LINEAR ALGEBRA

Draft Version

CONTENTS

Preface

"The purpose of computation is insight, not numbers"

Richard Hamming

When we communicate, we follow the structure of language: With some set of alphabets constructing words that have meanings, we deliver information by following a promised form of syntax. Likewise, when we "computationally" communicate, as in terms of both maths and computer, vectors and matrices can be considered as 'words'. If numbers can be compared as alphabets, certain combination of numbers in vectors and matrices can be compared as words - that is, as an object that has some meaning.

This note is the first series of *Social Data Science (SDS) 101: Computational Foundation*. There are in total of four series for SDS101: (i) Linear Algebra I : Introduction, (ii) Linear Algebra II : Application, (iii) Calculus, and (iv) Statistics. As one can notice, Linear Algebra is divided into two parts. Not because it covers more subjects than Calculus or Statistics, but because of its paramount importance in data science. Being the *language* we communicate with the computers, many of the problems that we are interested in solving are often represented through vectors or matrices.

For example, different *features* (or the *data*) of a sample are often represented as vectors.

Features	$sample_i$
X_1	$x_{1,i}$
X_2	$x_{2,i}$
⋮	⋮
X_N	$x_{N,i}$
Y	y_i

$$\begin{bmatrix} v_{1,1} \\ v_{2,1} \\ \vdots \\ v_{N,1} \end{bmatrix}$$

Vector

And a table of features for different samples is often represented as a matrix.

		Samples			
		$sample_1$	$sample_2$	...	$sample_M$
Features	X_1	$x_{1,1}$	$x_{1,2}$	...	$x_{1,M}$
	X_2	$x_{2,1}$	$x_{2,2}$	...	$x_{2,M}$
	⋮	⋮	⋮	⋱	⋮
	X_N	$x_{N,1}$	$x_{N,2}$	...	$x_{N,M}$
	Y	y_1	y_2	...	y_N

$$\begin{bmatrix} X_{1,1} & X_{1,2} & \cdots & X_{1,M} \\ X_{2,1} & X_{2,2} & \cdots & X_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ X_{N,1} & X_{N,2} & \cdots & X_{N,M} \end{bmatrix}$$

Matrix

Taking vectors or matrices as inputs, such data is then analyzed through different methods hired across different fields of mathematics, namely Calculus (i.e. Optimization) and Statistics (i.e. Regression); And the methods are communicated under the language of Linear Algebra. All in all, Linear Algebra is the backbone of Data Science.

This note has been developed with two main purposes: (i) To intuitively *explain* Linear Algebra, and (ii) To seamlessly *apply* such understanding in Python. As Strang (*2019, ix*) emphaiszed in *Introduction to Linear Algbrea*, a key goal in understanding such subject is learning how to "read" a matrix. It is far more important to see the meaning in the numbers, both geometrically and numerically, than to be familiar with the formal definitions. Thus, the majority of mathematical explanations would be less about formal proofs. Rather, it would be somewhere in between the boundaries of informal explanations and formal definitions.

At the end of each sub-chapters, a Python code is demonstrated to implement the topics learnt in that sub-chapter. No prior knowledge in Python is required, as it would be covered throughout the chapters.

There is one syntax you need to be familiar with, prior to reading the demonstrated Python codes. That syntax is called FOR LOOP. FOR LOOP consists of HEADER and BODY. HEADER, which often follows "`for i in range (1, n):`", decalres a range specifying the number of executions for the BODY. The two numbers in the parentheses are each lower and upper boundary, and the variable **`i`** is an index (or loop counter) which is counted per end of iterations. Once such index is counted as n - that is, after n-1th BODY is executed - the FOR LOOP is exited. Mathematically, such HEADER is equivalent to "`for all 1≤i<n`". BODY, which often updates some variable, is a sequence of statements which is exectued once per iteration.

Since FOR LOOPS will be regularly practiced throughout the chapters, three examples will be covered to foster the understanding of such syntax: (i) Sigma Summation (**Σ**), (ii) Pi Multiplication (**∏**), and (ii) Double-Sigma Summations (**ΣΣ**).

Starting with Sigma, such formula consists of three components: Index of Summation, Upper-limit of Summation, and Summands.

$$\sum_{i=0}^{n} x_i = x_0 + x_1 + \ldots + x_n$$

For the **Σ** above, `i` is the index, `n` is the upper-limit, and x_i is the summand. Such formula is read as "sum of the values of x_i, for `i = 0, 1, ..., n`" (or as "sum of the values of x_i, for `1≤i<n+1`"). With FOR LOOP, it is possible to translate the formula with Python. For simplicity, suppose `n=10` and $x_0, x_1, \ldots, x_9 = 1, 2, \ldots, 10$.

```
summands = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sigma = 0

for i in range (0, 10):
      sigma = sigma + summands[i]
```

Suppose x_0, x_1, ..., x_9 are saved in an ordered list named "summands" (ordered, such that x_0 is saved at the 0th sequence of the list, x_1 at 1st, x_2 at 2nd, and so on). And suppose a variable called sigma is created, which is a variable for iterative addition of summands.

Under the language of Mathematics, **Σ**, given index (and lower limit), upper limit, and summand, produces a summation. Under the language of Python, the particular FOR LOOP above, given HEADER (which incl. index, lower boundary, upper boundary) and BODY (statement), states identical operation as the SIGMA.

During its 1st iteration (i=0), x_0 is added on top of its original value 0. New value x_0 has been saved to variable sigma. During its 2nd iteration (i=1), x_1 is added on top of the updated sigma. New value x_0+x_1 has been updated to variable sigma. As one may have noticed, for every ith iteration, the BODY updates variable sigma, by adding x[i] to the previous value of sigma, which is $x_0+x_1+...+x_{i-1}$. FOR LOOP iterates until i=9, and once i=10, the HEADER "scans" that the index has exceeded the upper limit and terminates the LOOP. Note that, while **Σ** includes n in range ($0 \leq i \leq n$), FOR LOOP includes upto n−1 in range ($0 \leq i < n$).

For every iterations, the BODY updates variable sigma by adding x[i] on top of its previous summations (sigma = sigma + x[i]). In Python, there exists an equivalent shortcut for such update, called a compound operator.

```
for i in range (1, 10):
      sigma += summands[i]
```

Both commands, sigma = sigma + x[i] and sigma += x[i], performs the exact same operation. For all simple arithmetic operations (+,−,×,÷), compound operators exist. In Python, the commands for the four arithmetic operations are +, −, *, / and for the compound operators, +=, −=, *=, /=.

Now, let us consider the remaining two examples, Pi (**∏**) and double-Sigma (**ΣΣ**).

```
multiplicands = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

pi = 1

for i in range (0, 10):
     pi *= multiplicands[i]
```

Once you understand **Σ**, the Python implementation of **∏** is quite straight forward: Given HEADER and BODY, for every i^{th} iteration, such FOR LOOP multiplies `multiplicands[i]` to its previous Pi product. On the otherhand, **ΣΣ** is "sum of a sum". Mathematically:

$$\sum_{i=0}^{n}\sum_{j=0}^{m} x_i y_j = x_0(y_0+y_1+...+y_n) + x_1(y_0+y_1+...+y_n) + ... + x_n(y_0+y_1+...+y_n)$$

Since **ΣΣ** is sum of a sum, it involves double FOR LOOPs - an OUTER LOOP in range $0 \le i < n$, and an INNER LOOP in range $0 \le j < m$. During the 1^{st} iteration of OUTER LOOP ($i=0$), INNER LOOP iterates the whole range. During the 2^{nd} iteration of OUTER LOOP ($i=1$), INNER LOOP again iterates the whole range. OUTER LOOP terminates when the index `i` exceeds `n`.

Suppose n=5 and m=10. And suppose $x_0, x_1, ..., x_4 = 1, 2, ..., 5$ and $y_0, y_1, ..., y_9 = 1, 2, ..., 10$. The Python implementation of a double-Sigma would be as follow:

```
x = [1, 2, 3, 4, 5]
y = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

double_sigma = 0

for i in range (0, 5):
     for j in range (0, 10):
          double_sigma += x[i]*y[j]
```

For a more detailed discussion of FOR LOOP in general, visit *Python: For Loops* by W3SCHOOLS (https://www.w3schools.com/python/python_for_loops.asp).

The resources mentioned above are ...

The Essence of Linear Algebra (3Blue1Brown, 2016)
https://youtu.be/fNk_zzaMoSs

Introduction to Linear Algebra (Strang, 2019)
https://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/

Vision of Linear Algebra (Strang, 2020)
https://youtu.be/YrHlHbtiSM0

Mathematics for Machine Learning (Deisenroth et al.)
https://mml-book.github.io/book/mml-book.pdf

Linear Algebra Done Right (Axler)
https://linear.axler.net/LADRvideos.html

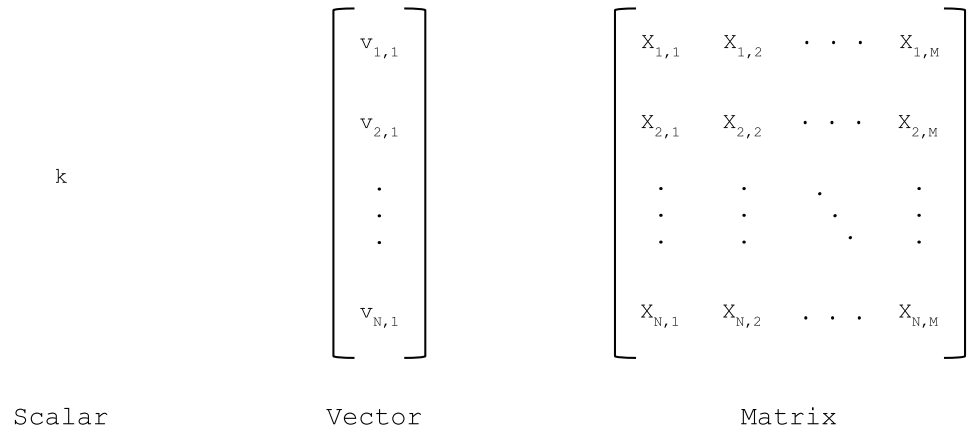
Python Numpy Tutorial (Standford CS231n)
https://cs231n.github.io/python-numpy-tutorial/#numpy

The resources mentioned above are ... Nonetheless, they are all free-to-access. Yet, if one is seeking for a deeper understanding or formal proofs, definitions, and languages, Gilbert Strang's *Introduction to Linear Algebra* and *Linear Algebra and Learning from Data*, and Sheldon Axler's *Linear Algebra Done Right* are the classics.

I. Vector

Before discussing vectors, it is necessary to have a brief understanding of what $\mathbf{R}$ is. For now, suppose $\mathbf{R}^N$ as *N-dimensional space* (the formal defintion of such "space", will be discussed in the last chapter, ***III. Space***). As an example, $\mathbf{R}^2$ would mean a 2-dimensional space constructed by two "axes" of real numbers (or the XY plane). And $\mathbf{R}^3$ would mean a 3-dimensional space described by three "axes" of real numbers (or the XYZ space).

Linear Algebra is a subject around three objects: *Scalar*, *Vector* and *Matrix*. This chapter is about the first two, Scalar and Vector. *Scalar* ($k \in \mathbf{R}^{1\times1}$) is an object of single entry, or a constant. *Vector* ($\boldsymbol{v} \in \mathbf{R}^{N\times1}$) is a N-tuple of entries, or a collection of N elements. The vector in the top-right diagram refers to a column vector, a single column with N rows (***column vector*** $\in \mathbf{R}^{N\times1}$). Contrariwise, row vector is a single row with M columns (***row vector*** $\in \mathbf{R}^{1\times M}$). In most cases, vector refers to the column vector.

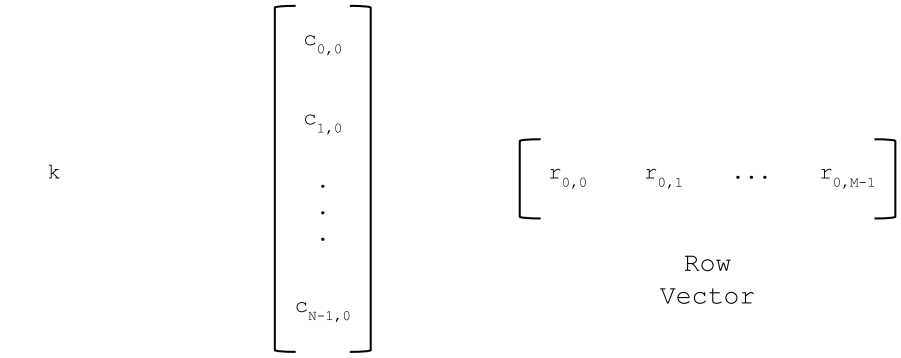


In Python, most, if not all, operations of Linear Algebra demands a package called **numpy**[1]. **Numpy** is an essential package to operate calculations of multidimensional array objects, such as vectors and matrices. Note that the index of an array object (incl. **numpy** array) always starts from `0`, instead of `1`. Hence, if a vector has a length of `N`, the last entry of such vector will have an index of `N-1`.

In **numpy**, a vector is defined by combining a outer square bracket with inner square brackets: Inner brackets represent the *row* and the sequence within the inner bracket represents the *column*. For example, a column vector with N rows will have N inner brackets, each with a single entry. And a row vector with N columns will have a single inner bracket with N entries.

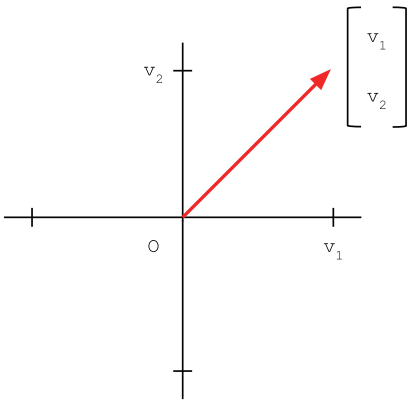
```
import numpy as np

k = integer

c = np.array([[c_0,0],
              [c_1,0],
              ... ,
              [c_N-1,0]])

r = np.array([[r_0,0, r_0,1, ... , r_0,M-1]])
```

[1] For a more detailed discussion about **numpy**, visit *Numpy Quickstart Tutorial* by Numpy.org (https://numpy.org/devdocs/user/quickstart.html) or *Python Numpy Tutorial* by Standford CS231n (https://cs231n.github.io/python-numpy-tutorial/#numpy).

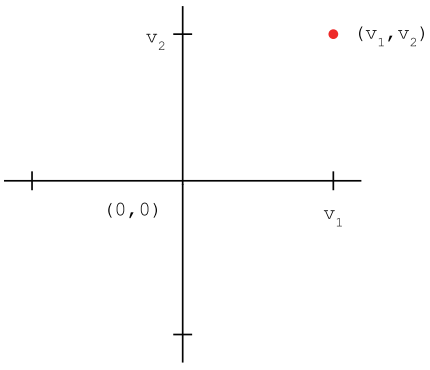


In Data Science, column vectors are often referred as *feature vectors*: A vector that represents N features of a sample. On the otherhand, row vectors are conventionally represented as $\boldsymbol{r}^T$ (superscript T is called *transpose*, which will be discussed later in this chapter).

But what exactly is a vector? We can describe a vector in two ways: (i) Numerically, either as a linear map or in parentheses, and (ii) Geometrically, either as an arrow or a point.



Numerically: Linear Map
Geometrically: "Arrow"



Numerically: Parentheses
Geometrically: "Point"

Visual animation is always helpful when learning math. Below is first episode of the series *Essence of Linear Algebra* by Grant Sanderson. Different episodes from such series are employed here and there throughout this note, to supplement the visually orientated explorations.

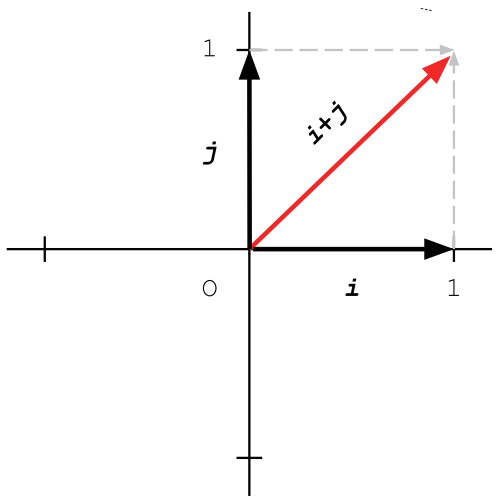
As suggested in the video, the essence of Linear Algebra is the ability to translate back-and-forth between the geometric interpretation and the numerical representation. That is, we are able to numerically translate the n-dimensional space through linear maps, where such space can be adequately conceptualized by *adding vectors* and *multiplying scalars.*

1.1 Vector Operations

1.1.1 Vector Addition and Scalar Multiplication

Vector Addition

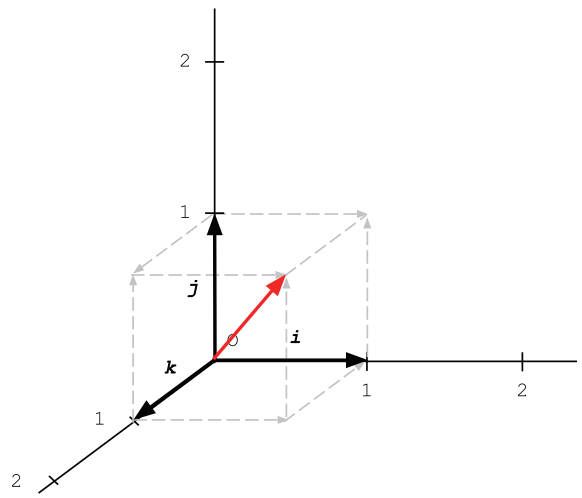
Algebraically, *Vector Addition* for two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ ($\boldsymbol{v}, \boldsymbol{w} \in \mathbf{R}^{N\times 1}$) is an element-wise addition of corresponding coordinates. Simply put, it is an addition of the entries at the same row (or "position"). Geometrically, vector addition can be visualized by using arrows, by putting the "tail" of $\boldsymbol{w}$ at the "head" of $\boldsymbol{v}$. That is, we initially travel along $\boldsymbol{v}$, and then along $\boldsymbol{w}$. Below represents vector adddition of two vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ in $\mathbf{R}^2$ ($\boldsymbol{i}, \boldsymbol{j} \in \mathbf{R}^{2\times 1}$).



$$\boldsymbol{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \boldsymbol{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\boldsymbol{i} + \boldsymbol{j} = \begin{bmatrix} 1 + 0 \\ 0 + 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

As shown geometrically, $\boldsymbol{i} + \boldsymbol{j}$ would give the same answer as $\boldsymbol{j} + \boldsymbol{i}$. Regardless of where you start from, addition of two vectors points at the same coordinate (1,1). More precisely, both *describes* the same vector $\boldsymbol{i} + \boldsymbol{j}$ (or $\boldsymbol{j} + \boldsymbol{i}$). The idea is consistent with $\mathbf{R}^3$.



$$\boldsymbol{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \boldsymbol{j} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \boldsymbol{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\boldsymbol{i} + \boldsymbol{j} + \boldsymbol{k} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

The idea of "connecting head to tail" equally applies when adding three vectors $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{k}$ in $\mathbf{R}^3$. No matter where you start from, addition of three vectors will end up describing the same.

While such an idea will be geometrically consistent for *N*-dimensional space, it is difficult to visualize beyond $\mathbf{R}^3$. However, aforementioned, the essence of Linear Algebra is to translate back-and-forth between the geometric interpretation and the numerical representation; It is possible to represent vector addition numerically, regardless of how large the *N* is.

$$\boldsymbol{v} + \boldsymbol{w} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} v_1 + w_1 \\ v_2 + w_2 \\ \vdots \\ v_N + w_N \end{bmatrix}$$

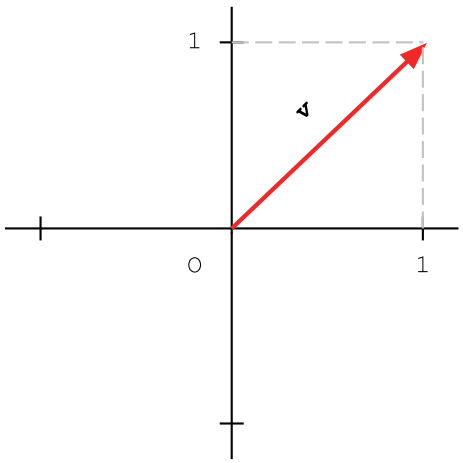
Scalar Multiplication

Scalar Multiplication is an element-wise multiplication of a scalar k ($\in \mathbf{R}$) on each coordinates of the vector $\boldsymbol{v}$ ($\in \mathbf{R}^{N\times 1}$).

Numerically,

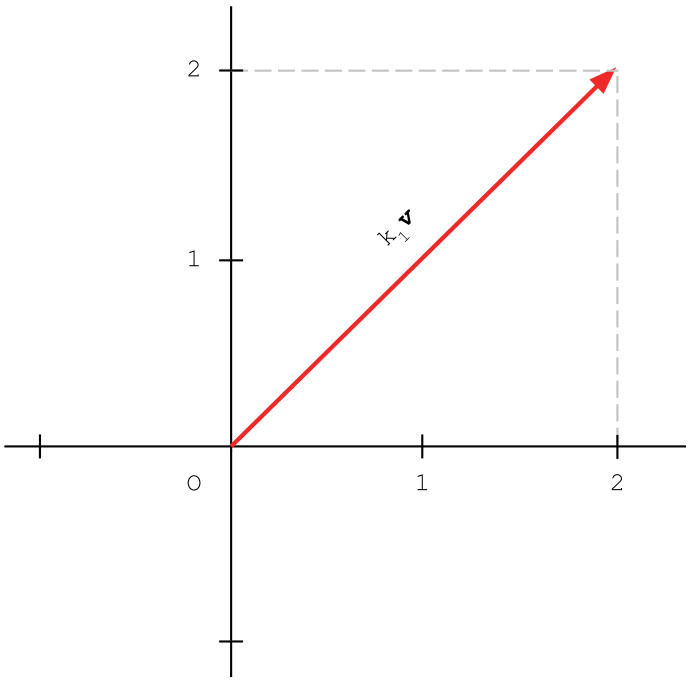
$$k\boldsymbol{v} = k \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix} = \begin{bmatrix} kv_1 \\ kv_2 \\ \vdots \\ kv_N \end{bmatrix}$$

Geometrically, scalar multiplication *scales* the "length" of a vector, by either stretching or reducing. Suppose there exists vector $\boldsymbol{v}$ ($\in \mathbf{R}^{2\times 1}$) with coordinates (1,1), scalars $k_1 = 2$ and $k_2 = 1/2$.



$$\boldsymbol{v} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Multiplying vector $\boldsymbol{v}$ by k_1 (= 2) would stretch such vector by twice of its orginal length.

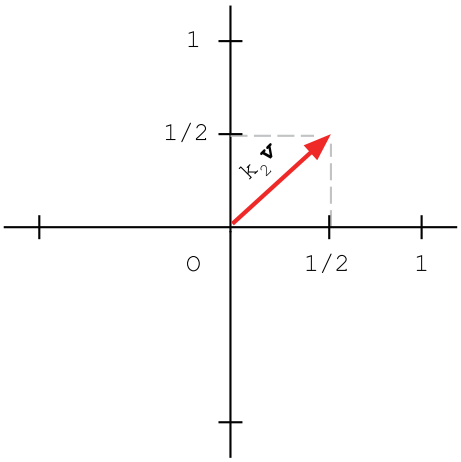


$$k_1 = 2$$

$$k_1\boldsymbol{v} = 2\begin{bmatrix}1\\1\end{bmatrix}$$

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

On the otherhand, multiplying vector $\boldsymbol{v}$ by k_2 (= 1/2) would reduce such vector by half of its orginal length.

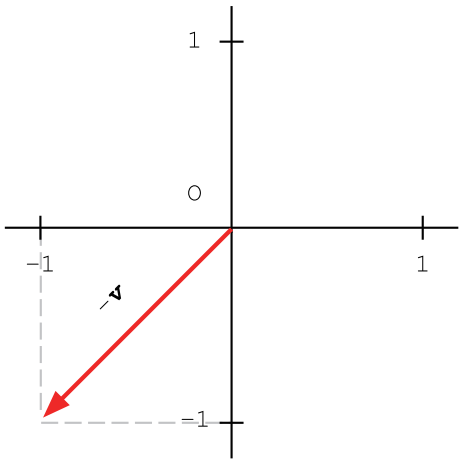


$$k_2 = \frac{1}{2}$$

$$k_2\boldsymbol{v} = \frac{1}{2}\begin{bmatrix}1\\1\end{bmatrix}$$

$$= \begin{bmatrix}\frac{1}{2}\\\frac{1}{2}\end{bmatrix}$$

When the scalar is negative, multiplying such scalar to a vector results an vector with an opposite "direction" to the original.

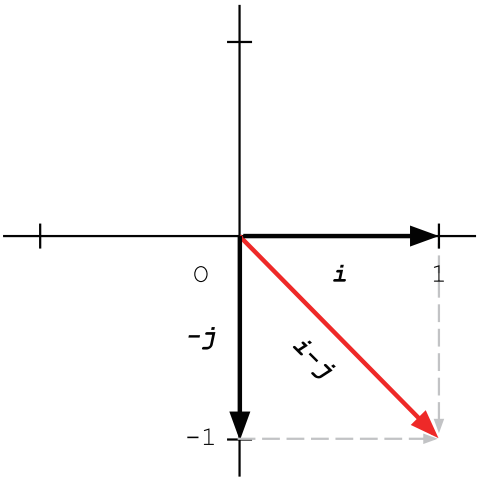


$$k = -1$$

$$k\boldsymbol{v} = -\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

Subtracting vector ***w*** from vector ***v*** (***v*** - ***w***) is equivalent to multuplying a negative to vector w, then adding ***v*** (***v*** + (- ***w***)). It is a vector addition.



$$\boldsymbol{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad -\boldsymbol{j} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$\boldsymbol{i} - \boldsymbol{j} = \begin{bmatrix} 1 - 0 \\ 0 - 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

While mathematically, it is unable to add a scalar to a vector, in Python, it is possible. The method that allows an element-wise addition (or subtraction) of a scalar to a vector is called *broadcasting*.

Broadcasting automatically converts a scalar ($k \in \mathbf{R}$) into a vector ($\boldsymbol{k} \in \mathbf{R}^{N \times 1}$), where such dimension N is determined by the dimension of the vector that is added to.

```
k = integer

v = np.array([[c_{0,0}],
              [c_{1,0}],
              ... ,
              [c_{N-1,0}]])

w = np.array([[w_{0,0}],
              [w_{1,0}],
              ... ,
              [w_{N-1,0}])

vector_addition = v + w
broadcasting = v + k
scalar_multiplication = k * v
```

1.1.2 Linear Combination

Linear Independence

Suppose there exists scalar $k \in \mathbf{R}$ and vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathbf{R}^{N\times 1}$. If $\boldsymbol{v}$ can be described by multiplying scalar k to $\boldsymbol{w}$ ($\boldsymbol{v} = k\boldsymbol{w}$), $\boldsymbol{v}$ is said to be *linearly dependent* to $\boldsymbol{w}$. Else, if $\boldsymbol{v} \neq k\boldsymbol{w}$, $\boldsymbol{v}$ is linearly independent to $\boldsymbol{w}$.

Now suppose there exists scalars $k_1, k_2, \ldots, k_j \in \mathbf{R}$ and $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_j \in \mathbf{R}^{N\times 1}$. If the only choice of $k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \ldots + k_j\boldsymbol{v}_j = 0$ is $k_1 = k_2 = \ldots = k_j = 0$, a set of vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_j$ is considered to be *linearly independent*. Otherwise, if $k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \ldots + k_j\boldsymbol{v}_j = 0$ is true for $k_1 = k_2 = \ldots = k_j$ not all 0, a set of vectors are *linearly dependent*: That is, if one of the vectors can be described by *combinations* of scalar multiplied vectors, $\boldsymbol{v}_i = k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \ldots + k_{i-1}\boldsymbol{v}_{i-1} + k_{i+1}\boldsymbol{v}_{i+1} + \ldots + k_j\boldsymbol{v}_j$, that set of vectors is said to be linearly dependent.

Linear Combination

Such combination, $k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \ldots + k_{i-1}\boldsymbol{v}_{i-1} + k_{i+1}\boldsymbol{v}_{i+1} + \ldots + k_j\boldsymbol{v}_j$, is called *Linear Combination*. It is called *linear* since all terms are linear (i.e. there are no squared terms).

Formally, the addition of scalar multipled vectors $k_1\boldsymbol{v}$ and $k_2\boldsymbol{w}$ is defined as a linear combination of $\boldsymbol{v}$ and $\boldsymbol{w}$.

$$k_1\boldsymbol{v} + k_2\boldsymbol{w} = k_1\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix} + k_2\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} k_1v_1 + k_2w_1 \\ k_1v_2 + k_2w_2 \\ \vdots \\ k_1v_N + k_2w_N \end{bmatrix}$$

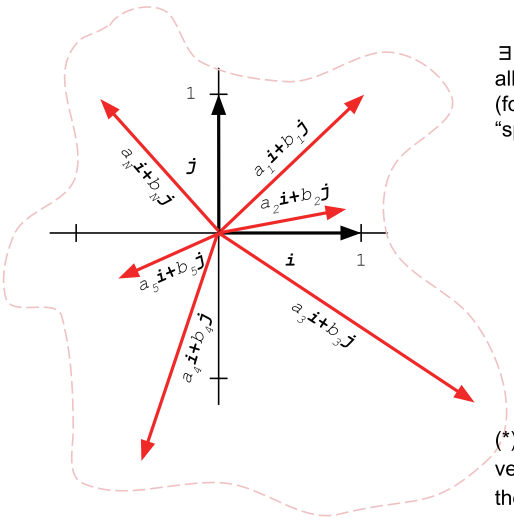
Suppose $k \in \mathbf{R}$ and $\boldsymbol{v} \in \mathbf{R}^{1\times1}$ where $\boldsymbol{v} \neq (0)$. In other words, suppose $\boldsymbol{v}$ is a 1-dimensional nonzero vector. For convenience, let us suppose $\boldsymbol{v} = (1)$. What would be the picture of all possible combinations of $k\boldsymbol{v}$? As one may have imagined, altering the scalar k - from the infinitely small case ($k \to 0$) to the infinitely large cases ($k \to \infty$ and $k \to -\infty$) - allows $k\boldsymbol{v}$ to describe the entire 1D space.

Now suppose $k_1, k_2 \in \mathbf{R}$ and $\boldsymbol{v}, \boldsymbol{w} \in \mathbf{R}^{2\times1}$, where $\boldsymbol{v}, \boldsymbol{w} \neq (0,0)$. Linear combinations of $k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2$ can describe all possible vectors of the entire 2D space, or a *plane*. This is always true except when either $\boldsymbol{v}$ or $\boldsymbol{w}$ is linearly dependent to each other. That is, when the pairs of such vectors line up.

Similarly, all possible linear combinations of 3 dimensional nonzero vectors $k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + k_3\boldsymbol{v}_3$ are able to describe vectors in the entire 3D space. This is always true except when one of the vectors is linearly dependent. That is, when one of the vectors is on the plane that the remaining two can describe.

The following video visually explores the idea of N vectors, linearly combined, can describe all possible vectors in N-dimensional space:

As shown, the linear combination of N linearly independent vectors, where such set of vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_N \in \mathbf{R}^{N\times1}$, can describe the vectors in the entire N-dimensional space. And such set of vectors is called the bases (or the basis vectors).



$\exists\ a_1,a_2,\ldots,a_N \in \mathbf{R}$, $b_1,b_2,\ldots,b_N \in \mathbf{R}$ and $\boldsymbol{v},\boldsymbol{w} \in \mathbf{R}^{2\times 1}$, all possible unique combinations of $a_i\boldsymbol{v} + b_i\boldsymbol{w}$ (for all i=1,…,N) can describe all possible vectors in 2D "space".

$$\boldsymbol{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \boldsymbol{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$a_i\boldsymbol{i} + b_i\boldsymbol{j} = \begin{bmatrix} a_i \\ b_i \end{bmatrix}$$

(*) ***v*** and ***w*** are used, instead of the standard basis vectors (sbv), because those are not the only bases in the 2-dimensional space. As far as the set of vectors are linearly independent, any sets could be the bases.