



Enter the Matrix

An introduction to Linear Algebra

Summary: Vectors and matrices, basically

Chapter I

Foreword

Linear algebra is a very important topic which is encountered almost everywhere in mathematics, computer science and physics.

It is the study of vector spaces, which consist of objects called vectors. Transformations of vectors, called linear maps, are generally represented as objects called "matrices" in the most usual (finite-dimensional) case. There also exist other, more complex, kinds of vector spaces: like real or complex polynomials, and function spaces. Linear algebra allows us to study these diverse subjects within a unifying framework. This makes the more complicated cases easier to understand, since their properties are generally the same as the simple cases, and so you can use the simple cases to mentally visualize what would happen in the complicated ones. For this reason, we will concentrate on the fundamental case of finite-dimensional vectors and matrices, with an emphasis on 2D and 3D, which can easily be visualized.

So what are some examples of the use of linear algebra?

Newtonian mechanics (everything concerning movement at the human scale) is full of vectors. Additionally, all "more complicated" forms of physics also generally depend on linear algebra or its extensions (multivariable calculus, topology, differential geometry, tensor algebra, operator theory...). One could describe quantum mechanics as the linear algebra and multivariable calculus over complex numbers. General relativity is just a fancy use of differential geometry and tensor algebra. Even signal theory can have a linear algebra "feel" to it, since signals are basically just functions, and numerical function spaces are just a special case of vector spaces.

Also, video games are basically just playful physical simulations: so video games use linear algebra a lot! Want Mario to jump? You will either need a "jump impulse vector" to force him upwards, and a "gravity" vector to bring him back down; or you will need a degree 2 polynomial to make the jump curve. Want to make cinema-level CGI? Raytracing is just a simulation of how light works, and that's just bouncing vectors called "rays". All of these objects are mostly defined through linear algebra. Not only that, but "2D" and "3D" (ie, the notion of "dimension"), are terms that originate from linear algebra. Computer graphics are chock-full of vectors, and their transformation through matrices. Want to rotate a 2D sprite for a fire spell? That's a matrix. Want

to create a 3D camera for an FPS? That's a matrix. The GPU was basically invented to handle linear algebra operations well (see below). High Performance Computing and Numerical Analysis thus depend on linear algebra a lot.

Statistics (and thus data science and machine learning) is also mostly based on linear algebra. Numerical data is often represented as vectors (a "data point cloud") in large-dimensional vector spaces. There are important analogies to draw between statistical concepts (respectively: covariance, variance, standard deviation, Pearson correlation coefficient, normalizing a Gaussian curve) and linear algebra operations (respectively: dot product, quadratic norm, norm, cosine of the angle between two vectors, normalizing a vector). Understanding the geometry of linear algebra is thus very useful to understand the geometry underlying statistics.

There are many, many more uses of linear algebra and its variants (in everything from cryptography to ecology) but what we listed here were the most frequent use cases in practice. Once you learn it, you will be seeing it more and more around you.

Because of its importance, because of how few prerequisites it actually needs when learning it (other than basic arithmetic), and because of all the domains of mathematics and science for which it opens the way, linear algebra has been chosen as the standard entry point into higher mathematics, all around the world. We hope you will enjoy how useful it is in practice as much as we do!

I.1 General Rules

- For this module, function prototypes are described in Rust, but you may use the language you want. There are a few constraints for your language of choice, though:
 - It must support generic types
 - It must support functions as first class citizens (example: supports lambda expressions)
 - Optional: support for operator overloading
- We recommend that you use paper if you need to. Drawing visual models, making notes of technical vocabulary... This way, it will be easier to wrap your head around the new concepts you will discover, and use them to build your understanding for more and more related concepts. Also, doing computations by head is really hard and error-prone, so having all the steps written down helps you figure out where you made a mistake when you did.
- Don't stay stuck because you don't know how to solve an exercise: make use of peer-learning! You can ask for help on Slack (42 or 42-AI) or Discord (42-AI) for example.
- You are not allowed to use any mathematical library, even the one included in your language's standard library, unless explicitly stated otherwise.
- For every exercise you may have to respect a given time and/or space complexity. These will be checked during peer review.
 - The time complexity is calculated relative to the number of executed instructions.
 - The space complexity is calculated relative to the maximum amount of memory allocated simultaneously.
 - Both have to be calculated against the size of the function's input (a number, the length of a string, etc...).

Chapter II

Introduction

For this module, we **highly** recommend the series of videos [Essence of linear algebra](#) by 3blue1brown, which provides a very good, comprehensive and visual explanation of this topic.

These videos are so good, you will probably want to watch them once before working on this module, you will be going back to specific parts of them during the module, and you will want to rewatch them once again after this module, perhaps a couple of months after. We're not kidding.

Many people to whom we had recommended these videos before getting into game physics programming, data science or computer graphics decided not to watch them until much later. The reaction was UNANIMOUS: those who had watched the videos had nothing but praise about how incredibly helpful they had been; those who had only watched the videos later in their learning of linear algebra deeply regretted the time that they wouldn't have wasted, had they watched the videos from the start.

Don't waste your time. WATCH. THE. VIDEOS. Even if you don't understand everything at first, the ordered presentation and the geometric visuals will provide you with a good amount of contextual background and mental models that will prove essential as you learn the various aspects of linear algebra.

II.1 Greek alphabet

As you may know, mathematics (and physics) use the Greek alphabet a lot. If you want to learn it, here it is:

| Lower case | Upper case | Name |
|------------|------------|---------|
| α | A | Alpha |
| β | B | Beta |
| γ | Γ | Gamma |
| δ | Δ | Delta |
| ϵ | E | Epsilon |
| ζ | Z | Zeta |
| η | E | Eta |
| θ | Θ | Theta |
| ι | I | Iota |
| κ | K | Kappa |
| λ | Λ | Lambda |
| μ | M | Mu |
| ν | N | Nu |
| ξ | Ξ | Xi |
| o | O | Omicron |
| π | Π | Pi |
| ρ | R | Rho |
| σ | Σ | Sigma |
| τ | T | Tau |
| υ | Υ | Upsilon |
| ϕ | Φ | Phi |
| χ | X | Chi |
| ψ | Ψ | Psi |
| ω | Ω | Omega |

II.2 Graphical Processing Units

A Graphical Processing Unit (GPU) is a piece of hardware present in a lot of modern computers, and supercomputers, which allows to perform parallel computations. The processing architecture of a GPU is referred to as SIMD (for "Single Instruction, Multiple Data"): what this means is that a GPU works well when it can run the same code (ie, functions without divergent conditional forks or gotos), at the same time, over different input values of the same type (typically, floating point numbers). You can see this as a factory where a sequence of the same machines do the same manipulation on objects, at the same time, on multiple parallel treadmills.

As stated by the name, GPUs were originally targeted towards graphical computations, which rely heavily on running linear algebra operations. Because of the ubiquity of linear algebra in mathematics, the use of GPUs was extended to more than video games and CGI, and most supercomputing today relies on GPU computation.

Many modern CPUs have also taken inspiration from the SIMD model of computation, and allow for specific SIMD operations.

The operations of linear algebra consist mostly of: summing vectors (a coordinate-wise

addition of two vectors), scaling vectors (multiplying all coordinates of a given vector by the same real or complex number), and matrix multiplications. Matrix multiplications are just a repeated succession of an operation called the **dot product** (or **inner product**) of two vectors. All of these operations are very well adapted to SIMD, since they are just a succession of additions or multiplications over arrays of data (which represent our vectors). Coding these various operations to understand them is a fundamental exercise. For this reason, you will get to code them all during this module.

II.3 Fused Multiply-Accumulate

Under the x86 architecture, there exist the instructions called: `vmadd132ps`, `vmadd213ps` and `vmadd231ps` (welcome to x86 :D), which allow us to compute what's called the **Fused Multiply-Accumulate** operation.

Check what these instructions do, and you will realize that they might become useful. ;)

The standard library of the C language (as well as that of many other languages) has a function to perform this operation (but we're not telling you which one, so search by yourself!).

II.4 Vector space

The following, which is a concise-but-terse list of theoretical info about vector spaces, can probably be useful for you to visit and revisit as you work on this module. It helps to work both practical applications and theory together, in order to gradually build up a deeper understanding. Don't feel discouraged if you don't understand everything immediately: that is obviously normal when approaching a new ecosystem of information.

If you remember the concept of "(algebraic) groups" from the Boolean Algebra module, this section will be a lot easier to understand (if you haven't done that module yet, we recommend you to do it, since sets, logic and algebraic structures are even more fundamental than linear algebra).

Understanding things from the point of view of algebraic structures can seem difficult at first, since there is a lot of vocabulary. However, this vocabulary is incredibly useful, because it transforms the mathematical properties of various sets into something simple, like knowing the rules of a card game. What you're allowed to do, and what is forbidden. If you can remember 4-5 board games' or card games' rules, you definitely have what it takes to learn the language of algebraic structures (often called "abstract algebra", which is a misleadingly scary name).

Actually, it's really not that complicated to learn, because it's basically ONLY a question of learning vocabulary, and not learning complex algorithmic methods. For this reason, we thought it was worthwhile to include the following: what follows is the

definition of a vector space, the principal type of structure that is studied in linear algebra, from the point of view of "algebraic structures" (ie, the rules that can be used over a certain set, in order to make that set become a "vector space").

A vector space V is a structure which associates:

- A **field** \mathbb{K} (roughly put: an algebraic structure where you can add, subtract, multiply and divide elements under the rules of usual arithmetic). The elements of \mathbb{K} are called "scalars". \mathbb{K} is generally chosen to be the real numbers or the complex numbers. Real numbers are generally represented as floats; complex numbers as pairs of floats, with a special multiplication and division. We generally denote scalars with Greek letters.
- A **commutative group** V (roughly put: an algebraic structure where you can add and subtract elements, under the rules of usual arithmetic; but not necessarily multiply them or divide them). The elements of V are called "vectors". For finite dimensions, V is always equivalent to \mathbb{K}^n , with n a natural number, and n is called the "dimension" of V . This means that in finite dimensions, V can ALWAYS be understood as a list/array/tuple containing n elements of \mathbb{K} . This also means that every field \mathbb{K} can be understood as a 1D vector space. We generally denote vectors with Latin letters.
- An operation, called scalar multiplication (or scaling multiplication), which allows us to combine elements of \mathbb{K} and V . We can't add or subtract an element of \mathbb{K} with an element of V , but we can multiply an element of \mathbb{K} and an element of V , which returns an element of V .

The operation of scalar multiplication must also fulfill the following properties:

- Scalar multiplication takes two elements, one in \mathbb{K} , one in V , and returns an element of V :

$$\lambda \in \mathbb{K}, \quad \forall u \in V, \quad \lambda u \in V$$

- Pseudo-distributivity on vectors (look up "distributivity"):

$$\forall \lambda \in \mathbb{K}, \quad \forall (u, v) \in V^2, \quad \lambda(u + v) = \lambda u + \lambda v$$

- Pseudo-distributivity on scalars (look up "distributivity"):

$$\forall (\lambda, \mu) \in \mathbb{K}^2, \quad \forall u \in V, \quad (\lambda + \mu)u = \lambda u + \mu u$$

- Multiplicative pseudo-associativity of scalar multiplication (look up "associativity"):

$$\forall (\lambda, \mu) \in \mathbb{K}^2, \quad \forall u \in V, \quad (\lambda \mu)u = \lambda(\mu u)$$

- Compatibility of the multiplicative identity of the field with scalar multiplication (look up "identity element"):

$$\forall u \in V, \quad 1_{\mathbb{K}}u = u$$

For the full, thorough list of the axioms/properties (ie, the "rules of the game") of vector spaces, do check the wikipedia page for [Vector space](#)

If the subjects of algebraic structures (which is EXCEEDINGLY helpful in understanding higher math) or complex numbers interest you, you can consult the following [PDFs](#), written by one of the authors of this module (the PDFs are also available in French).

Linear algebra also studies a special type of vector space, called a "Euclidean space" (also called [Inner Product Space](#) or IPS) to which we add another operator, the famous **dot product** (inner product). This operator takes two vectors and returns a scalar. Many other objects of study in physics and other domains that depend on linear algebra are just "vector spaces with some extra structure" (eg: topological vector spaces, metric spaces, manifolds, vector fields, \mathbb{K} -algebras, Lie algebras, tensor algebras, geometric algebras, function spaces of random vectors...)

II.5 Linear maps

Let V and W be two vector spaces over the same field \mathbb{K} .

Let $u, v \in V$ be two vectors and $\lambda \in \mathbb{K}$ be a scalar (where \mathbb{K} is the field associated with the vector space V).

The function $f : V \rightarrow W$ is said to be a **linear map** (or **linear transformation**) if:

$$\forall (u, v) \in V^2, \quad f(u + v) = f(u) + f(v) \quad (\text{aka "additivity"})$$

$$\forall \lambda \in \mathbb{K}, \forall u \in V, \quad f(\lambda u) = \lambda f(u) \quad (\text{aka "homogeneity of degree 1"})$$

If these properties are verified, we say that f is mapping values **linearly** from the vector space V to a new vector space W . Geometrically, "linearly" means that any two lines that are parallel in the input space are kept parallel after the transformation (those who watched 3blue1brown should know that by now).

These are necessary and sufficient conditions: to prove that f is linear, you must prove that the above two equations are true. See: [Necessity and sufficiency](#). Note that the second equation implies that the null vector 0_V of the input space V (the vector with only zeros as coefficients) is mapped to the null vector 0_W of the output space W .

You can also understand what is going on with the following diagrams (called **commutative diagrams**, in category theory):

$$\begin{array}{ccc} (u, v) \in V \times V & \xrightarrow{(f,f)} & (f(u), f(v)) \in W \times W \\ \downarrow +_V & & \downarrow +_W \\ u + v \in V & \xrightarrow{f} & f(u + v) = f(u) + f(v) \in W \end{array}$$

$$\begin{array}{ccc} (\lambda, u) \in K \times V & \xrightarrow{(id_K, f)} & (\lambda, f(u)) \in K \times W \\ \downarrow *_V & & \downarrow *_W \\ \lambda u \in V & \xrightarrow{f} & f(\lambda u) = \lambda f(u) = W \end{array}$$

or, in a more concise (but less explanatory) form:

$$\begin{array}{ccc} V \times V & \xrightarrow{(f,f)} & W \times W \\ \downarrow +_V & & \downarrow +_W \\ V & \xrightarrow{f} & W \end{array}$$

$$\begin{array}{ccc} K \times V & \xrightarrow{(id_K, f)} & K \times W \\ \downarrow *_V & & \downarrow *_W \\ V & \xrightarrow{f} & W \end{array}$$

These commutative diagrams, when true, mean that you can take whichever path from the starting point (two elements of V) to the end point (a single element of W), and you will arrive at the same result. The choice of first "applying f " or "using the vector space operator" doesn't matter, so long as the other operation is what follows. Try to stop for a second to understand how these diagrams are exactly equivalent to the equations for linearity written above. This will help you unpack what's happening in more detail.

Usually in linear algebra, a linear map f corresponds to a matrix. To be precise, if V is n -dimensional, and W is m -dimensional, and V and W are each provided with a basis (a chosen system of coordinates) then f can be represented uniquely by an $n \times m$ matrix, a matrix with n rows and m columns.

The set of linear maps from V to W is generally denoted $\mathcal{L}(V, W)$, and the space of matrices from V to W for given bases \mathcal{B} on V , and \mathcal{C} on W , is generally denoted $\text{Mat}_{\mathcal{B}, \mathcal{C}}(V, W)$, or more simply, $\mathbb{K}^{\dim(W) \times \dim(V)}$ (the output space's dimension is the left operand!).

So for example, the space of 2-by-3 matrices with real numbers as coefficients (aka 2-by-3 real matrices) is $\mathbb{R}^{2 \times 3}$, and corresponds to real linear maps from \mathbb{R}^3 to \mathbb{R}^2 , aka $\mathcal{L}(\mathbb{R}^3, \mathbb{R}^2)$.

A very important fact to note is that spaces of matrices are also vector spaces, since they abide by the vector space axioms (you can scale matrices, as well as add matrices of the same shape).

The following is an example of a square matrix $A \in \mathbb{R}^{3 \times 4}$ (a matrix of reals of size 3×4), representing a linear map f , which takes as input a 4-dimensional vector, and returns as output a 3-dimensional vector:

$$\begin{bmatrix} 8 & 2 & 3 & 3.73 \\ 7 & 5 & 0 & -1 \\ 17 & -32.3 & -4 & 5 \end{bmatrix}$$

A matrix, viewed as a linear transform, can be used as follows:

$$f(u) = v \Leftrightarrow Au = v$$

Which is a simple matrix multiplication (which you'll work on below).

Thus:

$A(u + v) = Au + Av$, ie, distributivity of matrix multiplication is additivity;

and

$A(\lambda u) = \lambda(Au) = (\lambda A)u$, 1-homogeneity means matrices commute with scalar multiplication of vectors.

The identity matrix (often denoted I_n), is a square matrix of size $n \times n$ which has no effect on the vector when multiplied with that vector.

The identity matrix $I_n \in \mathbb{R}^{n \times n}$ is defined as follows:

$$I_{ij} = \delta_{ij} = \delta_j^i$$

Where δ is the **Kronecker delta**. This means that the identity matrix has 1s along its principal diagonal, and 0s everywhere else.

For a given matrix A , the element of the matrix at the line i and column j is denoted A_{ij} .

Thus, the identity matrix I_3 of dimension 3 is defined as follows:

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

When you've got a handle on matrix multiplication, try to work out how $\forall v \in \mathbb{R}^3, I_3 v = v I_3 = v$.

II.6 Composition

The composition operator is defined as follows:

Let $f : A \rightarrow B$ (f is a function from a set A to a set B) and $g : B \rightarrow C$. Then there exists a function $h : A \rightarrow C$ such that:

$$\forall x \in A, h(x) = (g \circ f)(x) = g(f(x))$$

This is generally read a " g after f of x ", or " g of f of x ".

To put it simply, a composition of functions is a way to denote a specific succession of functions. This composition can exist if, and only if, the "middle space" B matches (ie, you can run g on f 's output if, and only if, f 's output type is the same as g 's input type).

When working on matrices, the composition operator for linear maps can be understood simply as successive matrix multiplications. This means that **matrix multiplication AB between matrices A and B works if, and only if, the number of columns of A matches the number of rows of B .**

Let A, B, C be three matrices of $\mathbb{K}^{n \times n}$, representing respectively three linear maps f, g, h of $\mathcal{L}(\mathbb{K}^n)$ and let u be a vector of \mathbb{K}^n . The composition of matrices can be written as follows:

$$(f \circ g \circ h)(u) = A \times B \times C \times u = ABCu$$



Matrix multiplication is associative, so parentheses can be omitted. However, for computational efficiency, it is often best to work out whether calculating AB first, and $(AB)C$ second, is faster than calculating BC first, and then $A(BC)$. The idea is to reduce the "middle space" that is the largest first, to be left with less computation on the second step.

Contents

| | | |
|-------------|---|-----------|
| I | Foreword | 1 |
| I.1 | General Rules | 3 |
| II | Introduction | 4 |
| II.1 | Greek alphabet | 4 |
| II.2 | Graphical Processing Units | 5 |
| II.3 | Fused Multiply-Accumulate | 6 |
| II.4 | Vector space | 6 |
| II.5 | Linear maps | 8 |
| II.6 | Composition | 11 |
| III | Code constraints | 14 |
| IV | Exercise 00 - Add, Subtract and Scale | 16 |
| V | Exercise 01 - Linear combination | 19 |
| VI | Exercise 02 - Linear interpolation | 21 |
| VII | Exercise 03 - Dot product | 23 |
| VII.1 | Inner product on complex vector spaces | 24 |
| VIII | Exercise 04 - Norm | 25 |
| VIII.1 | Triangle inequality | 26 |
| IX | Exercise 05 - Cosine | 27 |
| IX.1 | Pearson correlation coefficient | 29 |
| X | Exercise 06 - Cross product | 31 |
| X.1 | Wedge product and exterior algebra | 32 |
| XI | Exercise 07 - Linear map, Matrix multiplication | 34 |
| XI.1 | Special matrices | 36 |
| XII | Exercise 08 - Trace | 38 |
| XIII | Exercise 09 - Transpose | 40 |
| XIII.1 | Conjugate transpose | 41 |
| XIV | Interlude 00 - Solving systems of linear equations | 42 |
| XV | Exercise 10 - Reduced row-echelon form | 44 |

| | | |
|--------------|--|-----------|
| XVI | Exercise 11 - Determinant | 47 |
| XVI.1 | The determinant and the exterior algebra | 49 |
| XVII | Exercise 12 - Inverse | 50 |
| XVII.1 | Moore-Penrose pseudo-inverse | 52 |
| XVIII | Interlude 01 - Rank–nullity theorem | 53 |
| XIX | Exercise 13 - Rank | 55 |
| XX | Exercise 14 - Bonus: Projection matrix | 57 |
| XXI | Exercise 15 - Bonus: Complex vector spaces | 61 |

Chapter III

Code constraints

For this module, you will need to define two structures:

```
struct Vector::<K> {  
    // ...  
}  
  
struct Matrix::<K> {  
    // ...  
}
```

The above structures are declared in **Rust**, and correspond to a store of data, with no method. Feel free to adapt them to your language's most appropriate syntax (and to add your methods to them if that's the design choice you go for).

We recommend you to implement some utility functions, such as:

- A function to return the size/shape of a vector/matrix.
- A function to tell if a matrix is square.
- A function to print a vector/matrix on the standard output.
- A function to reshape a vector into a matrix, and vice-versa.

A couple of things of note.

In what follows, the prototypes of the function will define generic arguments for each functions. However, you only have to implement it for real numbers, which will be defined as the type `f32`, which is what you should understand when you see `K`. Do note, however, that there is a bonus at the end of this module which is to re-do every exercise with the field \mathbb{K} being implemented as your own `Complex` number type, rather than `f32`. If you've done your work in a nice and generic way, this shouldn't prove too much fixing.

Some of the prototypes given in Rust are more "pseudocode" than actual code: a good linear algebra library in Rust would leverage the Rust trait system, and you would thus need to specify the traits implemented by your generics in the signatures. Generally, it should even be expected that the generic types provided would actually have some form of type constraint in a real implementation (for example, making a `Field` type, and having the generic `K` extend `Field`; this is especially the case for functional languages). It is thus expected that the examples in some exercises, as they are, may not compile.

Also, in most of the exercises in this module, we may generally decide for V to be any sort of vector space, of finite dimension, over its scalar field \mathbb{K} . So V , as a generic, means a set that may contain scalars, or vectors, or matrices, or tensors, etc...

Finally, many of the exercises' functions store their result within one of their inputs. This is a choice. This type of function can be much faster when using your object as an accumulator, but you might want to have the inputs be immutable instead (ie, a "pure function"), and simply return your value. If your language of choice does not permit such object-oriented constructs, or if you simply prefer the pure function syntax, you are of course allowed to instead have a function that returns a scalar/vector/matrix as output, but keeps its inputs immutable. You may also implement both versions, if you find it worthwhile: the pure function, and the accumulating function. We do ask some amount of coherence though: if you choose to implement only one version, stick to your choice throughout the module.

Conclusion: there is thus some level of freedom of interpretation left to the student and their evaluator as to verify if the function signature is valid. What is really important is that the underlying mathematical function is implemented in a way that is:

- generic, so that the overall library of code written during this module may work with other fields than the real numbers;
- coherent, so that the library has a consistent style and usage;
- faithful, as-in, the function implemented is a true implementation of the required mathematical concept, and not something else.




In the following exercises, n , m and p each correspond to one size component of a vector/matrix. This means that $O(nm)$ for an n -by- m matrix signifies 'no more operations than a constant multiple of the amount of rows times the amount of columns'.



Unless specified otherwise, all matrices shown in the examples are in column-major order.

Chapter IV

Exercise 00 - Add, Subtract and Scale

| | |
|---|---------------|
|  | Exercise : 00 |
| Add, Subtract and Scale | |
| Allowed mathematical functions : None | |
| Maximum time complexity : $O(n)$ | |
| Maximum space complexity : $O(n)$ | |

IV.0.1 Goal

You must write functions that can add and subtract two vectors, or two matrices, of the same size; and a function to multiply a vector, or a matrix, by a scalar (ie, "scaling").

You must also turn in a main function in order to test your functions, ready to be compiled (if necessary) and run.

IV.0.2 Instructions

You must write 3 functions.

- The first one must compute the addition of 2 vectors.
- The second one must compute the subtraction of a vector by another vector.
- The third one must compute the scaling of a vector by a scalar.

If the mathematical operation is nonsensical (ie, summing a vector and a scalar, or

vectors of different sizes), the result is undefined.

The prototype of the functions to write are the following:

```
impl<K> Vector<K> {  
    fn add(&mut self, v: &Vector<K>);  
    fn sub(&mut self, v: &Vector<K>);  
    fn scl(&mut self, a: K);  
}  
  
impl<K> Matrix<K> {  
    fn add(&mut self, v: &Matrix<K>);  
    fn sub(&mut self, v: &Matrix<K>);  
    fn scl(&mut self, a: K);  
}
```

Examples:

```
let mut u = Vector::from([2., 3.]);  
let v = Vector::from([5., 7.]);  
u.add(v);  
println!("{}", u);  
// [7.0]  
// [10.0]  
  
let mut u = Vector::from([2., 3.]);  
let v = Vector::from([5., 7.]);  
u.sub(v);  
println!("{}", u);  
// [-3.0]  
// [-4.0]  
  
let mut u = Vector::from([2., 3.]);  
u.scl(2.);  
println!("{}", u);  
// [4.0]  
// [6.0]  
  
let mut u = Matrix::from([  
    1., 2.  
    3., 4.  
]);  
let v = Matrix::from([  
    7., 4.  
    -2., 2.  
]);  
u.add(v);
```

```
println!("{}", u);  
// [8.0, 6.0]  
// [1.0, 6.0]  
  
let mut u = Matrix::from([  
    1., 2.  
    3., 4.  
]);  
let v = Matrix::from([  
    7., 4.  
    -2., 2.  
]);  
u.sub(v);  
println!("{}", u);  
// [-6.0, -2.0]  
// [5.0, 2.0]  
  
let mut u = Matrix::from([  
    1., 2.  
    3., 4.  
]);  
u.scl(2.);  
println!("{}", u);  
// [2.0, 4.0]  
// [6.0, 8.0]
```

Chapter V

Exercise 01 - Linear combination

| | |
|---|---------------|
| Λ | Exercise : 01 |
| Linear combination | |
| Allowed mathematical functions : fused multiply-add function | |
| Maximum time complexity : $O(n)$ | |
| Maximum space complexity : $O(n)$ | |

V.0.1 Goal

You must write a function that computes a linear combination of the vectors provided, using the corresponding scalar coefficients.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.



The n in the complexity limit is the dimension of the vector/matrix, ie, the total number of coordinates.

V.0.2 Instructions

- Let $u = (u_1, \dots, u_k) \in V^k$ be a list of size k , containing vectors (V is a vector space).
- Let $\lambda = (\lambda_1, \dots, \lambda_k) \in \mathbb{K}^k$ be a list of size k , containing scalars.

You must calculate the linear combination of the vectors of u scaled by their respective

coefficients in λ .

If the two arrays provided as input are not of the same size, or if the array's contents are incoherent, the result is undefined.

The prototype of the function to write is the following:

```
fn linear_combination::<K>(u: &[Vector<K>], coefs: &[K]) -> Vector<K>;
```

Examples:

```
let e1 = Vector::from([1., 0., 0.]);
let e2 = Vector::from([0., 1., 0.]);
let e3 = Vector::from([0., 0., 1.]);


let v1 = Vector::from([1., 2., 3.]);
let v2 = Vector::from([0., 10., -100.]);

println!("{}", linear_combination<Vector<f32>, f32>([e1, e2, e3], [10., -2.,
    0.5]));
// [10.]
// [-2.]
// [0.5]

println!("{}", linear_combination<Vector<f32>, f32>([v1, v2], [10., -2.]));
// [10.]
// [0.]
// [230.]
```

Chapter VI

Exercise 02 - Linear interpolation

| | |
|---|---|
|  | Exercise : 02 |
| | Linear interpolation |
| | Allowed mathematical functions : <code>fused multiply-add function</code> |
| | Maximum time complexity : $O(n)$ |
| | Maximum space complexity : $O(n)$ |

VI.0.1 Goal

You must write a function that computes a linear interpolation between two objects of the same type.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.



The n in the complexity limit is the dimension of the vector/matrix, ie, the total number of coordinates.

VI.0.2 Instructions

- Let $t \in [0; 1](\subset \mathbb{R})$ (ie, t is real, and $0 \leq t \leq 1$) be a scalar.
- Let $f : (V \times V \times [0; 1]) \rightarrow V$ be the function to implement.

The value of t allows us to "slide" between the two values.

- If $t = 0$, then $f(u, v, t) = u$.
- If $t = 1$, then $f(u, v, t) = v$.
- If $t = 0.5$, then the function returns a value at the exact middle in between of u and v (the isobarycenter, which can be understood as the center of gravity, of the two points).

The prototype of the function to write is the following:

```
fn lerp:<V>(u: V, v: V, t: f32) -> V;
```

Examples:


```
println!("{}", lerp(0., 1., 0.));  
// 0.0  
  
println!("{}", lerp(0., 1., 1.));  
// 1.0  
  
println!("{}", lerp(0., 1., 0.5));  
// 0.5  
  
println!("{}", lerp(21., 42., 0.3));  
// 27.3  
  
println!("{}", lerp(Vector::from([2., 1.]), Vector::from(4., 2.), 0.3));  
// [2.6]  
// [1.3]  
  
println!("{}", lerp(Matrix::from([[2., 1.], [3., 4.]]), Matrix::from([[20.,  
    10.], [30., 40.]]), 0.5));  
// [[11., 5.5]  
// [16.5, 22.]
```



Even though linear interpolation is most often used to get points on the segment between u and v , technically, the formula should give you any point on the infinite line drawn by the points u and v , if any value for t is accepted. This is relevant when defining rays in raytracing, for example.

Chapter VII

Exercise 03 - Dot product

| | |
|---|---------------|
|  | Exercise : 03 |
| Dot product | |
| Allowed mathematical functions : fused multiply-add function | |
| Maximum time complexity : $O(n)$ | |
| Maximum space complexity : $O(n)$ | |

VII.0.1 Goal

You must write a function that computes the dot product of two vectors of the same dimension.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

VII.0.2 Instructions

Let $u, v \in V$, where V is a vector space of finite dimension over the real numbers \mathbb{R} (represented as the type `f32`).

The function must compute and return the scalar $\langle u|v \rangle = u \cdot v$, called the dot product, or inner product, of u and v .

If both vectors have different dimensions, the behaviour is undefined.

The prototype of the function to write is the following:


```
impl<K> Vector::<K> {  
    fn dot::<K>(&self, v: Vector::<K>) -> K;  
}
```

Examples:

```
let mut u = Vector::from([0., 0.]);  
let v = Vector::from([1., 1.]);  
println!("{}", u.dot(v));  
// 0.0  
  
let mut u = Vector::from([1., 1.]);  
let v = Vector::from([1., 1.]);  
println!("{}", u.dot(v));  
// 2.0  
  
let mut u = Vector::from([-1., 6.]);  
let v = Vector::from([3., 2.]);  
println!("{}", u.dot(v));  
// 9.0
```

VII.1 Inner product on complex vector spaces

If you're curious about vector spaces of complex numbers, and already know enough about complex numbers, you might want to look up the terms **conjugate transpose**, **sesquilinear algebra**, and **Pre-Hilbert space**.



Those interested in electronics, control systems in engineering, or quantum mechanics should *definitely* study complex numbers and sesquilinear algebra.

Chapter VIII

Exercise 04 - Norm

| | |
|--|---------------|
| Δ | Exercise : 04 |
| Norm | |
| Allowed mathematical functions : fused multiply-add function, pow, max | |
| Maximum time complexity : $O(n)$ | |
| Maximum space complexity : $O(n)$ | |

VIII.0.1 Goal

You must write functions that compute different kinds of norms.

You must also turn in a main function in order to test your functions, ready to be compiled (if necessary) and run.

VIII.0.2 Instructions

Let v be a vector of a vector space V .

You must implement the following norms:

- 1-norm: $\|v\|_1$ (also called the Taxicab norm or Manhattan norm)
- 2-norm: $\|v\|$ or $\|v\|_2$ (also called the Euclidean norm)
- ∞ -norm: $\|v\|_\infty$ (also called the supremum norm)

The prototype of the functions to write are the following:

```
impl<V> Vector::<V> {
    fn norm_1::<V>(&mut self) -> f32;
    fn norm::<V>(&mut self) -> f32;
    fn norm_inf::<V>(&mut self) -> f32;
}
```

Examples:

```
let u = Vector::from([0., 0., 0.]);
println!("{}", {}, {}, {}, {}, u.norm_1(), u.norm(), u.norm_inf());
// 0.0, 0.0, 0.0

let u = Vector::from([1., 2., 3.]);
println!("{}", {}, {}, {}, {}, u.norm_1(), u.norm(), u.norm_inf());
// 6.0, 3.74165738, 3.0

let u = Vector::from([-1., -2.]);
println!("{}", {}, {}, {}, {}, u.norm_1(), u.norm(), u.norm_inf());
// 3.0, 2.236067977, 2.0
```



Norms always return real numbers, even for complex-valued vectors.

VIII.1 Triangle inequality

Now would be a good time to get interested in what's called the **Triangle inequality** for Euclidean norms. If you're curious about infinite-dimensional vector spaces (like spaces of sequences or function), you might want to look at **Hölder norms**, and the **Minkowski inequality**.

You might also want to check out norms specific to vector spaces of matrices, like the **Frobenius norm**.

You might also want to check out the "cousin" of norms which can be negative for a given coordinate. These are called **pseudonorms** and are important, for example, in special relativity (also see *Lorentz transform*, *split-complex numbers*, *hyperbolic geometry*).

Side note: Encyclopedias with math content like Wikipedia, EncyclopediaOfMath, and Wolfram MathWorld can be useful from time to time, but generally, they're better to review a concept rather than learn it, so don't get discouraged if something doesn't immediately seem clear on these platforms!

Chapter IX

Exercise 05 - Cosine

| | |
|--|---------------|
| Δ | Exercise : 05 |
| Cosine | |
| Allowed mathematical functions : fused multiply-add function | |
| Maximum time complexity : $O(n)$ | |
| Maximum space complexity : $O(n)$ | |

IX.0.1 Goal

You must write functions that compute the cosine of the angle between two given vectors.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.



You should use the functions you wrote during the two previous exercises.

IX.0.2 Instructions

Let $u, v \in \mathbb{K}^n$ be vectors.

You must compute $\cos(u, v)$, the cosine of the angle between the two vectors u and v .



Reminder: The usage of the standard library's `cos` function is forbidden, of course.

If one or both vectors are zero, the behaviour is undefined.

If the vectors are of different sizes, the behavior is undefined.

The prototype of the function to write is the following:

```
fn angle_cos::<K>(u: &Vector::<K>, v: &Vector::<K>) -> f32;
```

Examples:

```
let u = Vector::from([1., 0.]);
let v = Vector::from([1., 0.]);
println!("{}", angle_cos(&u, &v));
// 1.0

let u = Vector::from([1., 0.]);
let v = Vector::from([0., 1.]);
println!("{}", angle_cos(&u, &v));
// 0.0

let u = Vector::from([-1., 1.]);
let v = Vector::from([ 1., -1.]);
println!("{}", angle_cos(&u, &v));
// -1.0

let u = Vector::from([2., 1.]);
let v = Vector::from([4., 2.]);
println!("{}", angle_cos(&u, &v));
// 1.0

let u = Vector::from([1., 2., 3.]);
let v = Vector::from([4., 5., 6.]);
println!("{}", angle_cos(&u, &v));
// 0.974631846
```

IX.1 Pearson correlation coefficient

There are a lot of analogies between linear algebra and statistics. The "vector" of probability and statistics is generally taken to be a **random variable**. A random variable is a numerical function. Its input space is a set of possible outcomes, like $\Omega = \{1, 2, 3, 4, 5, 6\}$ for what a dice can roll, with each outcome having a distinct probability (for example, a fair dice would have each possible outcome be a probability of $\frac{1}{6}$). Such an input space is called a **probability space**. The output space of a random variables is a space of values, generally taken to be \mathbb{R} (for example, you gain 3€ for every even roll of the dice, and lose 3€ for every odd roll).

With this setup, the **covariance** is the dot product for spaces of random variables, the **variance** is their quadratic norm (a squared norm, the dot product of a vector with itself), and the **standard deviation** is their norm (the square root of a quadratic norm).

The random variable's analogue for the cosine is called the **Pearson correlation coefficient**. This coefficient is calculated from two random variables using covariance as your dot product, and then normalizing, like in the usual vector cosine formula.

- If this coefficient is close to 1 (the cosine is close to 1, so the angle is small, and the two vectors are close to being colinear, with the same orientation), the two random variables are highly correlated.

- If this coefficient is close to 0 (the cosine is close to 0, so the angle is close to a right angle, and the two vectors are close to being perpendicular), the two random variables are not correlated.

- If this coefficient is close to -1 (the cosine is close to -1 , so the angle is a large (almost flat) angle, and the two vectors are close to being colinear, but with opposite orientation), the two random variables are highly anticorrelated.

This is one of the many examples that highlight how linear algebra can explain the geometry underlying statistics.


More generally, data science and statistics study data point clouds, which act like "having only a bunch of samples of the output values taken by a random variable, but no info about the random variable's set of events/outcomes (inputs), nor their probabilities". For this very frequent case, we instead use a concept called a "probability distribution", which is a way to infer information about the set of possible outcomes Ω and its probabilities, without actually knowing this space, simply based on the prevalence of some values over others. This is most often how data science is done, since we mostly have observations, but would like to figure out the underlying model (a hidden random variable), which should explain how these observations came to be.

Finally, note that there exist random variables that can give multiple output values (for example, if you roll a 3, you gain 2€, but lose 5 candy bars: that's a 2D vector of real values as output). These can be equivalently understood as vectors of random variables, and are called **multivariate random variables**, or **random vectors**. They are very important in data science since, generally, we're dealing with multiple different

observations - for example, looking at the distribution of size, weight and income (3 different values) over a population of individuals.

Chapter X

Exercise 06 - Cross product

| | |
|---|---------------|
|  | Exercise : 06 |
| Cross product | |
| Allowed mathematical functions : fused multiply-add function | |
| Maximum time complexity : N/A | |
| Maximum space complexity : N/A | |

X.0.1 Goal

You must write a function that computes the cross product of two 3-dimensional vectors.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

X.0.2 Instructions

Let $u, v \in V = \mathbb{R}^3$ be vectors.

You must implement a function that computes the cross product $u \times v$.

If one or both vectors are not 3-dimensional, the behaviour is undefined.

The prototype of the function to write is the following:

```
fn cross_product::<K>(u: &Vector::<K>, v: &Vector::<K>) -> Vector::<K>;
```


Examples:

```
let u = Vector::from([0., 0., 1.]);
let v = Vector::from([1., 0., 0.]);
println!("{}", cross_product(&u, &v));
// [0.]
// [1.]
// [0.]

let u = Vector::from([1., 2., 3.]);
let v = Vector::from([4., 5., 6.]);
println!("{}", cross_product(&u, &v));
// [-3.]
// [6.]
// [-3.]

let u = Vector::from([4., 2., -3.]);
let v = Vector::from([-2., -5., 16.]);
println!("{}", cross_product(&u, &v));
// [17.]
// [-58.]
// [-16.]
```

X.1 Wedge product and exterior algebra

There exists a very important, anticommutative product of two vectors, called the **wedge product** (or **exterior product**, or **outer product**) which returns an object called a bivector. Whereas a vector (or 1-vector) is a 1D geometric object (a line with a certain magnitude and orientation: visualized as a little arrow with a certain length and arrow-head), a bivector (or 2-vector) is a 2D geometric object (a plane with a certain magnitude and orientation: visualized as a little parallelogram with a certain area and a direction of rotation). This parallelogram corresponds precisely to the parallelogram formed by the two input vectors.

The outer product extends to bivectors and vectors, giving trivectors (3-vectors, which correspond to a 3D oriented volume), etc. The vector space of all k -vectors up to a maximal dimension n is called the **n -dimensional exterior algebra over a vector space V** and is denoted $\Lambda^n(V)$. These exterior algebras are extremely important in theories of calculus over higher dimensions, and calculus on curved spaces (a field of mathematics called **differential geometry**). If you are interested in advanced physics in any way, you will probably be dealing with these whether you know it or not.


The cross product is technically the dual (which you can understand as the orthogonal complement, here, the vector perpendicular to the parallelogram) of the wedge product of two vectors (bivector/parallelogram). The dual of a k -vector in an n -dimensional is always an $(n - k)$ -vector. This is why the cross product only makes sense in 3D (and

is thus a pretty poor mathematical construct). The wedge product, however, is always consistent.

Note that in the context of **geometric algebras**, a kind of vector space whose objects are called "multivectors", and are sums of k -vectors of various ranks (various values for k) there is a fundamental construct called the **geometric product**, which is the sum of the inner/dot product and the outer/wedge product. Geometric algebras, while complex to approach at first, offer a profound and beautiful framework that goes beyond linear algebra and often makes higher mathematics that rely on linear algebra much more intuitive and natural. Geometric algebra is, so to speak, "the best way" to extend the objects of linear algebra, and make it an improved system.

Chapter XI

Exercise 07 - Linear map, Matrix multiplication

| | |
|---|---------------|
|  | Exercise : 07 |
| Linear map, Matrix multiplication | |
| Allowed mathematical functions : fused multiply-add function | |
| Maximum time complexity : see below | |
| Maximum space complexity : see below | |

XI.0.1 Goal

You must write functions that multiply a matrix by a vector or a matrix by a matrix.

You must also turn in a main function in order to test your functions, ready to be compiled (if necessary) and run.

XI.0.2 Instructions

Let $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$ and $u \in \mathbb{R}^n$ where $(m, n, p) \in \mathbb{N}^3$ (represented as variables of type `u32`).

You must implement functions that compute:

- Au (which returns a vector in \mathbb{R}^m) (max time complexity $O(nm)$, max space complexity $O(nm)$)

- AB (which returns a matrix in $\mathbb{R}^{m \times p}$) (max time complexity $O(nmp)$, max space complexity $O(nm + mp + np)$)

The prototype of the functions to write are the following:

```
impl<K> Matrix::<K> {  
    fn mul_vec::<K>(&mut self, vec: Vector::<K>) -> Vector::<K>;  
    fn mul_mat::<K>(&mut self, mat: Matrix::<K>) -> Matrix::<K>;  
}
```

Examples:

```
let u = Matrix::from([  
    [1., 0.],  
    [0., 1.],  
]);  
let v = Vector::from([4., 2.]);  
println!("{}", u.mul_vec(&v));  
// [4.]  
// [2.]  
  
let u = Matrix::from([  
    [2., 0.],  
    [0., 2.],  
]);  
let v = Vector::from([4., 2.]);  
println!("{}", u.mul_vec(&v));  
// [8.]  
// [4.]  
  
let u = Matrix::from([  
    [2., -2.],  
    [-2., 2.],  
]);  
let v = Vector::from([4., 2.]);  
println!("{}", u.mul_vec(&v));  
// [4.]  
// [-4.]  
  
let u = Matrix::from([  
    [1., 0.],  
    [0., 1.],  
]);  
let v = Matrix::from([  
    [1., 0.],  
    [0., 1.],  
]);
```

```
]);  
println!("{}", u.mul_mat(&v));  
// [1., 0.]  
// [0., 1.]  
  
let u = Matrix::from([  
    [1., 0.],  
    [0., 1.],  
]);  
let v = Matrix::from([  
    [2., 1.],  
    [4., 2.],  
]);  
println!("{}", u.mul_mat(&v));  
// [2., 1.]  
// [4., 2.]  
  
let u = Matrix::from([  
    [3., -5.],  
    [6., 8.],  
]);  
let v = Matrix::from([  
    [2., 1.],  
    [4., 2.],  
]);  
println!("{}", u.mul_mat(&v));  
// [-14., -7.]  
// [44., 22.]
```

XI.1 Special matrices

Here, you might be interested in learning about the different types of "special" square matrices:

- matrices of the form λI_n , which act like a scaling by the value λ
- diagonal matrices, which scale each vector component independently, and are easy to raise to an integer power.
- orthogonal matrices, which conserve angles and lengths, and represent a combination of rotations and (bilateral-symmetric, ie, mirror) reflexions.

There are other special matrices, like those used to represent translations in raytracing (homogeneous model), or those used in diagonalization or SVD, but the above kinds of matrices are the ones that are most important, since you will see them most often.

They also make special stable groups (in the "algebraic structure" sense of the term), called Lie Groups, which are very important in physics, since they conserve some mathematical quantity (something called a generalized "symmetry"), which translates into a conservation of some physical quantity (see **Noether's theorem**).

Chapter XII

Exercise 08 - Trace

| | |
|--|---------------|
| Δ | Exercise : 08 |
| Trace | |
| Allowed mathematical functions : None | |
| Maximum time complexity : $O(n)$ | |
| Maximum space complexity : N/A | |

XII.0.1 Goal

You must write a function that computes the trace of the given matrix.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XII.0.2 Instructions

Let $A \in \mathbb{K}^{n \times n}$ be a square matrix, where \mathbb{K} is the real numbers (represented as the type `f32`) and $(m, n) \in \mathbb{N}^2$ (represented as two variables of type `u32`).

The function must compute and return $Tr(A)$

The prototype of the function to write is the following:

```
impl<K> Matrix::<K> {  
    fn trace::<K>(&mut self) -> K;  
}
```

Examples:


```
let u = Matrix::from([
  [1., 0.],
  [0., 1.],
]);
println!("{}", u.trace());
// 2.0
```

```
let u = Matrix::from([
  [2., -5., 0.],
  [4., 3., 7.],
  [-2., 3., 4.],
]);
println!("{}", u.trace());
// 9.0
```

```
let u = Matrix::from([
  [-2., -8., 4.],
  [1., -23., 4.],
  [0., 6., 4.],
]);
println!("{}", u.trace());
// -21.0
```


Chapter XIII

Exercise 09 - Transpose

| | |
|---|---------------|
|  | Exercise : 09 |
| Transpose | |
| Allowed mathematical functions : None | |
| Maximum time complexity : $O(nm)$ | |
| Maximum space complexity : $O(nm)$ | |

XIII.0.1 Goal

You must write a function that computes the transpose matrix of a given matrix.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XIII.0.2 Instructions

Let $A \in \mathbb{K}^{m \times n}$ where \mathbb{K} is the real numbers (represented as the type `f32`) and $(m, n) \in \mathbb{N}^2$ (represented as values of the type `u32`).

The function must return the transpose matrix $B \in \mathbb{K}^{n \times m}$

The prototype of the function to write is the following:

```
impl<K> Matrix::<K> {  
    fn transpose::<K>(&mut self) -> Matrix::<K>;  
}
```



The time complexity here is relative to assignment of values, unlike the other exercises, where the base operation counted is addition or multiplication.

XIII.1 Conjugate transpose

Here, you might want to know that the transposition of complex matrices uses something called the **conjugate transpose**, using the **conjugation** operation of the complex numbers.

The reason why this is different is linked to the fact that every complex number $z = a + ib$ can be represented uniquely as a 2-by-2 real matrix of the form:

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix}$$

called a rotation-scaling matrix. Adding or multiplying or inverting these matrices is precisely equivalent ("isomorphic") to the same operations on the complex numbers; while the conjugate of a complex number is precisely the transpose of this real matrix.

If you imagine replacing every 1-by-1 coordinate in an n -by- m complex matrix by its corresponding 2-by-2 rotation-scaling matrix, then run a simple transpose over this $2n$ -by- $2m$ matrix, you will get a $2m$ -by- $2n$ real matrix. If you change the order of operations, and do the transpose first, before expanding the coefficients into 2-by-2 blocks, you'll realize that you need to use the conjugate transpose, rather than a simple transpose, to get to the same final $2m$ -by- $2n$ matrix.

There's a commutative diagram hiding here. Can you figure it out? ;)

Chapter XIV

Interlude 00 - Solving systems of linear equations

Linear systems of equations are systems of equations where you only have **linear combinations** of variables (a linear combination is a sum of scaled variables, where there are no exponents on the variables). You may have already used linear systems of equations. Here is an example:

$$\begin{cases} 2x + 3y = 8 \\ -7x + 4y = 2 \end{cases}$$

If you've now understood matrix multiplication, it's easy to see that you can represent the above system in matrix form:

$$\begin{bmatrix} 2 & 3 \\ -7 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ 2 \end{bmatrix}$$

And it can thus be written algebraically as:

$$Ab = c$$

It can also be abbreviated as an "augmented matrix":

$$\left[\begin{array}{cc|c} 2 & 3 & 8 \\ -7 & 4 & 2 \end{array} \right]$$

Now, to solve the system, we want to compute the coordinate values of the unknown vector b , so we need to find some way to "pass it to the other side of the equal sign", like we did for usual equations in a single variable. Since matrix multiplication isn't *commutative*, and there are matrices that reduce spatial information available (these matrices are called

singular and have no inverse), matrix division is particular. In matrix form, we use **the inverse matrix** of a given matrix to divide. But the sides to which you multiply the inverse, and whether a specific matrix's inverse exists, are both very important questions which deserve careful consideration.

The inverse of the square matrix A of size n -by- n , if it exists, is denoted A^{-1} and is defined such as:

$$A^{-1}A = AA^{-1} = I_n$$

Where I_n is the identity matrix of \mathbb{R}^n .

Thus, if it exists, you can use the inverse matrix to solve the system (compute the desired values for the vector b containing all the unknowns):

$$\begin{aligned} Ab = c &\Leftrightarrow A^{-1}Ab = A^{-1}c && \text{(left-multiplication by the inverse on both sides)} \\ &\Leftrightarrow I_n b = A^{-1}c && \text{(reduction of inverses to the identity)} \\ &\Leftrightarrow b = A^{-1}c && \text{(computable solution to the equation)} \end{aligned}$$

For the above system, the inverse matrix is:

$$A^{-1} = \begin{bmatrix} \frac{4}{29} & -\frac{3}{29} \\ \frac{7}{29} & \frac{2}{29} \end{bmatrix}$$


Thus, the solution is:

$$b = \begin{bmatrix} \frac{26}{29} \\ \frac{60}{29} \end{bmatrix}$$

Also, the inverse matrix is not necessarily easy, nor fast, to compute. But you'll figure this out in the following exercises. ;)

Chapter XV

Exercise 10 - Reduced row-echelon form

| | |
|---|---------------|
|  | Exercise : 10 |
| Reduced row-echelon form | |
| Allowed mathematical functions : None | |
| Maximum time complexity : $O(n^3)$ | |
| Maximum space complexity : $O(n^2)$ | |

XV.0.1 Goal

You must write a function that computes the reduced row-echelon form of the given matrix.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XV.0.2 Instructions

Let $A \in \mathbb{K}^{m \times n}$, where \mathbb{K} is the real numbers (represented as the type `f32`) and $(m, n) \in \mathbb{N}^2$ (represented as two values of type `u32`).

The function must return the row-echelon form of the matrix.



The results may be a bit approximate (within reason) since making the algorithm numerically stable can sometimes be difficult.

The prototype of the function to write is the following:

```
impl<K> Matrix::<K> {
    fn row_echelon::<K>(&mut self) -> Matrix::<K>;
}
```

Examples:

```
let u = Matrix::from([
    [1., 0., 0.],
    [0., 1., 0.],
    [0., 0., 1.],
]);
println!("{}", u.row_echelon());
// [1.0, 0.0, 0.0]
// [0.0, 1.0, 0.0]
// [0.0, 0.0, 1.0]

let u = Matrix::from([
    [1., 2.],
    [3., 4.],
]);
println!("{}", u.row_echelon());
// [1.0, 0.0]
// [0.0, 1.0]

let u = Matrix::from([
    [1., 2.],
    [2., 4.],
]);
println!("{}", u.row_echelon());
// [1.0, 2.0]
// [0.0, 0.0]

let u = Matrix::from([
    [8., 5., -2., 4., 28.],
    [4., 2.5, 20., 4., -4.],
    [8., 5., 1., 4., 17.],
]);
println!("{}", u.row_echelon());
// [1.0, 0.625, 0.0, 0.0, -12.1666667]
// [0.0, 0.0, 1.0, 0.0, -3.6666667]
// [0.0, 0.0, 0.0, 1.0, 29.5 ]
```

Chapter XVI

Exercise 11 - Determinant

| | |
|--|---------------|
| Δ | Exercise : 11 |
| Determinant | |
| Allowed mathematical functions : fused multiply-add function | |
| Maximum time complexity : $O(n^3)$ | |
| Maximum space complexity : $O(n^2)$ | |

XVI.0.1 Goal

You must write a function that computes the determinant of the given matrix.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XVI.0.2 Instructions

Let $A \in \mathbb{K}^{n \times n}$ where \mathbb{K} is the real numbers (represented as the type `f32`) and $n \in \mathbb{N}$ and $n \leq 4$ (represented as a value of the type `u32`).

The function must return the determinant $\det(A)$ of the matrix.

If the matrix is not square, the behaviour is undefined.

Since algorithms to compute the determinant fast and accurately for higher dimensions tend to be pretty complex, we have limited the required cases for which you must compute the determinant to dimensions 4 and below.



For the simpler methods, you will want to use one specific method per dimension, and perhaps "recycle" the technique from lower dimensions when nothing seems to be available...



You must be able to explain during evaluation:

- What happens when $\det(A) = 0$
 - What the determinant represents geometrically in the vector space after using the matrix for a linear transformation.
- If you don't know this, you should have watched 3blue1brown and you didn't, you bad student.



The results may be a bit approximate (within reason) since making the algorithm numerically stable can be difficult.

The prototype of the function to write is the following:

```
impl<K> Matrix::<K> {  
    fn determinant::<K>(&mut self) -> K;  
}
```

Examples:

```
let u = Matrix::from([  
    [ 1., -1.],  
    [-1., 1.],  
]);  
println!("{}", u.determinant());  
// 0.0
```

```
let u = Matrix::from([  
    [2., 0., 0.],  
    [0., 2., 0.],  
    [0., 0., 2.],  
]);  
println!("{}", u.determinant());  
// 8.0
```

```
let u = Matrix::from([
```

```
[8., 5., -2.],  
[4., 7., 20.],  
[7., 6., 1.],  
]);  
println!("{}", u.determinant());  
// -174.0  
  
let u = Matrix::from([  
  [ 8., 5., -2., 4.],  
  [ 4., 2.5, 20., 4.],  
  [ 8., 5., 1., 4.],  
  [28., -4., 17., 1.],  
]);  
println!("{}", u.determinant());  
// 1032
```

XVI.1 The determinant and the exterior algebra

If you remember the aside on the exterior algebra from before, you might want to know that n -vectors inside an n -dimensional exterior algebra tend to behave a lot like scalars, and are referred to as "pseudoscalars" for this reason.

The determinant is actually the magnitude of the pseudoscalar (the measure of the n -parallelepiped) which is created by the successive wedge product of all the columns vectors of a square matrix. Read that again, slowly.

This should make sense to you if you understand the geometric nature of the determinant.

Chapter XVII

Exercise 12 - Inverse

| | |
|---|---------------|
| Δ | Exercise : 12 |
| Inverse | |
| Allowed mathematical functions : fused multiply-add function | |
| Maximum time complexity : $O(n^3)$ | |
| Maximum space complexity : $O(n^2)$ | |

XVII.0.1 Goal

You must write a function that computes the inverse matrix of a given matrix.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XVII.0.2 Instructions

Let $A \in \mathbb{K}^{n \times n}$ where \mathbb{K} is the real numbers (represented as the type `f32`) and $n \in \mathbb{N}$ (represented as a value of the type `u32`).

The function must return the inverse matrix $A^{-1} \in \mathbb{K}^{n \times n}$ such that $A^{-1}A = I_n$, where I_n is the identity matrix.

If the matrix is singular, the function should return an error.

If the matrix is not square, the behaviour is undefined.



The results may be a bit approximate (within reason) since making the algorithm numerically stable can be difficult. Specifically, finding the inverse is generally numerically unstable when the matrix is close to being singular; or when the matrix is very large.

The prototype of the function to write is the following:

```
impl<K> Matrix::<K> {  
    fn inverse::<K>(&mut self) -> Result<Matrix::<K>>;  
}
```

Examples:

```
let u = Matrix::from([  
    [1., 0., 0.],  
    [0., 1., 0.],  
    [0., 0., 1.],  
]);  
println!("{}", u.inverse());  
// [1.0, 0.0, 0.0]  
// [0.0, 1.0, 0.0]  
// [0.0, 0.0, 1.0]  
  
let u = Matrix::from([  
    [2., 0., 0.],  
    [0., 2., 0.],  
    [0., 0., 2.],  
]);  
println!("{}", u.inverse());  
// [0.5, 0.0, 0.0]  
// [0.0, 0.5, 0.0]  
// [0.0, 0.0, 0.5]  
  
let u = Matrix::from([  
    [8., 5., -2.],  
    [4., 7., 20.],  
    [7., 6., 1.],  
]);  
println!("{}", u.inverse());  
// [0.649425287, 0.097701149, -0.655172414]  
// [-0.781609195, -0.126436782, 0.965517241]  
// [0.143678161, 0.074712644, -0.206896552]
```

XVII.1 Moore-Penrose pseudo-inverse

n -by- m matrices have a "kind" of inverse, called the **Moore-Penrose pseudo-inverse**. It corresponds to a best fit to the "least squares problem". It also plays a role in the Singular Value Decomposition, which is perhaps the most important matrix decomposition algorithm (and there are more of these algorithms than you'd think!).

Chapter XVIII

Interlude 01 - Rank–nullity theorem

The rank-nullity theorem states:

Let E and F be two vector spaces and let $f \in \mathcal{L}(E, F)$ be a linear map. Then:

$$\text{rank}(f) + \text{null}(f) = \dim(\text{im}(f)) + \dim(\text{ker}(f)) = \dim(E)$$

Where,

- $\text{rank}(f)$ is the rank of the linear transformation's matrix (dimension of the image of f)
- $\text{null}(f)$ is the nullity of a linear transformation's matrix (dimension of the kernel of f)
- $\dim(E)$ is the dimension of the (input) vector space E

XVIII.0.1 But wait, what's an image? What's a kernel?

Let f be a function defined as: $f : E \rightarrow F$ (mapping values from the set E to the set F)

The **image** of the function f , which respects $\text{im}(f) \subseteq F$, is defined as the set of all the values $y \in F$ such that:

$$\forall y \in \text{im}(f), \exists x \in E, f(x) = y$$

Simply put, the image of a function is the set of points reached by function f in the output space.

The **kernel** of the function f , which respects $\text{ker}(f) \subseteq E$, is defined as the set of all the values $x \in E$ such that:

$$\forall x \in \ker(f), f(x) = 0_F$$


In linear algebra, the kernel is commonly called the **nullspace** and it represents the space of all points that get squished at the origin of the vector space after the linear transformation.



Watch out, kernel is one of the most overused terms in computer science and mathematics. It has a LOT of different meanings.

Chapter XIX

Exercise 13 - Rank

| | |
|---|---------------|
|  | Exercise : 13 |
| Rank | |
| Allowed mathematical functions : None | |
| Maximum time complexity : $O(n^3)$ | |
| Maximum space complexity : N/A | |

XIX.0.1 Goal

You must write a function that computes the rank of a matrix.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

XIX.0.2 Instructions

Let $A \in \mathbb{K}^{n \times m}$ where \mathbb{K} is the real numbers (represented as the type `f32`) and $(m, n) \in \mathbb{N}^2$ (represented as two values of type `u32`).

The function must return the rank of the matrix: $rank(A)$



You must be able to explain during evaluation what the rank represents.

The prototype of the function to write is the following:


```
impl<K> Matrix::<K> {  
    fn rank::<K>(&mut self) -> usize;  
}
```

Examples:

```
let u = Matrix::from([  
    [1., 0., 0.],  
    [0., 1., 0.],  
    [0., 0., 1.],  
]);  
println!("{}", u.rank());  
// 3  
  
let u = Matrix::from([  
    [ 1., 2., 0., 0.],  
    [ 2., 4., 0., 0.],  
    [-1., 2., 1., 1.],  
]);  
println!("{}", u.rank());  
// 2  
  
let u = Matrix::from([  
    [ 8., 5., -2.],  
    [ 4., 7., 20.],  
    [ 7., 6., 1.],  
    [21., 18., 7.],  
]);  
println!("{}", u.rank());  
// 3
```

Chapter XX

Exercise 14 - Bonus: Projection matrix

| | |
|---|---------------|
|  | Exercise : 14 |
| Bonus: Projection matrix | |
| Allowed mathematical functions : <code>tan</code> , fused multiply-add function | |
| Maximum time complexity : N/A | |
| Maximum space complexity : N/A | |

XX.0.1 Goal

You must write a function that computes a projection matrix to be used to render 3D objects.

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

A display software is provided with the subject. It'll allow you to test this exercise.

XX.0.2 Instructions

The projection matrix is an important part of modern rendering pipelines. In OpenGL for example, each point in the 3D space has to be converted to a point in the screen's 2D space.

To do so, we will be using the following formula.

Let $M, V, P \in \mathbb{R}^{4 \times 4}$ be the **model**, **view** and **projection** matrices (these are some-

times also called **object-to-world**, **world-to-camera** and **camera-to-screen** matrices, respectively) and $s, p \in \mathbb{R}^4$ be the position of the point on screen and the position of the point in the 3D space.

$$s = PVMp$$

The view matrix V is defined as:

$$V = RT$$

$$T = \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where:

- T is a translation matrix, placing the camera at the origin of the vector space.
- R is a rotation matrix, rotating the vector space according to the camera's angle.
- p is the vector representing the camera's position
- (X, Y, Z) is the vector space's basis after transformation.

The reason for the matrices to be in $3 + 1 = 4$ dimensions is because we are working with **Homogeneous coordinates** instead of **Cartesian coordinates**. Linearity implies that the origin is conserved: this means that translations are not linear maps. Homogeneous coordinates are a trick, which uses an extra dimension with the space \mathbb{K}^{n+1} , to be able to use translations in a way that makes them linear.

In this exercise, we will only build the Projection matrix. To do so, the function will take the following arguments:

- **fov**: Field-of-view, the angle of the cone of vision
- **ratio**: The window size ratio: $\frac{w}{h}$ where w and h are the width and height of the rendering window
- **near**: The distance of the near plane.
- **far**: The distance of the far plane.

The prototype of the function to write is the following:

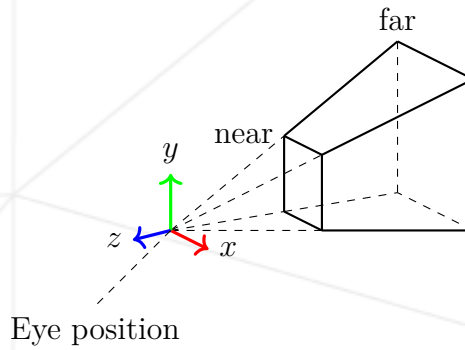


Figure XX.1: A graphical representation of the above explanations. Everything to be rendered on screen is located inside of the frustum.

```
fn projection(fov: f32, ratio: f32, near: f32, far: f32) -> Matrix::<f32>;
```

To test your projection matrix, you can plug it in the display software like so:

Example:

```
$ cat proj
1., 0., 0., 0.
0., 1., 0., 0.
0., 0., 1., -1.
0., 0., 0., 0.
$ ./display
```

(The above example will not work properly for a projection)



The display software expects the input matrix to be in **row-major** order. Its **Normalized Device Coordinates** are in the ranges $[-1,1]$ for the X and Y axes, and $[0,1]$ for the Z axis.




A projection matrix made for the wrong NDC and in the wrong major order may appear to work at first glance, but will fail under careful scrutiny. Try playing around with different near and far planes, and justify the changes that you observe.



Figure XX.2: The expected result

Chapter XXI

Exercise 15 - Bonus: Complex vector spaces

| | |
|---|---------------|
|  | Exercise : 15 |
| Bonus: Complex vector spaces | |
| Allowed mathematical functions : N/A | |
| Maximum time complexity : N/A | |
| Maximum space complexity : N/A | |

XXI.0.1 Goal

You've now succeeded in coming to grips with all the fundamental constructs of linear algebra.

But you've only ever used one field of scalars: the real numbers. Sure it's the most used, but it's certainly not the only one.

Do all the previous exercises again, this time interpreting \mathbb{K} in the function signatures, not as the field \mathbb{R} of real numbers, but as the field \mathbb{C} of complex numbers. Vector spaces over complex numbers are notoriously hard to visualize. However they are essential in much of electronics, signal analysis, quantum mechanics, and have even seen applications in machine learning (with complex-valued tensors).

You must also turn in a main function in order to test your function, ready to be compiled (if necessary) and run.

If you're feeling frisky, you might want to look into **finite fields**, try to build a finite field type, and build your vector spaces over that. These are easier to visualize than vector spaces over complex numbers. You can for example represent any vector space of

dimension 2, based on a finite field, as a space of tiles over the surface of a donut! :D



This bonus exercise is the reason why we kept most of our declarations with a generic K , rather than replacing K with f32 in the declarations by default. This is meant to showcase a powerful aspect of abstract algebra: genericity. Through a concept called a "functor", you can often mix-and-match algebraic structures like you would Lego bricks to build more interesting mathematical spaces, just like generic types work in computer science!

Contact

You can contact our association, 42AI, by email: contact@42ai.fr

You are also welcome to join the association on [42AI slack](#) and/or apply to [one of the association's teams](#).

Acknowledgements

This subject is the result of a collective work, we would like to thanks:

- Luc Lenôtre - [llenotre@student.42.fr](mailto:lленотре@student.42.fr)
- Tristan Duquesne - tduquesn@student.42.fr