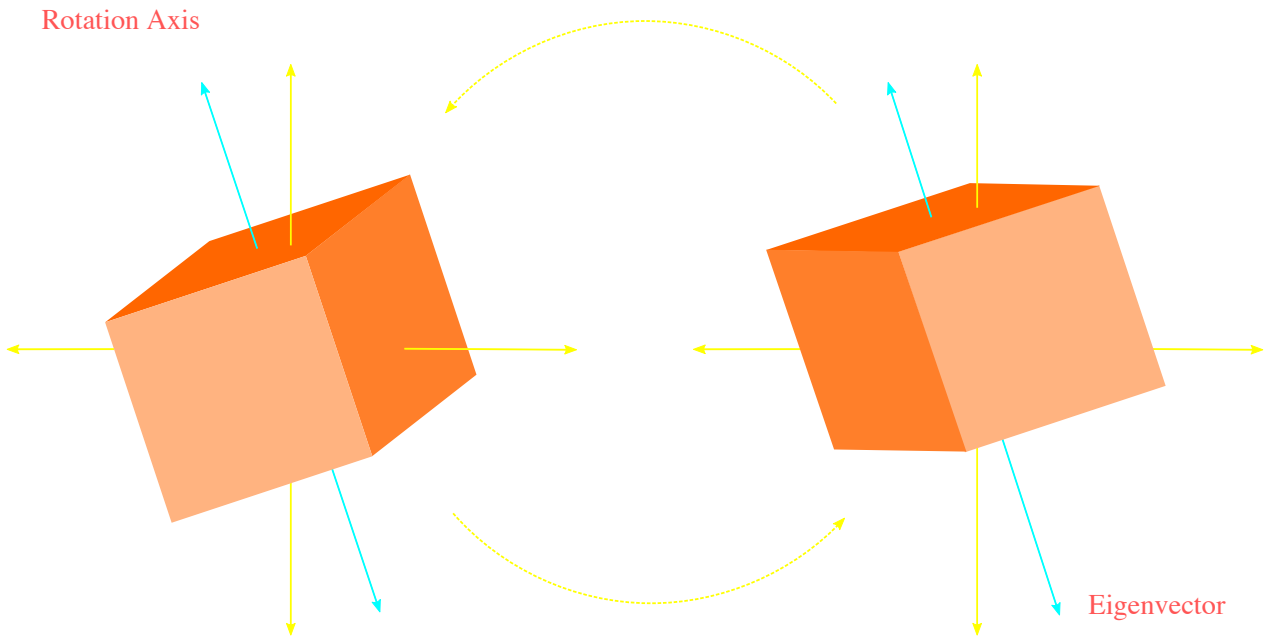
Eigenvectors, Eigenvalues and Eigenspaces

Eigen is a German word meaning "own" or "characteristic". Thus, roughly speaking, the eigenvector and eigenvalue of a matrix refer to their "characteristic vector" and "characteristic value" for that vector, respectively. As a cognitive scientist, I like to think in eigenvectors as the "pivotal" personality trait of someone, i.e., the personality "axis" around which everything else revolves, and the eigenvalue as the "intensity" of that trait.

Put simply, the **eigenvector of a matrix** is a non-zero vector that *only gets scaled* when multiplied by a transformation matrix A. In

other words, the vector does not rotate or change direction in any manner. It just gets larger or shorter. The **eigenvalue of a matrix** is the factor by which the eigenvector gets scaled. This is a bit of a stretch, but in terms of the personality analogy, we can think in the eigenvector as the personality trait that does not change even when an individual change of context: Lisa Simpson "pivotal" personality trait is *conscientiousness*, and no matter where she is, home, school, etc., their personality revolves around that. Following the analogy, the eigenvalue would represent the magnitude or intensity of such traits in Lisa. Fig 20. illustrate the geometrical representation of an eigenvector with a cube rotation.

Fig. 20: Eigenvector in a 3-dimensional rotation



More formally, we define eigenvectors and eigenvalues as:

$$A\mathbf{x} := \lambda\mathbf{x}$$

Where A is a square matrix in $R^{n \times n}$, $\mathbf{x}$ the eigenvector, and λ an scalar in R. This identity may look weird to you: **How do we go from matrix-vector multiplication to scalar-vector multiplication?** We are basically saying that somehow multiplying $\mathbf{x}$ by a matrix A or a scalar λ yields the same result. To make sense of this, recall our discussion about the effects of a matrix on a vector. Mappings or transformation like reflection and shear boils down to *a combination of scaling and rotation*. If a mapping A does not rotate

$\mathbf{x}$, it makes sense that such mapping can be reduced to a simpler scalar-vector multiplication $\lambda\mathbf{x}$.

If you recall our discussion about elementary matrices, you may see a simple way to make the $\mathrm{A}\mathbf{x} = \lambda\mathbf{x}$ more intuitive. Elementary matrices allow us to encode row and column operations on a matrix. Scalar multiplication, can be represented as by multiplying either the rows or columns of the identity matrix by the desired factor. For instance, for $\lambda = 2$:

$$2\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} = \begin{bmatrix}2 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 2\end{bmatrix}$$

This allow us to rewrite as $\lambda\mathrm{I}\mathbf{x}$, and to maintain the matrix-vector multiplication form as:

$$\mathrm{A}\mathbf{x} = \lambda\mathrm{I}\mathbf{x}$$

We can go further, and rearange our expression to:

$$\mathrm{A}\mathbf{x} - \lambda\mathrm{I}\mathbf{x} = 0$$

And to factor our $\mathbf{x}$ to get:

$$(\mathrm{A} - \lambda\mathrm{I})\mathbf{x} = 0$$

The first part of our new expression, $(\mathrm{A} - \lambda\mathrm{I})$, will yield a matrix, meaning that now we have matrix-vector multiplication. In particular, we want a non-zero vector $\mathbf{x}$ that when multiplied by $(\mathrm{A} - \lambda\mathrm{I})$ yields 0. The only way to achieve this is when the scaling factor associated with $(\mathrm{A} - \lambda\mathrm{I})$ is 0 as well. Here is when **determinants** come into play. Recall that the determinant of a matrix represents the scaling factor of such mapping, which in this specific case, happens to be the *eigenvalue* of the matrix. Consequently, we want:

$$\det(\mathrm{A} - \lambda\mathrm{I}) = 0$$

Since A and I are fixed, in practice, we want to find a value of λ that will yield a 0 determinant of the matrix. Any matrix with a determinant of 0 will be *singular*. This time, we want the matrix to be singular, as we are trying to solve a problem with three

unknowns and two equations, therefore, it is the only way to solve it.

By finding a value for λ that makes the determinat 0, we are effectively making the equality $(\mathrm{A} - \lambda \mathrm{I})\mathbf{x} = 0$ true.

Let's do an example to make these ideas more concrete. Consider the following matrix:

$$\begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$$

Let's first multiply $\mathrm{A} - \lambda \mathrm{I}$ to get a single matrix:

$$\begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} = \begin{bmatrix} 4 - \lambda & 2 \\ 1 & 3 - \lambda \end{bmatrix}$$

We begin by computing the determinant as:

$$\det(\mathrm{A}) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = (ad) - (bc)$$

Which yield the following polynomial:

$$\begin{vmatrix} 4 - \lambda & 2 \\ 1 & 3 - \lambda \end{vmatrix} = (4 - \lambda)(3 - \lambda) - 2 \times 1$$

That we solve as any other quadratic polynomial, which receives the special name of **characteristc polynomial**. When we equate the characteristic polynomial to 0, we call such expression the **characteristic equation**. The roots of the characteristic equation, are the eigenvalues of the matrix:

$$(4 - \lambda)(3 - \lambda) - 2 \times 1 = 12 - 4\lambda - 3\lambda + \lambda^2 - 3 = 10 - 7\lambda + \lambda^2$$

Wich can be factorized as:

$$(2 - \lambda)(5 - \lambda)$$

There you go: we obtain **eigenvalues $\lambda_1 = 2$, and $\lambda_2 = 5$.** this simply means that $\mathrm{A}\mathbf{x} = \lambda\mathbf{x}$ can be solved for eigenvalues equal to 2 and 5, assuming non-zero eigenvectors.

Once we find the eigenvalues, we can compute the eigenvector for each of them. Let's start with $\lambda_1 = 2$:

$$\begin{bmatrix} 4-\lambda & 2 \\ 1 & 3-\lambda \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4-2 & 2 \\ 1 & 3-2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0$$

Since the first and second column are identical, we obtain that the solution for the system is pair of such that $\mathbf{x}_1 = -\mathbf{x}_2$, for instance:

$$\mathrm{E}_{\lambda=2} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Such vector correspond to the **eigenspace** for the eigenvalue $\lambda = 2$. An eigenspace denotes all the vectors that correspond to a given eigenvalue, which in this case is the span of $\mathrm{E}_{\lambda=2}$.

Now let's evaluate for $\lambda = 5$:

$$\begin{bmatrix} 4-\lambda & 2 \\ 1 & 3-\lambda \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4-5 & 2 \\ 1 & 3-5 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 & 2 \\ 1 & -2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0$$

Since the first column is just -2 times the second, the solution for the system will be any pair such that $2\mathbf{x}_1 = \mathbf{x}_2$, i.e.:

$$\mathrm{E}_{\lambda=5}\begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

With the span of $\mathrm{E}_{\lambda=5}$ as the eigenspace for the eigenvalue $\lambda = 5$.

As usual, we can find the eigenvectors and eigenvalues of a matrix with NumPy. Let's check our computation:

```
A = np.array([[4, 2],
              [1, 3]])

values, vectors = np.linalg.eig(A)


print(f'Eigenvalues of A:\n{values}\n')
```

```
print(f'Eigenvectors of A:\n{np.round(vectors,3)}')
```

```
Eigenvalues of A:
[5. 2.]

Eigenvectors of A:
[[ 0.894 -0.707]
 [ 0.447  0.707]]
```

The eigenvalues are effectively 5 and 2. The eigenvectors (aside rounding error), match exactly what we found. For $\lambda = 5$, $2\mathbf{x}_1 = \mathbf{x}_1$, and for $\lambda = 2$ that $\mathbf{x}_1 = -\mathbf{x}_2$.

Not all matrices will have eigenvalues and eigenvectors in $\mathbb{R}$. Recall that we said that eigenvalues essentially indicate scaling, whereas eigenvectors indicate the vectors that remain unchanged under a linear mapping. It follows that if a linear transformation does not stretch vectors and rotates all of them, then no eigenvectors and eigenvalues should be found. An example of this is a rotation matrix:

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Let's compute its eigenvectors and eigenvalues in NumPy:

```
B = np.array([[0, -1],
              [1, 0]])

values, vectors = np.linalg.eig(B)

print(f'B Eigenvalues:\n{values}\n')
print(f'B Eigenvectors:\n{vectors}\n')
```

```
B Eigenvalues:
[0.+1.j 0.-1.j]

B Eigenvectors:
```

```
[[0.70710678+0.j         0.70710678-0.j        ]
 [0.        -0.70710678j 0.        +0.70710678j]]
```

The ***+0.j*** indicates the solution yield imaginary numbers, meaning that there are not eigenvectors or eigenvalues for the matrix $B \in R$

Trace and determinant with eigenvalues

The **trace** of a matrix is the *sum of its diagonal elements*. Formally, we define the trace for a square matrix $A \in R^{n \times n}$ as:

$$tr(A) := \sum_{i=1}^{n} = a_{ii}$$

There is something very special about eigenvalues: *its sum equals the trace of the matrix*. Recall the matrix A from the previous section:

$$\begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$$

Which has a trace equal to $4 + 3 = 7$. We found that their eigenvalues were $\lambda_1 = 2$ and $\lambda_2 = 5$, which also add up to 7.

Here is another curious fact about eigenvalues: *its product equals to the determinant of the matrix*. The determinant of A equals to $(4 \times 3) - (2 \times 1) = 10$. The product of the eigenvalues is also 10.

These two properties hold only when we have a full set of eigenvalues, this is when we have as many eigenvalues as dimensions in the matrix.

Eigendecomposition

In previous sections, we associated LU decomposition with Gaussian Elimination and QR decomposition with Gram-Schmidt Orthogonalization. Similarly, we can associate the Eigenvalue algorithm to find the eigenvalues and eigenvectors of a matrix, wit the **Eigendecomposition** or **Eigenvalue Decomposition**.

We learned that we can find the eigenvalues and eigenvectors of a

square matrix (assuming they exist) with:

$$(A - \lambda I)\mathbf{x} = 0$$

Process that entail to first solve the characteristic equation for the polynomial, and then evaluate each eigenvalue to find the corresponding eigenvector. The question now is how to express such process as a single matrix-matrix operation. Let's consider the following transformation matrix:

$$T = \begin{bmatrix} 5 & 3 & 0 \\ 2 & 6 & 0 \\ 4 & -2 & 2 \end{bmatrix}$$

Let's begin by computing the eigenvalues and eigenvectors with NumPy:

```
A = np.array([[5,  3, 0],
              [2,  6, 0],
              [4,  -2, 2]])


eigenvalues, eigenvectors = np.linalg.eig(A)

print(f'B Eigenvalues:\n{eigenvalues}\n')
print(f'B Eigenvectors:\n{eigenvectors}\n')


B Eigenvalues:
[2. 8. 3.]

B Eigenvectors:
[[ 0.          0.6882472   0.18291323]
 [ 0.          0.6882472  -0.12194215]
 [ 1.          0.22941573  0.97553722]]
```

We obtained a vector of eigenvalues and a Now, we know that the following identity must be true for scalar-matrix multiplication:

$$A\mathbf{x} = \lambda I\mathbf{x}$$

Since we want to multiply a matrix of eigenvalues by the matrix of

eigenvectors, we have to be careful about selecting the order of the multiplication. Recall that matrix-matrix multiplication *is not commutative*, meaning that the multiplication order matters. Before this wasn't a problem, because scalar-matrix multiplication is commutative. What we want, is in operation such that eigenvalues scale eigenvectors. For this, we will put the eigenvectors in a matrix X, the result of λI in a matrix Λ, and multiply X by Λ from the right side as:

$$A\mathbf{X} = \mathbf{X}\Lambda$$

Let's do this with NumPy

```
X = eigenvectors
I = np.identity(3)
L = I * eigenvalues
```

```
print(f'Left-side of the equation AX:\n{A @ X}\n')
print(f'Right-side of the equation XL:\n{X @ L}\n')
```

```
Left-side of the equation AX:
[[ 0.          5.50597761  0.54873968]
 [ 0.          5.50597761 -0.36582646]
 [ 2.          1.83532587  2.92661165]]

Right-side of the equation XL:
[[ 0.          5.50597761  0.54873968]
 [ 0.          5.50597761 -0.36582646]
 [ 2.          1.83532587  2.92661165]]
```

Verify equality

```
print(f'Entry-wise comparison: {np.allclose(A @ X, X @ L)}')
```

```
Entry-wise comparison: True
```

A a side note, it is not a good idea to compare NumPy arrays with the equality operator, as rounding error and the finite internal

bit representation may yield `False` when values are technically equal. For instance:

```
(A @ X == X @ L)
```

```
array([[ True,  True, False],
       [ True,  True, False],
       [ True,  True,  True]])
```

We still have one issue to address to complete the Eigendecomposition of A: to get rid of X on the left side of the equation. A first thought is simply to multiply by the X^{-1} to cancel X on both sides. This won't work because on the left side of the equation, X is multiplying from the right of A, whereas on the right side of the equation, X is multiplying from the left of Λ. Yet, we still can get take the inverse to eliminate only from the left side of the equation and obtain:

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

Lo and behold, **we have found the expression for the Eigendecomposition.**

Let's confirm this works:

```
X_inv = np.linalg.inv(X)

print(f'Original matrix A:\n{A}\n')
print(f'Reconstruction of A with Eigen Decomposition of A:\n{X @ L
```

```
Original matrix A:
[[ 5  3  0]
 [ 2  6  0]
 [ 4 -2  2]]

Reconstruction of A with Eigen Decomposition of A:
[[ 5.  3.  0.]
 [ 2.  6.  0.]
 [ 4. -2.  2.]]
```

Eigenbasis are a good basis

There are cases when a transformation or mapping T has associated a full set of eigenvectors, i.e., as many eigenvectors as dimensions in T. We call this set of eigenvectors an **eigenbasis**.

When approaching linear algebra problems, selecting a "good" basis for the matrix or vector space can significantly simplify computation, and also reveals several facts about the matrix that would be otherwise hard to see. Eigenbasis, are an example of a basis that would make our life easier in several situations.

From the previous section, we learned that the Eigenvalue Decomposition is defined as:

$$A := X\Lambda X^{-1}$$

Conceptually, a first lesson is that transformations, like A, have two main components: a matrix Λ that stretch, shrink, or flip, the vectors, and X, which represent the "axes" around which the transformation occurs.

Eigenbasis also make computing the power of a matrix easy. Consider the case of A^2:

$$A^2 = X\Lambda X^{-1} X\Lambda X^{-1}$$

Since $X^{-1}X$ equals the identity, we obtain:

$$A^2 = X\Lambda^2 X^{-1}$$

The pattern:

$$A^n = X\Lambda^n X^{-1}$$

Generalizes to any power. For powers of $n = 2$ or $n = 3$ such approach may not be the best, as computing the power directly on A may be easier. But, when dealing with large matrices with powers of thousands or millions, this approach is far superior. Further, it even works for the inverse:

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

We can see this is true by testing that AA^{-1} equals the identity:

$$AA^{-1} = X\Lambda X^{-1}X\Lambda^{-1}X^{-1}$$

Pay attention to what happens now: $X^{-1}X = I$, which yields:

$$AA^{-1} = X\Lambda I\Lambda^{-1}X^{-1} = X\Lambda\Lambda^{-1}X^{-1}$$

Now, $\Lambda\Lambda^{-1}$, also yields the identity:

$$AA^{-1} = XIX^{-1} = XX^{-1}$$

Finally, XX^{-1}, also yields the identity:

$$AA^{-1} = I$$

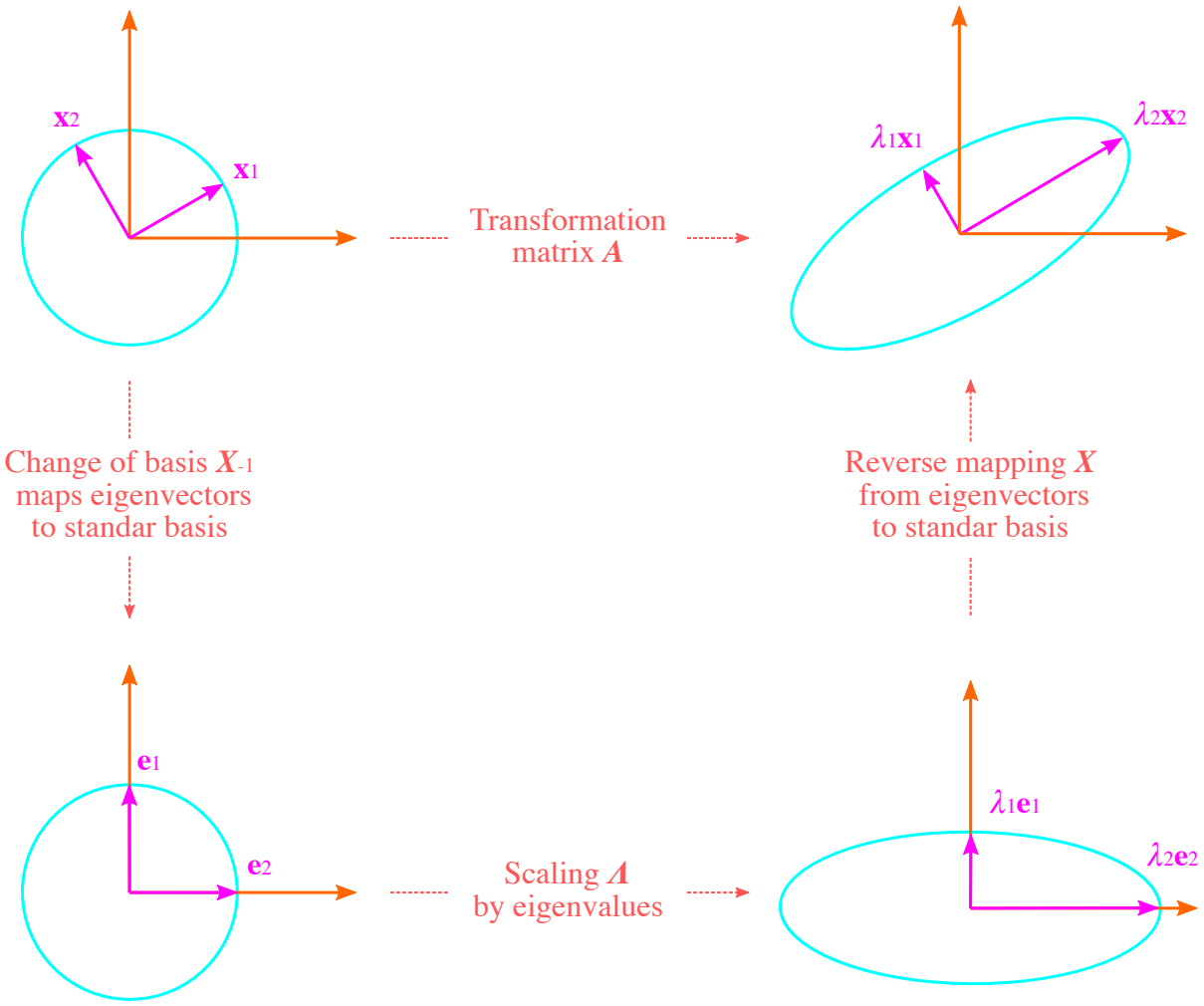
Geometric interpretation of Eigendecomposition

We said that Eigenbasis is a good basis as it allows us to perform computations more easily and to better understand the nature of linear mappings or transformations. The geometric interpretation of Eigendecomposition further reinforces that point. In concrete, the Eigendecomposition elements can be interpreted as follow:

1. X^{-1} change basis (rotation) from the standard basis into the eigenbasis
2. Λ scale (stretch, shrinks, or flip) the corresponding eigenvectors
3. X change of basis (rotation) from the eigenbasis basis onto the original standard basis orientation

Fig 21. illustrate the action of X^{-1}, Λ, and X in a pair of vectors in the standard basis.

Fig. 21: Eigendecomposition



The problem with Eigendecomposition

The problem is simple: **Eigendecomposition can only be performed on square matrices, and sometimes the decomposition does not even exist**. This is very limiting from an applied perspective, as most practical problems involve non-square matrices.

Ideally, we would like to have a more general decomposition, that allows for non-square matrices and that exist for all matrices. In the next section we introduce the **Singular Value Decomposition**, which takes care of these issues.

Singular Value Decomposition

Singular Value Decomposition (SVD) is one the most relevant decomposition in applied settings, as it goes beyond the limitations of Eigendecomposition. Specifically, SVD can be

performed for **non-squared matrices and singular matrices (i.e., matrices without a full set of eigenvectors)**. SVD can be used for the same applications that Eigendecomposition (e.g., low-rank approximations) plus the cases for which Eigendecomposition does not work.

Singular Value Decomposition Theorem

Since we reviewed Eigendecomposition already, understanding SVD becomes easier. The SVD theorem states that any rectangular matrix $A \in R^{m \times n}$ can be decomposed as the product of an orthogonal matrix $U \in R^{m \times m}$, a diagonal matrix $\Sigma \in R^{m \times m}$, and another orthogonal matrix $X^{-1} \in R^{n \times n}$:

$$A := U\Sigma X^{-1}$$

Another common notation is: $A := U\Sigma V^T$. Here I'm using X^{-1} just to denote that the right orthogonal matrix is the same as in the Eigenvalue decomposition. Also notice that the inverse of an square orthogonal matrix is $X^{-1} = X^T$.

The *Singular Values* are the non-negative values along the diagonal of Σ, which play the same role as eigenvalues in Eigendecomposition. You may even find some authors call them eigenvalues as well. Since Σ is a rectangular matrix of the shape as A, the diagonal of the matrix which contains the singular values will necessarily define a square submatrix within Σ. There are two situations to pay attention to: (1) when $m > n$, i.e., more rows than columns, and (2) when $m < n$, i.e., more columns than rows.

For the first case, $m > n$, we will have zero-padding at the bottom of Σ as:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_n \\ 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

For the second case, $m < n$, we will have zero-padding at the right of Σ as:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & \ddots & 0 & \vdots & \ddots & \vdots \\ 0 & 0 & \sigma_n & 0 & \cdots & 0 \end{bmatrix}$$

Take the case of $A^{3\times 2}$, the SVD is defined as:

$$A = U\Sigma V^T = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \begin{bmatrix} \sigma_{11} & 0 \\ 0 & \sigma_{22} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix}$$

Now let's evaluate the opposite case, $A^{2\times 3}$, the SVD is defined as:

$$A = U\Sigma V^T = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \begin{bmatrix} \sigma_{11} & 0 & 0 \\ 0 & \sigma_{22} & 0 \end{bmatrix} \begin{bmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \\ v_{31} & v_{32} & v_{33} \end{bmatrix}$$

Singular Value Decomposition computation

SVD computation leads to messy calculations in most cases, so this time I'll just use NumPy. We will compute three cases: a wide matrix $A^{2\times 3}$, a tall matrix $A^{3\times 2}$, and a square matrix $A^{3\times 3}$ with a pair of linearly dependent vectors (i.e., a "defective" matrix, or singular, or not full rank, etc.).

```
# 2 x 3 matrix
A_wide = np.array([[2, 1, 0],
                   [-3, 0, 1]])

# 3 x 2 matrix
A_tall = np.array([[2, 1],
                   [-3, 0],
                   [0, 2]])
```

```
# 3 x 3 matrix: col 3 equals 2 x col 1
A_square = np.array([[2, 1, 4],
                     [-3, 0, -6],
                     [1, 2, 2]])


U1, S1, V_T1 = np.linalg.svd(A_wide)
U2, S2, V_T2 = np.linalg.svd(A_tall)
U3, S3, V_T3 = np.linalg.svd(A_square)



print(f'Left orthogonal matrix wide A:\n{np.round(U1, 2)}\n')
print(f'Singular values diagonal matrix wide A:\n{np.round(S1, 2)}
print(f'Right orthogonal matrix wide A:\n{np.round(V_T1, 2)}')


Left orthogonal matrix wide A:
[[-0.55  0.83]
 [ 0.83  0.55]]

Singular values diagonal matrix wide A:
[3.74 1.  ]

Right orthogonal matrix wide A:
[[-0.96 -0.15  0.22]
 [-0.    0.83  0.55]
 [ 0.27 -0.53  0.8 ]]
```

As expected, we obtain a $n \times n$ orthogonal matrix on the left, and a $m \times m$ orthogonal matrix on the right. NumPy only returns the singular values along the diagonal instead of the 2×3 matrix, yet it makes no difference regarding the values of the SVD.

```
print(f'Left orthogonal matrix for tall A:\n{np.round(U2, 2)}\n')
print(f'Singular values diagonal matrix for tall A:\n{np.round(S2,
print(f'Right orthogonal matrix for tall A:\n{np.round(V_T2, 2)}')


Left orthogonal matrix for tall A:
[[-0.59 -0.24 -0.77]
```

```
 [ 0.8  -0.32 -0.51]
 [-0.13 -0.91  0.38]]

Singular values diagonal matrix for tall A:
[3.67 2.13]

Right orthogonal matrix for tall A:
[[-0.97 -0.23]
 [ 0.23 -0.97]]
```

As expected, we obtain a $m \times m$ orthogonal matrix on the left and a $n \times n$ orthogonal matrix on the right. Notice that NumPy returns the singular values in descending order of magnitude. This is a convention you'll find in the literature frequently.

```
print(f'Left orthogonal matrix for square A:\n{np.round(U3, 2)}\n'
print(f'Singular values diagonal matrix for square A:\n{np.round(S
print(f'Right orthogonal matrix for square A:\n{np.round(V_T3, 2)}
```

```
Left orthogonal matrix for square A:
[[-0.54 -0.2  -0.82]
 [ 0.79 -0.46 -0.41]
 [-0.29 -0.86  0.41]]

Singular values diagonal matrix for square A:
[8.44 1.95 0.  ]

Right orthogonal matrix for square A:
[[-0.44 -0.13 -0.89]
 [ 0.06 -0.99  0.12]
 [ 0.89  0.   -0.45]]
```

Although column three is just two times column one (i.e., linearly dependent), we obtain the SVD for A. Notice that the third singular value equals 0, which is a reflection of the fact that the third column just contains redundant information.

Geometric interpretation of the Singular Value Decomposition

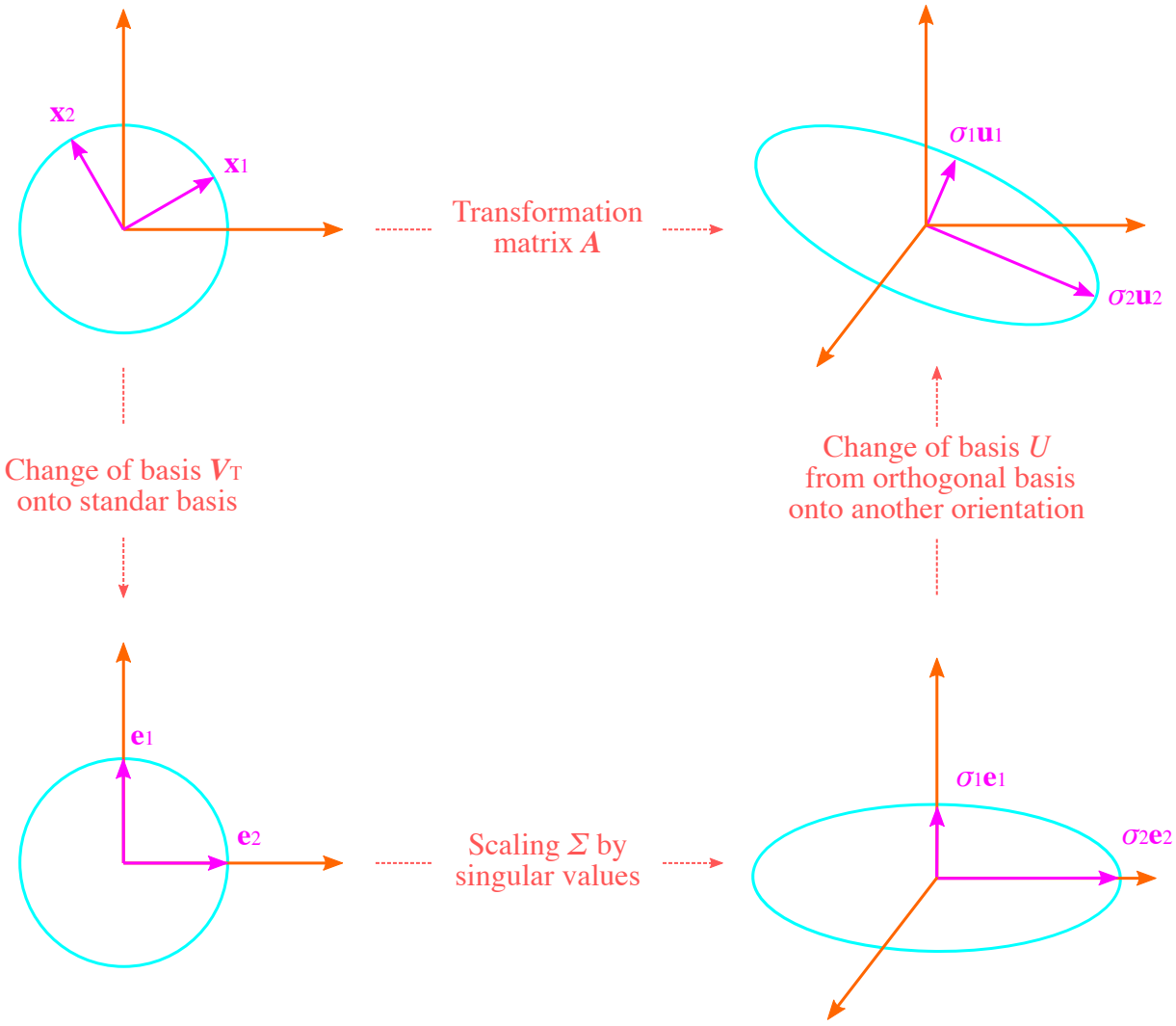
As with Eigendecomposition, SVD has a nice geometric interpretation as a sequence of linear mappings or transformations. Concretely:

1. V^T change basis (rotation) from the standard basis into a set of orthogonal basis
2. Σ scale (stretch, shrinks, or flip) the corresponding orthogonal basis
3. U change of basis (rotation) from the new orthogonal basis onto some other orientation, i.e., not necessarily where we started.

The key difference with Eigendecomposition is in U: instead of going back to the standard basis, U performs a change of basis onto another direction.

Fig 22. illustrate the effect of $\mathrm{A}^{3\times2}$, i.e., V^T, Σ, and U, in a pair of vectors in the standard basis. The fact that the right orthogonal matrix has 3 column vectors generates the third dimension which is orthogonal to the ellipse surface.

Fig. 22: Singular Value Decomposition



Singular Value Decomposition vs Eigendecomposition

The SVD and Eigendecomposition are very similar, so it's easy to get confused about how they differ. Here is a list of the most important ways on which both are different:

1. The SVD decomposition exist for any rectangular matrix $\in \mathrm{R}^{m \times n}$, while the Eigendecomposition exist only for square matrices $\in \mathrm{R}^{n \times n}$.
2. The SVD decomposition exists even if the matrix A is defective, singular, or not full rank, whereas the Eigendecomposition does not have a solution in R in such a case.
3. Eigenvectors X are orthogonal only for *symmetric matrices*, whereas the vectors in the U and V are orthonormal. Hence, X represents a rotation only for symmetric matrices, whereas U and V are always rotations.
4. In the Eigendecomposition, X and X^T are the inverse fo each

other, whereas U and V in the SVD are not.
5. The singular values in Σ are always real and positive, which is not necessarily the case for Λ in the Eigendecomposition.
6. The SVD change basis in both the domain and codomain. The Eigendecomposition change basis in the same vector space.
7. For symmetric matrices, $A \in R^{n \times n}$, the SVD and Eigendecomposition yield the same results.

Matrix Approximation

In machine learning applications, it is common to find matrices with thousands, hundreds of thousands, and even millions of rows and columns. Although the Eigendecomposition and Singular Value Decomposition make matrix factorization efficient to compute, such large matrices can consume an enormous amount of time and computational resources. One common way to "get around" these issues is to utilize **low-rank approximations** of the original matrices. By low-rank we mean utilizing a subset of orthogonal vectors instead of the full set of orthogonal vectors, such that we can obtain a "reasonably" good approximation of the original matrix.

There are many well-known and widely use low-approximation procedures in machine learning, like Principal Component Analysis, Factor Analysis, and Latent Semantic analysis, and dimensionality reduction techniques more generally. Low-rank approximations are possible because in most instances, a small subset of vectors contains most of the information in the matrix, which is a way to say the most data points can be computed as linear combinations of a subset of orthogonal vectors.

Best rank-k approximation with SVD

So far we have represented the SVD as the product of three matrices, U, Σ, and V^T. We can represent this same computation as a the sum of the matching columns of each of these components as:

$$A := \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{u}_i^T$$

Notice that each iteration of $\sum_{i=1}^{r} \mathbf{u}_i \mathbf{u}_i^T$ will generate a matrix $\sigma_i A_i$,

which then can be multiplied by σ_i. In other words, the above expression also equals:

$$\sum_{i=1}^{r} \sigma_i \mathrm{A}_i$$

In matrix notation, we can express the same idea as:

$$\mathrm{A}_k = \mathrm{U}_k \Sigma_k \mathrm{V}_k^T$$

Now, we can approximate A by taking the sum over k values instead of r values. For instance, for a square matrix with $r = 100$ orthogonal vectors, we can compute an approximation with the $k = 5$ orthogonal vectors as:

$$\hat{\mathrm{A}} := \sum_{i=1}^{k=5} \sigma_i \mathbf{u}_i \mathbf{u}_i^T = \sum_{i=1}^{k} \sigma_i \mathrm{A}_i$$

In practice, this means that we take $k = 5$ orthogonal vectors from U and V^T, times 5 singular values, which requires considerably less computation and memory than the 100×100 matrix. We call this the **best low-rank approximation** simply because it takes the 5 largest singular values, which account for most of the information. Nonetheless, we still a precise way to estimate how good is our estimation, for which we need to compute the norm for $\hat{\mathrm{A}}$ and A, and how they differ.

Best low-rank approximation as a minimization problem

In the previous section, we mentioned we need to compute some norm for $\hat{\mathrm{A}}$ and A, and then compare. This can be conceptualized as a error minimization problem, where we search for the smallest distance between A and the low-rank approximation $\hat{\mathrm{A}}$. For instance, we can use the Frobenius and compute the distance between $\hat{\mathrm{A}}$ and A as:

$$\|\mathrm{A} - \hat{\mathrm{A}}\|_F$$

Alternatively, we can compute the explained variance for the decomposition, where the highest the variance the better the approximation, ranging from 0 to 1. We can perform the SVD

approximation with NumPy and sklearn as:

```
from sklearn.decomposition import TruncatedSVD

A = np.random.rand(100,100)


SVD1 = TruncatedSVD(n_components=1, n_iter=7, random_state=1)
SVD5 = TruncatedSVD(n_components=5, n_iter=7, random_state=1)
SVD10 = TruncatedSVD(n_components=10, n_iter=7, random_state=1)

SVD1.fit(A)
SVD5.fit(A)
SVD10.fit(A)


TruncatedSVD(algorithm='randomized', n_components=10, n_iter=7, ra
             tol=0.0)


print('Explained variance by component:\n')
print(f'SVD approximation with 1 component:\n{np.round(SVD1.explai
print(f'SVD approximation with 5 components:\n{np.round(SVD5.expla
print(f'SVD approximation with 10 component:\n{np.round(SVD10.expl


Explained variance by component:

SVD approximation with 1 component:
[0.01]

SVD approximation with 5 components:
[0.01 0.04 0.03 0.03 0.03]

SVD approximation with 10 component:
[0.01 0.04 0.03 0.03 0.03 0.03 0.03 0.03 0.03 0.03]
```

```
print('Singular values for each approximation:\n')
print(f'SVD approximation with 1 component:\n{np.round(SVD1.singul
print(f'SVD approximation with 5 components:\n{np.round(SVD5.singu
print(f'SVD approximation with 10 component:\n{np.round(SVD10.sing
```

```
Singular values for each approximation:

SVD approximation with 1 component:
[50.42]

SVD approximation with 5 components:
[50.42  5.47  5.26  5.16  5.08]

SVD approximation with 10 component:
[50.42  5.47  5.26  5.16  5.08  5.06  4.99  4.88  4.72  4.63]
```

```
print('Total explained variance by each approximation:\n')
print('Singular values for each approximation:\n')
print(f'SVD approximation with 1 component:\n{np.round(SVD1.explai
print(f'SVD approximation with 5 components:\n{np.round(SVD5.expla
print(f'SVD approximation with 10 component:\n{np.round(SVD10.expl
```

```
Total explained variance by each approximation:

Singular values for each approximation:

SVD approximation with 1 component:
0.01

SVD approximation with 5 components:
0.15

SVD approximation with 10 component:
0.29
```

As expected, the more components (i.e., the highest the rank of the approximation), the highest the explained variance.

We can compute and compare the norms by first capturing each matrix of the SVD as recovering $\hat{A}$, then compute the Frobenius norm of the difference between A and $\hat{A}$.

```
from sklearn.utils.extmath import randomized_svd

U, S, V_T = randomized_svd(A, n_components=5, n_iter=10, random_st
A5 = (U * S) @ V_T


print(f"Norm of the difference between A and rank 5 approximation:
```

```
Norm of the difference between A and rank 5 approximation:
26.32
```

This number is not very informative in itself, so we usually utilize the explained variance as an indication of how good is the low-rank approximation.

Epilogue

Linear algebra is an enormous and fascinating subject. These notes are just an introduction to the subject with machine learning in mind. I am no mathematician, and I have no formal mathematical training, yet, I greatly enjoyed writing this document. I have learned quite a lot by doing it and I hope it may help others that, like me, embark on the journey of acquiring a new skill by themselves, even when such effort may seem crazy to others.