Numerical Linear Algebra A1 Recap

01/05/2025

The lectures below refer to Trefethen's book on numerical linear algebra

1. Lecture 3 - Norms

Diclaimer: The norm's chapter has pretty abstract concepts, some of them are not very **intutive**, so try to abstract and accept they exist for now, later we will show that they are very useful.

1.1. Vector norms

Definition 1.1.1 (Norm): A **norm** is a function $\| \cdot \| : \mathbb{C}^m \to \mathbb{R}$ that satisfies 3 properties:

1. $\|x\| \geq 0$, and $\|x\| = 0 \Leftrightarrow x = 0$
2. $\|x + y\| \leq \|x\| + \|y\|$
3. $\|\alpha x\| = |\alpha| \|x\|$

We normally see the 2-norm, or the **Euclidian Norm**, that represents the **size** of a vector. Based on that, we can define a **p-norm**.

Definition 1.1.2 (p-norm): The **p-norm** of $x \in \mathbb{C}^n$ (or $\|x\|_p$) is defined as:

$$\|x\|_p = \left(\sum_{i=1}^n |x_i|^p\right)^{\frac{1}{p}}$$

So we can have multiple types of norms, from 1 to ∞, and we define it as well!

Definition 1.1.3 (Inifinite-norm): The **infinite norm** of $x \in \mathbb{C}^n$ (or $\|x\|_\infty$) is defined as:

$$\|x\|_\infty = \max |x_i|$$

You probably wondering "why would I need something like this"? But trust me, it will be useful in the future! There is a pretty useful type of norm (According to the book), that is called the **weighted norm**.

Definition 1.1.4 (Weighted norm): The **weighted norm** of $x \in \mathbb{C}^n$ is:

$$\|x\|_W = \|Wx\| = \left(\sum_{i=1}^n |w_{ii} x_i|^p\right)^{\frac{1}{p}}$$

Where W is a **diagonal matrix** and p is an arbitrary number

1.2. Matrix Norms

WHAT?? MATRICES HAVE NORMS???? Yes, my young Padawan! The book tells that we could see a matrix as a vector in a $m \times n$ space, and we could use any mn-norm to measure it, but some norms are more useful than the ones already discussed.

Definition 1.2.1 (Induced norm): Given $A \in \mathbb{C}^{m \times n}$, the induced norm $\|A\|_{m \to n}$ is the smallest integer for wich the inequality holds:

$$\|Ax\|_m \leq C \|x\|_n$$

In other words:

$$\|A\|_{m \to n} = \sup_{x \neq 0} \frac{\|Ax\|_m}{\|x\|_n}$$

This definition may seem stupid and useless by now, but it will be very useful when we see about errors and conditioning.

A useful norm we may say is the ∞-norm of a Matrix

Definition 1.2.2 (Infinite norm of a Matrix): Given $A \in \mathbb{C}^{m \times n}$, if a_j is the j^{th} row of A, $\|A\|_\infty$ is defined by:

$$\|A\|_\infty = \max_{1 \le i \le m} \|a_i\|_1$$

1.3. Cauchy-Schwarz & Hölder Inequalities

When we are using norms, usually it is difficult to compute p-norms with high values of p, so we manage them using inequalities! A very useful inequality is the Hölder inequality:

Definition 1.3.1 (Hölder Inequality): Given $1 \le p, q \le \infty$, and $\frac{1}{p} + \frac{1}{q} = 1$, then, for any vectors x, y:

$$|x^* y| \le \|x\|_p \|y\|_q$$

and the Cauchy-Schwarz inequality is a special case where $p = q = 2$

1.4. Bounding $\|AB\|$

We can bound $\|AB\|$ as we do with vector norms

Theorem 1.4.1: Given $A \in \mathbb{C}^{l \times m}$,$B \in \mathbb{C}^{m \times n}$ and $x \in \mathbb{C}^n$, then the induced norm of AB must satisfy:

$$\|AB\|_{l \to n} \le \|A\|_{l \to m} \|B\|_{m \to n}$$

Proof: $\|ABx\|_l \le \|A\|_{l \to m} \ \|Bx\|_m \le \|A\|_{l \to m} \ \|B\|_{m \to n} \ \|x\|_n$ □

1.5. Generalization of Matrices Norms

We saw that a norm follows 3 properties, we define a general matrix norm the same way!!

Definition 1.5.1: Given matrices A and B, a norm $\| \cdot \| : \mathbb{C}^{m \times n} \to \mathbb{R}^+$ is a function that follow these 3 properties:

1. $\|A\| \ge 0$, and $\|A\| = 0 \Leftrightarrow A = 0$
2. $\|A + A\| \le \|A\| + \|B\|$
3. $\|\alpha A\| = |\alpha| \|A\|$

the most important one is the **Frobenius Norm**, defined as:

Definition 1.5.2: Given a matrix $A \in \mathbb{C}^{m \times n}$, its **Frobenius Norm** is defined as:

$$\|A\|_F = \left(\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2\right)^{\frac{1}{2}} = \sqrt{tr(A^* A)} = \sqrt{tr(AA^*)}$$

Theorem 1.5.1: $\|AB\|_F \le \|A\|_F \ \|B\|_F$

Proof: $\|AB\|_F = \left(\sum_{i=1}^m \sum_{j=1}^n |c_{ij}|^2\right)^{\frac{1}{2}} \le \left(\sum_{i=1}^m \sum_{j=1}^n \left(\|a_i\|_2 \|b_j\|_2\right)^2\right)^{\frac{1}{2}} = \left(\sum_{i=1}^m \|a_i\|_2^2 \sum_{j=1}^n \|b_j\|_2^2\right)^{\frac{1}{2}} = \|A\|_F \|B\|_F$ □

2. Lecture 4 and 5 - The SVD

Quick Disclaimer: When we start talking about the factorization itself, we are going to talk about matrices in $\mathbb{C}^{m \times n}$ with $m \geq n$, because it's the most common when we talk about real problems, rarely is the situations with more variables then equations

Aaaaaah, the SVD, why it exists? What does it mean? Remember that, in Linear Algebra, when we have a base of a Vector Space and a Linear Transformation, we know how the Linear Transformation affects **every** vector on that Vector Space? No? Let me refresh your memmory:

Theorem 2.1: Given $\{a_j\}$ $(1 \leq j \leq n)$ being the base of a Vector Space and T a Linear Transformation in that space, if we know how T affects the vectors of the base, we know how T affects **every** vector on that space

Proof: Being v a vector on the described Vector Space, we know v can be expressed as

$$v = \alpha_1 a_1 + \ldots + \alpha_n a_n$$

Applying T on v

$$T(v) = T(\alpha_1 a_1 + \ldots + \alpha_n a_n) \Rightarrow T(v) = \alpha_1 T(a_1) + \ldots + \alpha_n T(a_n)$$

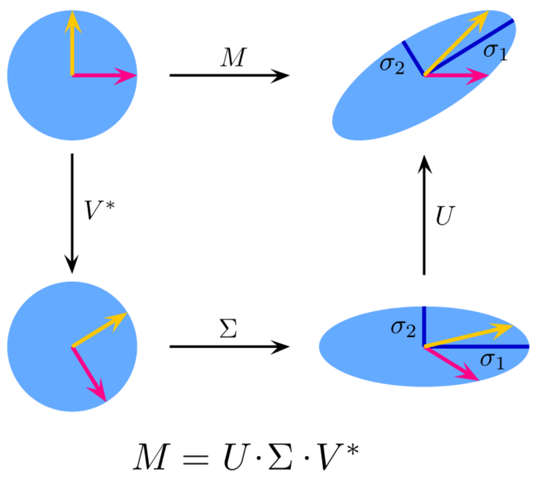
This implies that, if we know a base of the Vectorial Space and how T affects it, we know how T affects every vector on the space $\square$

RIGHT! Memory refreshed, why did I say that? Remember that matrices are linear transformations? So if we have a base $\{s_j\}$ of a Vectorial Space S, we can know what happens to every linear combination of $\{s_j\}$ if we apply A on it right? Right!

We can resume the operations we do in vectors in two: **stretch**, and **rotate**, so basically, when we apply a linear transformation on a vector, we are rotating it, then stretching it.

Okay, but why am I saying that? Where the hell is the S.V.D? Well, I basically already described the S.V.D to you! When we apply A as a linear transformation, if we do the operations described early, do you agree we can decompose A as a matrix product of Orthogonal Matrices and Diagonal Matrices? What? Why? When? Wait, young padawan! Remember I said a linear transformation can be resumed in stretching and rotating vectors? Do you remember what kind of matrices do EXACTLY what I said? Yes, orthogonal matrices do rotations and diagonal matrices do stretching.

Now we can introduce that classic visualization of how S.V.D works, imagine a orthonormal base in $\mathbb{R}^2$, looks what happens if we apply A on it:



(Change the M in the image for A)

Based on that, we can define that, given $A \in \mathbb{C}^{m \times n}$:

$$Av_j = \sigma_j u_j$$

Where v_j and u_j are from two different orthonormal bases and $\sigma_j \in \mathbb{C}$.

2.1. Reduced Form

We can rewrite this equation as a matrix product!

$AV = \hat{U}\hat{\Sigma}$

Where

$$V = \begin{pmatrix} | & & | \\ v_1 & \dots & v_n \\ | & & | \end{pmatrix}, \Sigma = \begin{pmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{pmatrix}, U = \begin{pmatrix} | & & | \\ u_1 & \dots & u_n \\ | & & | \end{pmatrix}$$

This is known as the **reduced** SVD factorization. We can see that V is a square orthogonal matrix (For Av_j be a valid multiplication, $v_j \in \mathbb{C}^n$), so we can rewrite A as:

$$A = \hat{U}\hat{\Sigma}V^*$$

2.2. Full SVD

Okay, if $v_j \in \mathbb{C}^n$ and $Av_j = \sigma_j u_j$, then $u_j \in \mathbb{C}^m$! That means, beside the u vectors we added in $\hat{U}$, we have $m - n$ more orthonormal vectors to the columns of $\hat{U}$, finding those vectors, we can build another matrix U that the columns are a orthonormal base of $\mathbb{C}^m$, that means the new matrix U is orthogonal!

$$V = \begin{pmatrix} | & & | \\ v_1 & \dots & v_n \\ | & & | \end{pmatrix}, U = \begin{pmatrix} | & & | \\ u_1 & \dots & u_m \\ | & & | \end{pmatrix}$$

Nice! But what about the $\hat{\Sigma}$ matrix? How does it change? Well, we want to maintain V and U as we wanted right? Well, the thing we did was add columns to $\hat{U}$, so, in the multiplication, we only need those columns to disappear, how we do that? Multiplying by 0! So, before, $\hat{\Sigma}$ was a square matrix with the single values on the diagonal, specifically, n single values. If we added $m - n$ vectors on U, we can add $m - n$ zeros on $\hat{\Sigma}$, so our new matrix multiplication is

$$A = U\Sigma V^*$$

$$\begin{pmatrix} | & & | \\ a_1 & \dots & a_n \\ | & & | \end{pmatrix} = \begin{pmatrix} | & & | \\ u_1 & \dots & u_m \\ | & & | \end{pmatrix} \begin{pmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ - & 0 & - \\ & \vdots & \end{pmatrix} \begin{pmatrix} -v_1- \\ \dots \\ -v_n- \end{pmatrix}$$

2.3. Formal Definition

Definition 2.3.1: Given $A \in \mathbb{C}^{m \times n}$ with $m \geq n$, the Singular Value Decomposition of A is:

$$A = U\Sigma V^*$$

where $U \in \mathbb{C}^{m \times m}$ is unitary, $V \in \mathbb{C}^{n \times n}$ is unitary and $\Sigma \in \mathbb{C}^{m \times n}$ is diagonal. For **convinience**, we denote:

$$\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \dots \geq \sigma_n$$

Where σ_j is the j-th entry of Σ

Okay, we saw a intuitive method for seeing that every matrix has this decomposition, but how do we prove it mathematically?

Theorem 2.3.1: Every $A \in \mathbb{C}^{m \times n}$ matrix has a S.V.D decomposition

Proof: Let $\{v_j\}$ be an orthonormal base of $\mathbb{C}^n$, $\{u_j\}$ an orthonormal base of $\mathbb{C}^m$, $Av_j = \sigma_j u_j$, U_1 and V_1 be unitary matrices of columns $\{u_j\}$ and $\{v_j\}$ respectively and that, for every matrix with less than m lines and n columns the factorization is valid:

$$A = U_1 S V_1^* \Leftrightarrow U_1^* A V_1 = S$$

Então temos $S = \begin{pmatrix} \sigma_1 & w^* \\ 0 & B \end{pmatrix}$ onde σ_1 é 1×1, w^* é $1 \times (n-1)$ e B é $(m-1) \times (n-1)$. Beleza, mas o que é w? Bem, podemos chegar nesse resultado fazendo umas manipulações com $\left\| \begin{pmatrix} \sigma_1 & w^* \\ 0 & B \end{pmatrix} \begin{pmatrix} \sigma_1 \\ w \end{pmatrix} \right\|_2$:

$$\left\| \begin{pmatrix} \sigma_1 & w^* \\ 0 & B \end{pmatrix} \begin{pmatrix} \sigma_1 \\ w \end{pmatrix} \right\|_2^2 \geq \sigma_1^2 + w^* w$$

What? Why this is valid? Because:

$$\|Mx\|_2 \leq \|M\|_2 \ \|x\|_2 \Rightarrow \|M\|_2 \geq \frac{\|Mx\|_2}{\|x\|_2}$$

If we set $x = \begin{pmatrix} \sigma_1 \\ w \end{pmatrix}$ and $M = S$, then we have:

$$Mx = \begin{pmatrix} \sigma_1^2 + \|w\|^2 \\ Bw \end{pmatrix} \Rightarrow \|M\|_2 \geq \frac{|\sigma_1^2 + \|w\|^2|^2 + \|Bw\|^2}{\sigma_1^2 + \|w\|^2}$$

But notice that the numerator is always greater than the denominator, so that means

$$\frac{|\sigma_1^2 + \|w\|^2|^2 + \|Bw\|^2}{\sigma_1^2 + \|w\|^2} \geq \sigma_1^2 + \|w\|^2 = (\sigma_1^2 + w^* w)^{\frac{1}{2}} \left\| \begin{pmatrix} \sigma_1 \\ w \end{pmatrix} \right\|$$

Now we can go back to see what w is! Well, now is easy! We know that $\|S\|_2 = \|U_1^* A V_1\|_2 = \|A\|_2 = \sigma_1$ because U_1 and V_1 are orthogonal. That means $\|S\|_2 \geq (\sigma_1^2 + \|w\|^2)^{\frac{1}{2}} \Rightarrow \sigma_1 \geq (\sigma_1^2 + \|w\|^2)^{\frac{1}{2}} \Leftrightarrow \sigma_1^2 \geq \sigma_1^2 + \|w\|^2 \Rightarrow w = 0$.

By the inductive hypothesis described in the start of the proof, we know $B = U_2 \Sigma_2 V_2^*$, so we can easily write A as

$$A = U_1 \begin{pmatrix} 1 & 0 \\ 0 & U_2 \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 \\ 0 & \Sigma_2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & V_2^* \end{pmatrix}^* V_1^*$$

That is a S.V.D of A, using the base case of $m = 1$ and $n = 1$, we finish the existence's proof □

2.4. Change of Basis

Given $b \in \mathbb{C}^m$, $x \in \mathbb{C}^n$ and $A \in \mathbb{C}^{m \times n}$, $A = U \Sigma V^*$ we can get the coordinates of b on the basis of the columns of U and x in the columns of V. Just to remember:

Definition 2.4.1: Given $w \in V$ where V is a Vector Space, $\exists! x_1, ..., x_n \in \mathbb{C}$ such that $w = v_1 x_1 + ... + v_n x_n$ where $\{v_j\}$ is a base of V. The vector $\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$, also denoted as $[w]_v$, is the w's **coordinate vector** in the v base

Getting back, we can express $[b]_u = U^* b$ and $[x]_v = V^* x$, but why?

Theorem 2.4.1: Given a orthonormal base $\{v_k\}$ of V and $w \in V$, then

$$([w]_v)_j = v_j^* w$$

Proof:

$$w = \alpha_1 v_1 + ... + \alpha_n v_n$$

$$v_j^* w = \alpha_1 v_j^* v_1 + ... + \alpha_n v_j^* v_n$$

Knowing that $\{v_j\}$ is a orthonormal base, the product $\alpha_i v_j^* v_i$ is equal to 0 if $j \neq i$ and equal to α_i if $j = i$, that is:

$$v_j^* w = \alpha_j$$

□

Ok, now that we remembered all these properties, we can express the relation $b = Ax$ in terms of $[b]_u$ and $[x]_v$, let's see:

$$b = Ax \Leftrightarrow U^* b = U^* Ax = U^* U \Sigma V^* x \Leftrightarrow U^* b = \Sigma V^* x$$

$$\Leftrightarrow [b]_u = \Sigma [x]_v$$

So we can reduce A to the Σ matrix and b and x to its coordinates on u base and v base

2.5. S.V.D vs Eigenvalue Decomposition

We can do something similar with the eigenvalue decomposition. Given $A \in \mathbb{C}^{m \times m}$ with linear independent eigenvectors, i.e we can express $A = S \Lambda S^{-1}$ with the columns of S being the eigenvectors of A and Λ is a diagonal matrix with the eigenvalues of A as entries.

Defining $b, x \in \mathbb{C}^m$ satisfying $b = Ax$, we can write:

$$[b]_{s^{-1}} = S^{-1} b \text{ and } [x]_{s^{-1}} = S^{-1} x$$

Where I'm denoting s^{-1} as the base expressed by the columns of S^{-1}, then the new expanded expression is:

$$b = Ax \Leftrightarrow S^{-1} b = S^{-1} Ax = S^{-1} S \Lambda S^{-1} x \Leftrightarrow S^{-1} b = \Lambda S^{-1} x$$

$$[b]_{s^{-1}} = \Lambda [x]_{s^{-1}}$$

2.6. Matrix Properties with SVD

For the next properties, let $A \in \mathbb{C}^{m \times n}$ and $r \leq \min(m, n)$ be the number of non-zero singular values

Theorem 2.6.1: $\operatorname{rank}(A) = r$

Proof: The rank of a diagonal matrix is the number of non-zero entries, well, if $A = U \Sigma V^*$, we know U and V are full-rank, then the rank of A must be the same as Σ, that is, r □

Theorem 2.6.2: $C(A) = \operatorname{span}\{u_1, ..., u_r\}, C(A^*) = \operatorname{span}\{v_1, ...v_r\}, N(A) = \operatorname{span}\{v_{r+1}, ..., v_n\}, N(A^*) = \operatorname{span}\{u_{r+1}, ..., u_m\}$

Proof: Let's remember how each matrix is structured:

$$A = \begin{pmatrix} | & & | \\ u_1 & \cdots & u_m \\ | & & | \end{pmatrix} \begin{pmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_r & \\ & & & 0 \\ & & & & \ddots \end{pmatrix} \begin{pmatrix} -v_1^*- \\ \vdots \\ -v_n^*- \end{pmatrix}$$

It's easy to see why $C(A) = \operatorname{span}\{u_1, ..., u_r\}$, because the entries of Σ makes only possible to span the first r columns of U.

About $N(A) = \operatorname{span}\{v_{r+1}, ..., v_n\}$, notice how, if we do Av_j $r + 1 \leq j \leq n$, the first r lines will turn to 0 (All v_k are orthonormal between them) and, because the diagonal entries after the r-th one are 0, then we have U times the 0 matrix

To see the A^*'s properties, let's transpose A

$$A^* = \begin{pmatrix} | & & | \\ v_1 & \cdots & v_n \\ | & & | \end{pmatrix} \begin{pmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_r & \\ & & & 0 \\ & & & & \ddots \end{pmatrix} \begin{pmatrix} -u_1^*- \\ \vdots \\ -u_m^*- \end{pmatrix}$$

So, again, is easy to see $C(A^*) = \operatorname{span}\{v_1, ...v_r\}$ and, using the same argument shown before, $N(A^*) = \operatorname{span}\{u_{r+1}, ..., u_m\}$ □

Theorem 2.6.3: $\|A\|_2 = \sigma_1$ and $\|A\|_F = \sqrt{\sigma_1^2 + \ldots + \sigma_r^2}$

Proof:
1. $\|A\|_2 = \|U\Sigma V\|_2 = \|\Sigma\|_2$, as we denoted before, from all entries, σ_1 is the greatest, that means $\|A\|_2 = \|\Sigma\|_2 = \sigma_1$
2. We know that $\|A\|_F = \sqrt{\operatorname{tr}(A^* A)} = \sqrt{\operatorname{tr}(V\Sigma^* U^* U \Sigma V^*)} = \sqrt{\operatorname{tr}(V\Sigma^*\Sigma V^*)}$. We also know that $\operatorname{tr}(A) = \lambda_1 + \ldots + \lambda_n$ with λ_j being the eigenvalues of A, and w can clearly see that the eigenvalues of $V\Sigma^*\Sigma V^*$ are σ_j^2, therefore $\|A\|_F = \sqrt{\sigma_1^2 + \ldots + \sigma_r^2}$

□

Theorem 2.6.4: $\sigma_j = \sqrt{\lambda_j}$ with σ_j being the singular values of A and λ_j the eigenvalues of $A^* A$

Proof: $A^* A = V\Sigma^* U^* U \Sigma V^* = V\Sigma^*\Sigma V^*$ □

Theorem 2.6.5: if $A = A^*$, then the singular values of A are the absolute values of A's eigenvalues

Proof: By the Spectral Theorem, we know A has an eigenvalue decomposition

$$A = Q\Lambda Q^*$$

We can rewrite it as

$$A = Q|\Lambda| \operatorname{sign}(\Lambda) Q^*$$

Where the entries of $|\Lambda|$ is $|\lambda_j|$ and the entries of $\operatorname{sign}(\Lambda)$ are $\operatorname{sign}(\lambda_j)$. We can show that, if Q is unitary, $\operatorname{sign}(\Lambda)Q$ is unitary, that means $Q|\Lambda|\operatorname{sign}(\Lambda)Q^*$ is a SVD of A □

Theorem 2.6.6: For $A \in \mathbb{C}^{m\times m}$, $|\det(A)| = \prod_{i=1}^{m} \sigma_i$

Proof: $|\det(A)| = |\det(U\Sigma V^*)| = |\det(U)\det(\Sigma)\det(V)| = |\det(\Sigma)|$ □

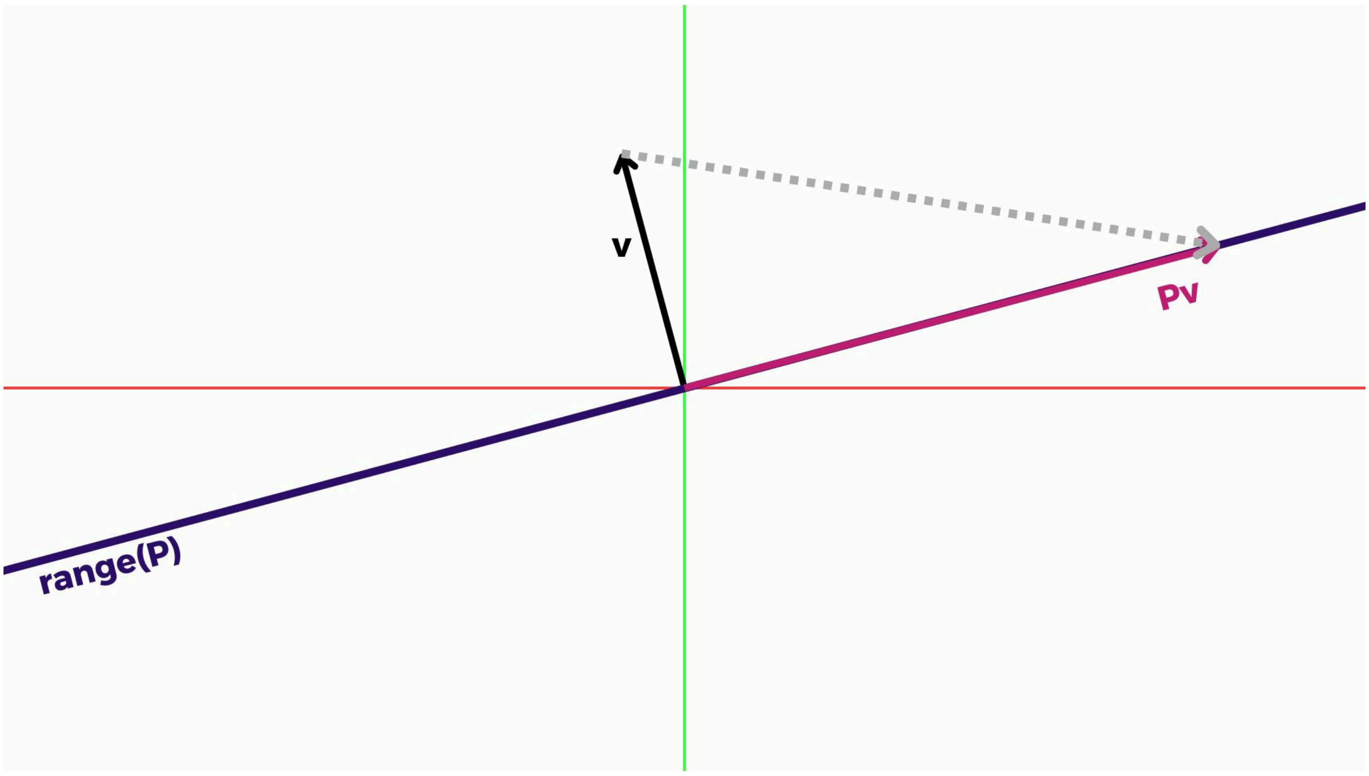
3. Lecture 6 - Projectors

$P \in C^{m\times n}$ is said to be a **Projector** if

$$P^2 = P$$

also called *idempotent*. You might confuse thinking only on orthogonal projections, that ones we get the vector and project it in a way to make a 90 degree angle in the projected space! But we are talking about **EVERY** projection, including non-orthogonal ones

Imagine we put a light on that vector, it will give a shadow somewhere, but you agree with me we can get that shadow somehow right? Let's see a 2D example



As you can see, the dashed vector tells us the direction where the light is projecting the shadow of v onto P. We can express this direction as $Pv - v$. Is important to remember that, if you're laying on the ground, you won't have a shadow right? Or even better, shadows doesn't have shadows! Translating this on our contexts:

Theorem 3.1: If $v \in C(P)$, then $Pv = v$

Proof: Every $v \in C(P)$ can be expressed as $v = Px$ for some x, that means $Pv = P^2x = Px = v$ □

Notice that, if we apply the projection onto the direction we had early

$$P(Pv - v) = P^2v - Pv = Pv - Pv = 0$$

That means $Pv - v \in \text{null}(P)$. Also notice we can rewrite the direction as

$$Pv - v = (P - I)v = -(I - P)v$$

Look what's even stranger!

$$(I - P)^2 = I - 2P + P^2 = I - P$$

That mens $I - P$ is also a projector! A projector that projects into the direction of projection of P

3.1. Complementary Projectors

If P is a projector, $I - P$ is its complementary projector

Theorem 3.1.1: $I - P$ projects onto $\text{null}(P)$ and P projects onto $\text{null}(I - P)$

Proof:

1. $C(I - P) \subseteq N(P)$ because $v - Pv \in N(P)$ and $C(I - P) \supseteq N(P)$ because, if $Pv = 0$, we can rewrite it as $(I - P)v = v$, that means $N(P) = C(I - P)$
2. If we rewrite the expression as $P = I - (I - P)$, then, using the same argument as before, we have $C(P) = N(I - P)$

□

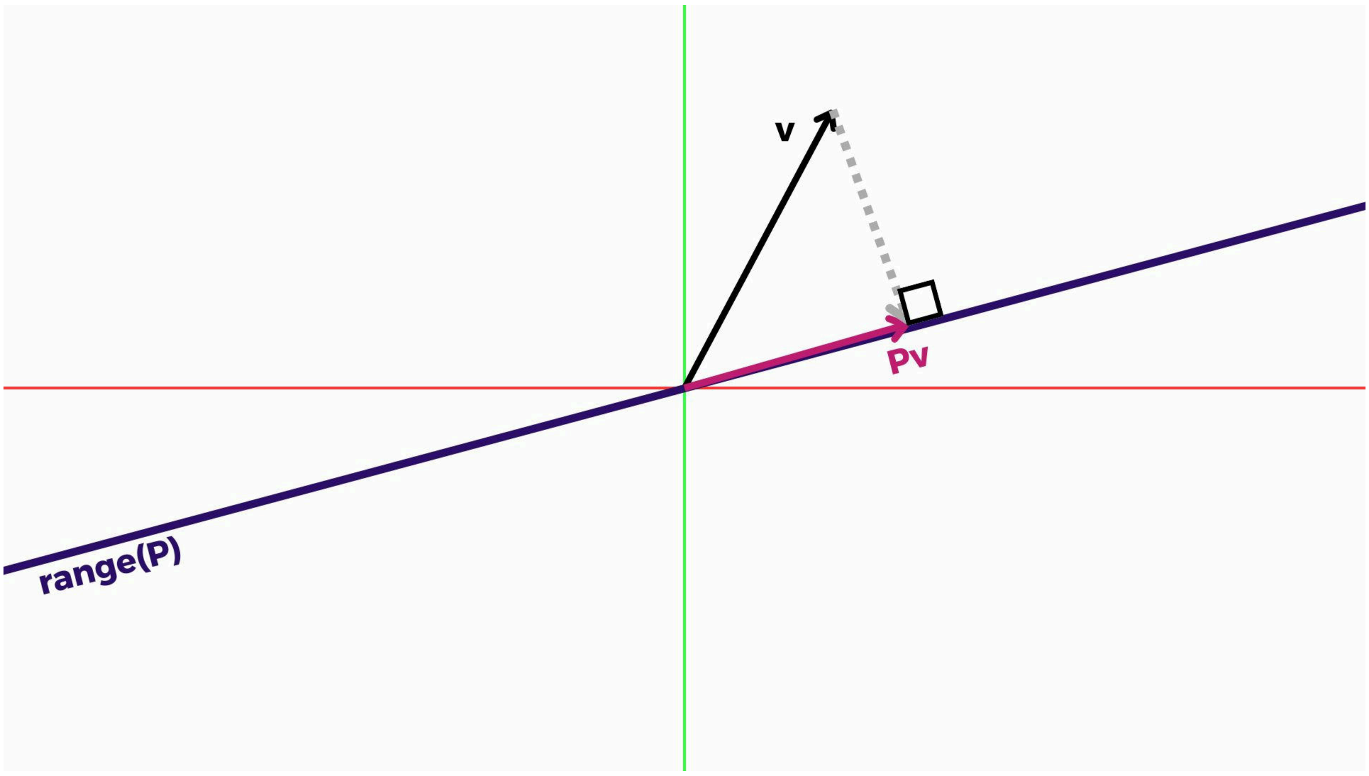
Theorem 3.1.2: $N(I - P) \cap N(P) = \{0\}$

Proof: $N(A) \cap C(A) = \{0\} \Rightarrow N(P) \cap C(P) = \{0\} \Leftrightarrow N(P) \cap N(I - P) = \{0\}$ □

That means, if we have a projector P in $\mathbb{C}^{m \times m}$, this projector separates $\mathbb{C}^m$ in to spaces $S_1 \wedge S_2$, in a way that $S_1 \cap S_2 = \{0\}$ and $S_1 + S_2 = \mathbb{C}^m$

3.2. Orthogonal Projectors

Finally! The projectors we hear all the time! They project a vector in a space making the direction form a 90 degree with the projection



That means $(Pv)^*(v - Pv) = 0$

Theorem 3.2.1: P is a orthogonal projector $\Leftrightarrow P = P^*$

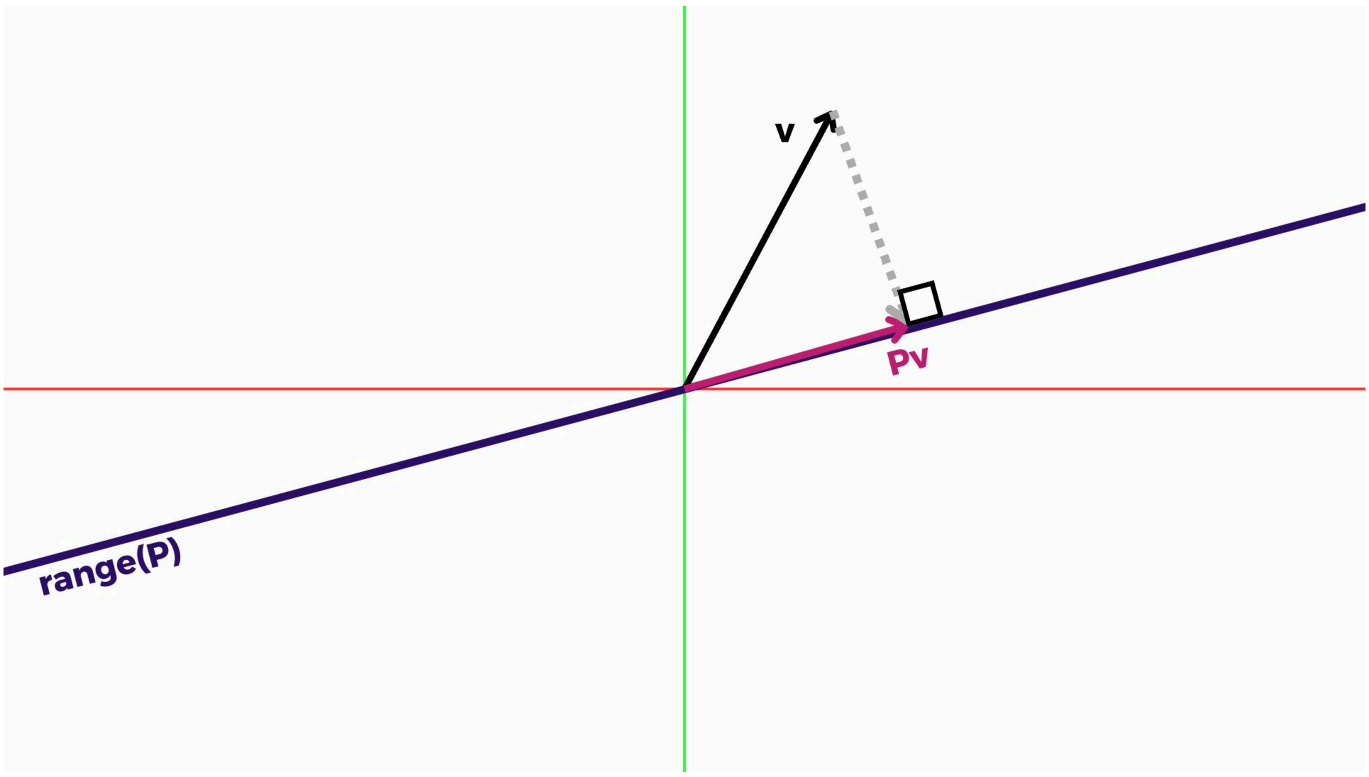
Proof:
1. $\Leftarrow$) Given $x, y \in \mathbb{C}^m$, then $x^* P^* (I - P) y = x^* (P^* - P^* P) y = x^* (P - P^2) y = x^* (P - P) y = 0$
2. $\Rightarrow$) Let $\{q_1, ..., q_m\}$ be a orthonormal base of $\mathbb{C}^m$ where $\{q_1, ..., q_n\}$ is base of S_1 and $\{q_{n+1}, ..., q_m\}$ is base of S_2. For $j \leq n$ we have $Pq_j = q_j$ and for $j > n$ we have $Pq_j = 0$, let Q be the matrix with columns $\{q_1, ..., q_m\}$ we have: $Q = \begin{pmatrix} | & & | \\ q_1 & \cdots & q_m \\ | & & | \end{pmatrix} \Leftrightarrow PQ = \begin{pmatrix} | & & | & | & \\ q_1 & \cdots & q_n & 0 & \cdots \\ | & & | & | & \end{pmatrix} \Leftrightarrow Q^* P Q =$ $\begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & 1 & & \\ & & & & 0 & \\ & & & & & \ddots \end{pmatrix}$, wich means we found a SVD decomposition for P:

$$P = Q \Sigma Q^* \Leftrightarrow P^* = Q \Sigma^* Q^* = Q \Sigma Q^* = P$$

□

3.3. Orthogonal projection onto a vector

Let's use the same example used previously



Let q be the vector that spans P, we know that $Pv = \alpha q$

$$(v - Pv)^* q = 0 = (v - \alpha q)^* q = 0$$

Now we look for the α that makes this equation valid

$$v^* q - \alpha q^* q = 0 \Leftrightarrow v^* q = \alpha q^* q \Leftrightarrow \alpha = \frac{v^* q}{q^* q}$$

$$Pv = \alpha q \Leftrightarrow Pv = \frac{v^* q}{q^* q} q \Leftrightarrow Pv = \frac{q^* v}{q^* q} q \Leftrightarrow Pv = q \frac{q^* v}{q^* q} \Leftrightarrow Pv = \frac{q q^*}{q^* q} v \Rightarrow P = \frac{q q^*}{q^* q}$$

3.4. Projection with orthonormal basis

We saw in Theorem 3.2.1's proof that some singular values of P are 0, so we could remove those lines of Σ and reduce it to I, also removing the columns and lines of Q, getting:

$$P = \hat{Q} \hat{Q}^*$$

Let $\{q_1, …, q_n\}$ be any set of orthonormal vectors in $\mathbb{C}^m$ and let them be the columns of $\hat{Q}$, we know that, for any vector $v \in \mathbb{C}^m$:

$$v = r + \sum_{i=1}^{n} q_i q_i^* v$$

What? when we saw that? Calm down, let me recap for you:

Theorem 3.4.1: Let $\{q_1, ..., q_n\}$ be any set of orthonormal vectors in $\mathbb{C}^m$, then any $v \in \mathbb{C}^m$ can be expressed as

$$v = r + \sum_{i=1}^{n} q_i q_i^* v$$

With r being another vector in C^m orthogonal to $\{q_1, ..., q_n\}$ and, $\rightarrow n = m \Rightarrow r = 0$ and the set of vectors chosen is a base for $\mathbb{C}^m$

Proof: You know that, given a base of $\mathbb{C}^m$, any vector can be expressed as a linear combination of those vectors. Imagine the cannon base (With some rotations, this logic can be expanded for other orthonormal bases), you can imagine that, if you project the vector you have onto any vector of the cannon base, you'll obtain a vector that, if you sum with another vector r, you'll obtain your original vector again! And we can continue this process until we do it with n vectors of the canon base, obtaining the original r that, if we sum all our projections, we get the original vector again, that is:

$$v = r + \sum_{i=1}^{n} q_i q_i^* v$$

□

Ok, knowing a vector can be expressed like this, we can see that the sum part is the same as doing:

$$\hat{Q}\hat{Q}^* v$$

That is, $\sum_{i=1}^{n} q_i q_i^* v$ is a projector onto $C\left(\hat{Q}\right)$

Theorem 3.4.2: The complement of an orthogonal projector is also an orthogonal projector

Proof:
1. $\left(I - \hat{Q}\hat{Q}^*\right)^2 = I - 2\hat{Q}\hat{Q}^* + \left(\hat{Q}\hat{Q}^*\right)^2 = I - 2\hat{Q}\hat{Q}^* + \hat{Q}\hat{Q}^* = I - \hat{Q}\hat{Q}^*$
2. $\left(I - \hat{Q}\hat{Q}^*\right)^* = I - \left(\hat{Q}\hat{Q}^*\right)^* = I - \hat{Q}\hat{Q}^*$

□

A special case is the rank-one orthogonal projector, this gets the vector and get the component of a single direction q, wich can be written:

$$P_q = qq^*$$

And its complement is the $(m - 1)$ rank matrix

$$P_{\perp q} = I - qq^*$$

This concept is valid for non-unitary vectors too:

$$P_a = \frac{aa^*}{a^* a}$$

$$P_{\perp a} = I - \frac{aa^*}{a^* a}$$

Just to clear the things. If we project a vector v onto a vector a, we're restraining v in the direction of the projection, so, if we project onto the complement of a, is like, we can express v as a linear combination of a and some other vectors, and then remove the part of a in this linear combination, having only the other vectors expressing a new vector

3.5. Projection on arbitrary basis

Given an arbitrary base $\{a_j\}$, we let the vectors of this base be the columns of A. Given v with $Pv = y \in C(A)$, that means $y - v \perp C(A)$, that means $a_j^*(y - v) = 0\ \forall j$. We know $y \in C(A)$, so let's write it as $Ax = y$, then we can rewrite $a_j^*(y - v) = 0\ \forall j$ as:

$$A^*(Ax - v) = 0 \Leftrightarrow A^*Ax - A^*v = 0 \Leftrightarrow A^*Ax = A^*v \Leftrightarrow x = (A^*A)^{-1}A^*v$$

$$Ax = A(A^*A)^{-1}A^*v \Leftrightarrow y = A(A^*A)^{-1}A^*v$$

$$\Rightarrow P = A(A^*A)^{-1}A^*$$

4. Lecture 7- QR Factorization

The scary one, the part where anyone knows anything! Let's calm down and see everything with patiently.

How does this factorization works? We want to express A as:

$$A = QR$$

With Q being an orthogonal matrix and R an upper triangular matrix. But why would I want to do such a thing? The main reason is resolving linear systems! Let's get the system $b = Ax$, we rewrite it as

$$b = QRx \Leftrightarrow Q^*b = Rx \Leftrightarrow c = Rx$$

We have a equivalent system, and this one is a **triangular** system, i.e, a trivial system for us and for a computer to solve!

4.1. The idea of the reduced factorization

Let $\{a_j\}$ denote the columns of $A \in \mathbb{C}^{m \times n},\ m \geq n$. In some applications, we are interested in the columns *spaces* of A, i.e, the sequential spaces spanned by the columns of A:

$$\text{span}\{a_1\} \subseteq \text{span}\{a_1, a_2\} \subseteq \text{span}\{a_1, a_2, a_3\} \subseteq \dots$$

For now, we'll assume A has full-rank n. First of all, we want to obtain a set of orthonormal vectors with the following property:

$$\text{span}\{a_1, \dots, a_j\} = \text{span}\{q_1, \dots, q_j\} \text{ with } j = 1, \dots, n$$

Well, I think a good idea for doing this, is vectors such that we could express a_j as a linear combination of $\{q_1, \dots, q_j\}$. So that means:

$$a_1 = r_{11}q_1$$

$$a_2 = r_{12}q_1 + r_{22}q_2$$

$$\vdots$$

$$a_n = r_{1n}q_1 + r_{2n}q_2 + \dots + r_{nn}q_n$$

We could express this equations as a matrix product!

$$\begin{pmatrix} | & | & & | \\ a_1 & a_2 & \dots & a_n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} | & | & & | \\ q_1 & q_2 & \dots & q_n \\ | & | & & | \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ & r_{22} & & \vdots \\ & & \ddots & \vdots \\ & & & r_{nn} \end{pmatrix}$$

So we have $A = \hat{Q}\hat{R}$, where $Q \in \mathbb{C}^{m \times n}$ and $R \in \mathbb{C}^{n \times n}$

4.2. Full QR Factorization

Goes a bit further. We know $\{q_1, \dots, q_n\}$ is a set of orthonormal vectors of $\mathbb{C}^m$, this means that we have $m - n$ more vectors orthonormal to the ones we had before, so we can create a base for $\mathbb{C}^m$, adding these vectors as columns of $\hat{Q}$, we have a orthogonal matrix Q. But what do we do for A stay the same? We can simply add lines of 0 bellow $\hat{R}$, creating $R \in \mathbb{C}^{m \times n}$ $(m \geq n)$, obtaining

$$A = QR$$

4.3. Gram-Schidt Orthogonalization

Oh no... the scary one... Let's go very calmly. We saw previously a way to calculate all q_j, let's remember it:

$$a_1 = r_{11} q_1$$

$$a_2 = r_{12} q_1 + r_{22} q_2$$

$$\vdots$$

$$a_n = r_{1n} q_1 + r_{2n} q_2 + \ldots + r_{nn} q_n$$

Well, this suggests an algorithm for calculating the next q_j, let's think, we have all a_j, and each q_j needs the vectors $\{q_1, \ldots, q_{j-1}\}$. Well, we can have some freedom here! Let's see what happens when we try to calculate q_j:

$$a_j = r_{1j} q_1 + \ldots + r_{jj} q_j$$

Let's isolate q_j:

$$q_j = \frac{a_j - r_{1j} q_1 - r_{2j} q_2 - \ldots - r_{2(j-1)} q_{j-1}}{r_{jj}}$$

Well, that suggest us that r_{jj} is the norm of the vector $a_j - \sum_{k=1}^{j-1} r_{ij} q_i$, but what is r_{ij}? Remember the decomposition on orthogonal factors? Yes, that one, $v = r + \sum_{i=1}^{n} q_i q_i^* v$. If we change v for a_j, we have almost the same thing we defined previously!

$$a_j - \sum_{k=1}^{j-1} r_{ij} q_i, \ \ v - \sum_{i=1}^{k} q_i q_i^* v$$

And you remember that r is orthogonal to span$\{q_1, \ldots, q_k\}$? That's exactly what q_j is! All this things I just said, suggest I can define r_{ij} as $q_i^* a_j$ $(i \neq j)$. And our algorithm is done! Let's recap it all here:

$$q_1 = \frac{a_1}{\|a_1\|_2}$$

$$q_2 = \frac{a_2 - q_1 q_1^* a_2}{\|a_2 - q_1 q_1^* a_2\|_2}$$

$$q_3 = \frac{a_3 - q_1 q_1^* a_3 - q_2 q_2^* a_3}{\|a_3 - q_1 q_1^* a_3 - q_2 q_2^* a_3\|_2}$$

$$\vdots$$

$$q_n = \frac{a_n - \sum_{i=1}^{n-1} q_i q_i^* a_n}{\|a_n - \sum_{i=1}^{n-1} q_i q_i^* a_n\|_2}$$

Writing in an algorithm form:

```
for $j = 1$ to $n$
    $v_j = a_j$
    for $i = 1$ to $j - 1$
        $r_{ij} = q_i^* a_j$
        $v_j = v_j - r_{ij} q_i$
    $r_{jj} = \|v_j\|_2$
    $q_j = \frac{v_j}{r_{jj}}$
```

4.4. Existence and Uniqueness

Theorem 4.4.1: Every $A \in \mathbb{C}^{m \times n}, \ (m \geq n)$ has a full QR factorization, hence also a reduced QR factorization

Proof: If $\operatorname{rank}(A) = n$, we can build the reduced factorization using Gram-Schmidt as we did before. The only problem here is if, at some point, $v_j = a_j - \sum_{k=1}^{j-1} q_k q_k^* a_j = 0$ thus cannot be normalized. If this happens, that means A hasn't full-rank, that means I can choose whatever orthogonal vector I want to continue the process. □

Theorem 4.4.2: Each $A \in \mathbb{C}^{m \times n} \ (m \geq n))$ of full rank has a unique reduced QR factorization $A = \hat{Q}\hat{R}$ with $r_{jj} > 0$

Proof: We know that, if A is full rank $\Rightarrow r_{jj} \neq 0$ and thus at each successive step j the formulas shown previously determine r_{ij} and q_j fully, the only problem is the signal of r_{jj}, once we say $r_{jj} > 0$, this problem is solved □

5. Lecture 8 - Gram-Schmidt Orthogonalization

We can describe the Gram-Schmidt algorithm using projectors, but why would we want this? Actually this is an introduction for another algorithm we'll see later. When we talk about algorithms, we want them to be stable, in a sense that if we put an input into the computer, it will return me an answer next to the right one (Computers won't solve continuous problems exactly), and the Gram-Schmidt process isn't stable (We'll talk about it on next lectures)

Remember I told you that, if you have a vector v and decompose it as

$$v = r + \sum_{k=1}^{n} q_k q_k^* v$$

The part $\sum_{k=1}^{n} q_k q_k^*$ is a projector that project onto the matrix $\hat{Q}\hat{Q}^*$ ($\hat{Q}$ has columns $\{q_1, ..., q_n\}$)? Well, it turns out that we can express the Gram-Schmidt algorithm steps the same way! Let's remember. At the j-th step, we have:

$$q_j = \frac{a_j - \sum_{i=1}^{j-1} q_i q_i^* a_j}{\|a_j - \sum_{i=1}^{j-1} q_i q_i^* a_j\|_2}$$

That means we can rewrite this as

$$q_j = \frac{\left(I - \hat{Q}_{j-1}\hat{Q}_{j-1}^*\right)a_j}{\|\left(I - \hat{Q}_{j-1}\hat{Q}_{j-1}^*\right)a_j\|_2}$$

Where $\hat{Q}_{j-1} = \begin{pmatrix} | & & | \\ q_1 & & q_{j-1} \\ | & & | \end{pmatrix}$. Let's define, for simplification, the projector P_j as:

$$P_j = I - \hat{Q}_{j-1}\hat{Q}_{j-1}^*$$

5.1. Modified Gram-Schmidt Algorithm

Using the definitions before, let's rewrite the Gram-Schmidt Algorithm.

For each value of j, the original Gram-Schmidt algorithm computes a single orthogonal projection of rank $m - (j - 1)$. I'm just translating to language using projectors, it does this:

$$v_j = P_j a_j = \left(I - \hat{Q}_{j-1} \hat{Q}_{j-1}^*\right) a_j$$

If you go back to what I said early, you get the original formula, I'm just changing that bunch of sums and vectors for a matrix product. The original algorithm makes this computation using a single projector, but the one we'll see does this by a sequence of $j - 1$ projectors of rank $m - 1$. By the definition of P_j we can state that:

Theorem 5.1.1:

$$P_j = P_{\perp q_{j-1}} ... P_{\perp q_2} P_{\perp q_1}$$

Proof: Remember that $P_{\perp q_k} = I - q_k q_k^*$, and what does it do? It projects a vector v onto the subspace orthogonal to $\{q_1, ..., q_{k-1}\}$, that is, removing the components $\{q_1, ..., q_{k-1}\}$ of v. The projector P_j does the exact same thing right? So, you can think that, if I project onto the complement of q_1, then onto the complement of q_2 and so on, at the j-th step, I'll have a vector that is orthogonal to the previous ones, that means, I remove all the previous components, leaving the projected vector as a linear combination of $\{q_k, ...\}$ □

Okay! If we define $P_1 = I$ we can rewrite $v_j = P_j a_j$ as:

$$v_j = P_{\perp q_{j-1}} ... P_{\perp q_2} P_{\perp q_1} a_j$$

The new modified algorithm is based on this new equation. We can get the same result stated on the previous version of the algorithm as:

$$v_j^{(1)} = a_j$$

$$v_j^{(2)} = P_{\perp q_1} v_j^{(1)}$$

$$v_j^{(3)} = P_{\perp q_2} v_j^{(2)}$$

$$v_j = v_j^{(j)} = P_{\perp q_{j-1}} v_j^{(j-1)}$$

We can rewrite it in form of pseudo-code:

```
for $i = 1$ to $n$
    $v_i = a_i$
for $i = 1$ to $n$
    $r_{ii} = \|v_i\|$
    $q_i = \frac{v_i}{r_{ii}}$
    for $j = i + 1$ to $n$
        $r_{ij} = q_i^* v_j$
        $v_j = v_j - r_{ij} q_i$
```

5.2. Gram-Schmidt as Triangular Orthogonalization

We can interpret each step of the Gram-Schmidt algorithm as a right-multiplication by a square upper-triangular matrix. Wait, what? Why? Get the R matrix:

$$\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & & \vdots \\ & & \ddots & \vdots \\ & & & r_{nn} \end{pmatrix}$$

You can separate it as:

$$\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & 1 & & \vdots \\ & & \ddots & \vdots \\ & & & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & \cdots & 0 \\ & r_{22} & & \vdots \\ & & \ddots & \vdots \\ & & & 1 \end{pmatrix} \cdots$$

So, we can easily see that, for the j-th matrix, the inverse of it is:

$$\begin{pmatrix} \ddots & & & \\ & 1 & & \\ & & r_{jj} & r_{j(j+1)} & \cdots \\ & & & \ddots & \end{pmatrix}^{-1} = \begin{pmatrix} \ddots & & & \\ & 1 & & \\ & & \frac{1}{r_{jj}} & -\frac{r_{j(j+1)}}{r_{jj}} & \cdots \\ & & & \ddots & \end{pmatrix}$$

That means we can understand the Gram-Schmidt algorithm as a orthogonalization by triangular matrices

$$AR_1R_2 \ldots R_n = \hat{Q}$$

$$R_1R_2 \ldots R_n = R^{-1}$$

6. Lecture 10 - Householder Triangularization

NOOOOO, HOUSEHOLDER NOOOOOO! Wait wait wait, let's get into it step-by-step! We saw on the last chapter, that the Gram-Schmidt algorithm can be written as a series of upper triangular matrices multiplication, right? Well, the householder triangularization's algorithm is really similar, but as the name suggests, instead of getting an orthogonal matrix at the end, we end up having an upper triangular matrix

$$Q_1Q_2 \ldots Q_nA = R$$

Is easy to see that $Q_n^* \ldots Q_2^* Q_1^*$ is a unitary matrix, that means $A = Q_n^* \ldots Q_2^* Q_1^* R$ is a full QR factorization of A

6.1. Triangularizing by Introducing Zeros

At the heart of the Householder algorithm, we have an idea of applying an orthogonal matrix that introduces zeros bellow the main diagonal! Like this (In this example, x means a non-zero entry, $\boldsymbol{x}$ means a entry that has changed since the last orthogonal application and nothing means 0)

$$\begin{pmatrix} x & x & x \\ x & x & x \\ x & x & x \\ x & x & x \\ x & x & x \end{pmatrix}_A \rightarrow Q_1A \rightarrow \begin{pmatrix} \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ & \boldsymbol{x} & \boldsymbol{x} \\ & \boldsymbol{x} & \boldsymbol{x} \\ & \boldsymbol{x} & \boldsymbol{x} \\ & \boldsymbol{x} & \boldsymbol{x} \end{pmatrix}_{Q_1A} \rightarrow Q_2Q_1A \rightarrow \begin{pmatrix} x & x & x \\ & \boldsymbol{x} & \boldsymbol{x} \\ & & \boldsymbol{x} \\ & & \boldsymbol{x} \\ & & \boldsymbol{x} \end{pmatrix}_{Q_2Q_1A} \rightarrow Q_3Q_2Q_1A \rightarrow \begin{pmatrix} x & x & x \\ & x & x \\ & & \boldsymbol{x} \\ & & \\ & & \end{pmatrix}_{Q_3Q_2Q_1A}$$

6.2. Householder Reflectors

Ok, we understood how the algorithm will work, but what kind of matrices can do such a thing? This is when the **Householder reflectors** enters the scene! Each Q_k will have this structure:

$$Q_k = \begin{pmatrix} I & 0 \\ 0 & F \end{pmatrix}$$

Where I is the identity matrix of size $k - 1 \times k - 1$ and F is an $m - k + 1 \times m - k + 1$ unitary matrix. But why this structure, where did we see that? Well, remember that, as we saw earlier, when we multiply

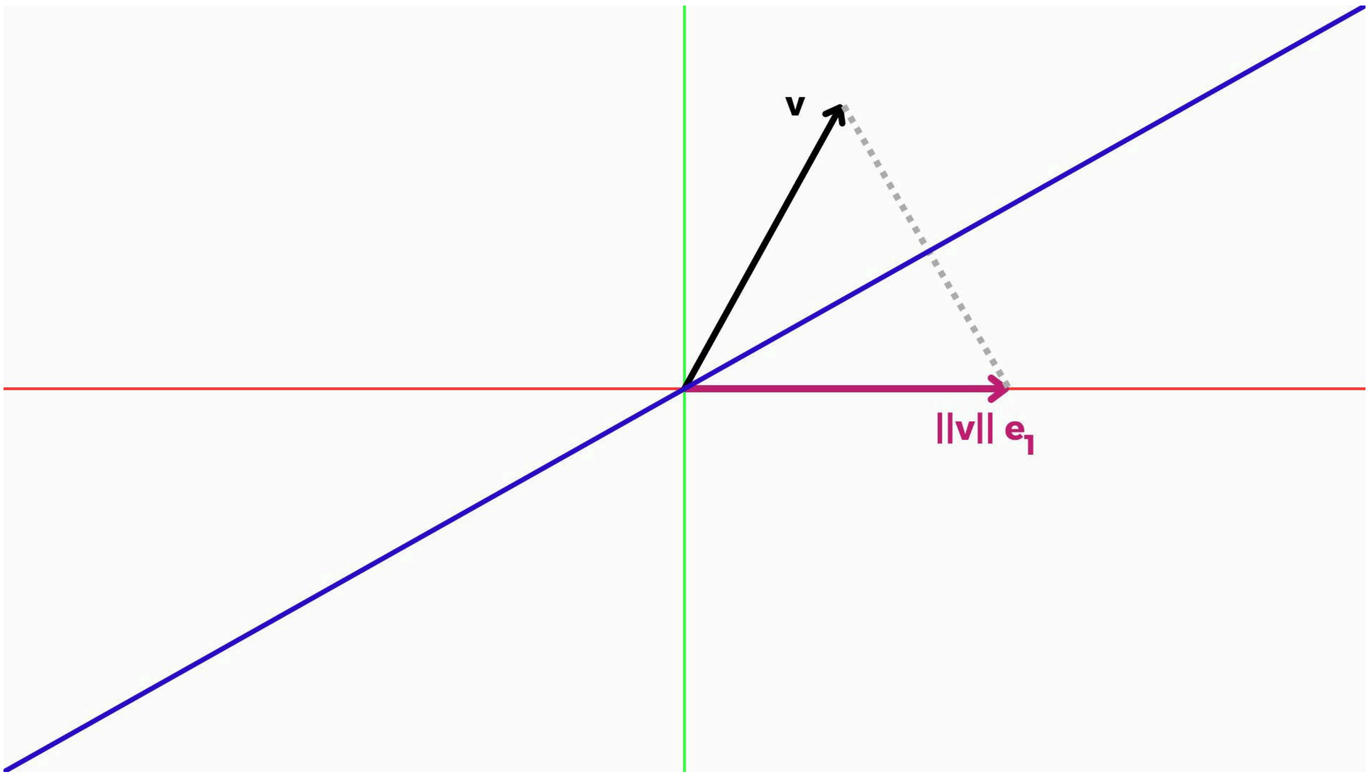
$Q_{k-1}...Q_1 A$ by Q_k, we want to maintain the lines 1 to $k-1$ untouched, so, for doing that, we make a $k-1 \times k-1$ block matrix of the identity to make this lines be untouched.

And why F is a unitary matrix? Well, we know that, because of the zeros bellow I and above F, the columns of F will be orthogonal to the columns of I independently of what I put there, but we want Q_k to be orthogonal, so, if the columns of I are already orthonormal, we just need the columns of F to be orthonormal too, that is, F to be orthogonal.

Ok, but now the most difficult part, we need that, when we multiply by F, it will introduce zeros bellow the k-th diagonal's entry and still be orthogonal. Let's make F be in $\mathbb{C}^{m-k+1 \times m-k+1}$ and affect vectors of $\mathbb{C}^{m-k+1}$ (We can see the lines bellow the k-th entries of vectors of this space), so we want the vectors' first entry to different then 0 and the rest be 0, knowing that orthogonal matrices are rotations in a space, we can make F do this:

$$x = \begin{pmatrix} x \\ x \\ x \\ \vdots \\ x \end{pmatrix} \to Fx = \begin{pmatrix} \|x\| \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \|x\| e_1$$

As shown in this figure:



Notice that, projecting v to get $\|v\|e_1$ won't give me a orthogonal projection, but I can project it on the blue line, that is the bisector of the angle between v and $\|v\|e_1$. This bisector forms a 90 degree angle with $\|v\|e_1 - v$. This is a 2D plane, so it is the bisector of the angle, but in a greater dimension space, it will be a hyperplane that is orthogonal to $\|v\|e_1 - v$. Let's define $w = \|v\|e_1 - v$, so , if H is the hyperplane orthogonal to w, we can project v onto H doing:

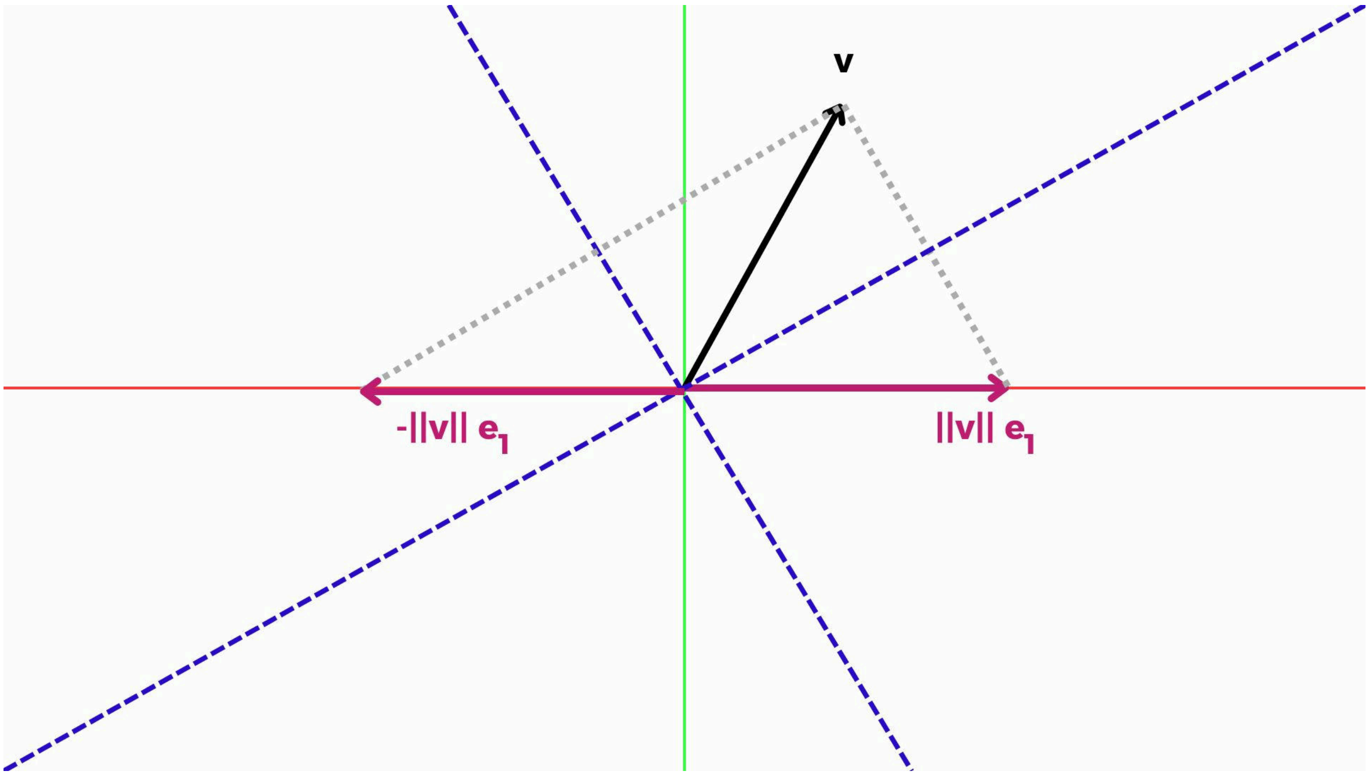
$$Pv = I - \frac{ww^*}{w^* w} v$$

But, as we can see, if we project v onto H, to get to $\|v\|e_1$, we need to go twite the distance we just went, so, the final equation for the Householder reflector is:

$$F = I - 2\frac{ww^*}{w^* w}$$

6.3. The Better of Two Reflectors

Actually, we can have lots of Householder reflectors, for example in the complex case, we can project v onto any vector $z\|v\|e_1$ with $|z| = 1$. In the real case, we have two alternatives:



So what should I pick? What vector is better for my algorithm? Everyone will be the same thing? Actually there is a best option you can pick! Mathematically, all of them is the same thing, but for numerical stability (Insensitivity for rounding errors), we'll pick the $z\|v\|e_1$ that is not too close to v, to achieve this, we'll project it onto $-\operatorname{sign}(v_1)\|v\|e_1$ where v_1 is the first entry of v, that means:

$$w = -\operatorname{sign}(v_1)\|v\|e_1 - v \lor w = \operatorname{sign}(v_1)\|v\|e_1 + v$$

And we can define that:

$$\operatorname{sign}(0) = 1$$

Just to clarify why we made this choice, imagine the angle between v and $\|v\|e_1$ is REALLY SMALL, this means that, when we do $\|v\|e_1 - v$, we are subtracting nearby quantities, depending on what quantities, this could lend us to imprecise calculations, lending to big errors

6.4. The Algorithm

Now we can rewrite it as an algorithm, but before that:

Definition 6.4.1: Given the matrix A, $A_{i:i',j:j'}$ is the $(i' - i + 1) \times (j' - j + 1)$ submatrix of A with the upper left corner element equal to $(A)_{ij}$ and the lower right corner element equal to $(A)_{i'j'}$. If the submatrix is a line or column vector we can write it as $A_{i,j:j'}$ or $A_{i:i',j}$

Given this definition, let's rewrite the algorithm

```
for $k = 1$ to $n$
    $x = A_{k:m,k}$
    $v_k = \operatorname{sign}(x_1)\|x\|e_1 + x$
    $v_k = \frac{v_k}{\|v_k\|}$
    $A_{k:m,k:n} = A_{k:m,k:n} - 2v_k\left(v_k^* A_{k:m,k:n}\right)$
```

6.5. Applying on forming Q

Notice that we didn't build the whole matrix Q on the algorithm, we just applied:

$$Q^* = Q_n \dots Q_1 \Leftrightarrow Q = Q_1 \dots Q_n$$

(There's no missing asterisks, because each Q_j is hermitian!)

We do this because build Q requires extra work, so we work directly with Q_j. For example, remember we can rewrite $b = Ax$ as $Q^* b = Rx$? Well, we can do this as in the previous algorithm:

```
**for** $k = 1$ **to** $n$
    $b_{k:m} = b_{k:m} - 2v_k(v_k^* b_{k:m})$
```

Notice we did the same process as we did with A, I just didn't explicit the parts where I defined v_k and normalized it.

7. Lecture 11 - Least Squares problems

What is the problem we are trying to look here? Well, we have a set of m equations with n variables, and we have more equations then variables $(m \geq n)$, and we want to find a solution to this system! But you agree with me, if we make the QR factorization of A, most of this equations will have no solution right? Because the entries bellow the n-th line of R will be equal to 0, so for the vector $Q^* b$ to have all entries equal to zero bellow the n-th line, just some specific choices of b will satisfy this!

$$A = QR \Rightarrow Ax = b \Leftrightarrow Rx = Q^* b$$

So what can we do to this system? Ignore it? Well, not at all! We know b will have a solution only if it lies in $C(A)$, that means, if $b \notin C(A)$, we have:

$$b - Ax = r \ (r \neq 0)$$

So we could find a way that r is the lowest as possible, so our new goal is to **minimize** $b - Ax$. To measure how small r is, we can choose any norm, but the 2-norm is a good choice, and have some good properties to work with.

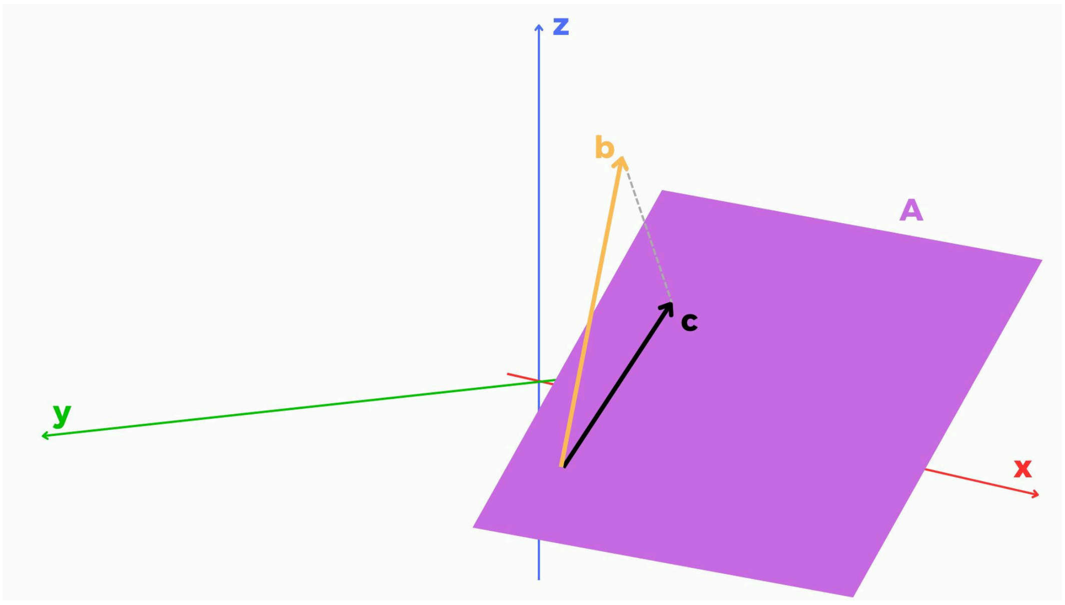
$$\text{Given } A \in \mathbb{C}^{m \times n}, m \geq n \text{ and } b \in \mathbb{C}^m$$

$$\text{find } x \in \mathbb{C}^n \text{ such that } \|b - Ax\|_2 \text{ is minimized}$$

So how do we solve it? Is there a fixed way to solve this problem? What do we do? Actually, there is a fixed way for resolving it, and it circles around **orthogonal projection**

7.1. Orthogonal Projections and the Normal Equations

It makes sense that the solution is obtained by a projection of b in $C(A)$, but what kind of projection? You can imagine a 3D plane, if you visualize A as a plane, it makes sense that the Ax that makes $\|b - Ax\|_2$, look to the picture at the next page:



Ok, it looks right, and we can think that intuitively, but is it mathematical right?

Theorem 7.1.1: Let $A \in \mathbb{C}^{m \times n}$ $(m \geq n)$, $b \in \mathbb{C}^m$.

$$x \text{ minimizes } \|b - Ax\|_2 \Leftrightarrow b - Ax \perp C(A)$$

Proof: First, define as P an orthogonal projector that projects onto $C(A)$ and $c = Pb$

Well, we know that $c \perp b - Ax$ (See the previous picture), then, because c is in $C(A)$, we can express $c = Ay$ for some $y \in \mathbb{C}^n \neq 0$. Let's write all down:

$$c^*(b - Ax) = 0 \Leftrightarrow y^* A^*(b - Ax) = 0$$

We know that $y \neq 0$, that means $A^*(b - Ax) = 0$

$$A^*(b - Ax) = 0 \Leftrightarrow A^*b - A^*Ax = 0 \Leftrightarrow A^*b = A^*Ax$$

We want x that satisfies this equation, if A^*A is invertible, then

$$x = (A^*A)^{-1} A^*b$$

And notice that if we apply A on x, it gives me the exact formula of the orthogonal projection of b onto A:

$$Ax = A(A^*A)^{-1} A^*b$$

□

The equation $A^*b = A^*Ax$ is known as "normal equation", and using it, we can get some different algorithms to compute the least square solution!

7.1.1. Standart

If A has full rank, this means A^*A is a square, hermitian and positive definite system of equations with dimension n. So we can make the Cholesky's Factorization of A^*A, obtaining R^*R where R is upper triangular, then we can make the reduction:

$$A^*b = A^*Ax \Leftrightarrow A^*b = R^*Rx$$

Then we can make the algorithm:

1. Form the matrix A^*A and A^*b
2. Compute the Cholesky Factorization of A^*A, R^*R
3. Solve the lower triangular system $R^*w = A^*b$ for w

4. Solve the upper triangular system $Rx = w$ for x

7.1.2. QR Factorization

A "modern" method uses the reduced QR factorization. Using the Householder's algorithm, we compute $A = \hat{Q}\hat{R}$ (Remember that $\hat{R}$ is squared and $\hat{Q}$ is $m \times n$). We can then rewrite the orthogonal projector $P = (A^*A)^{-1}A$ as $P = \hat{Q}\hat{Q}^*$, because $C(A) = C(Q)$.

$$\hat{Q}\hat{R}x = \hat{Q}\hat{Q}^*b \Leftrightarrow \hat{R}x = \hat{Q}^*b$$

And if R has inverse, we can multiply it by R^{-1} and have $A^+ = \hat{R}\hat{Q}^*$

1. Compute the reduced QR factorization $A = \hat{Q}\hat{R}$
2. Compute the vector $\hat{Q}^*b$
3. Solve the upper-triangular system $\hat{R}x = \hat{Q}^*b$ for x

7.1.3. S.V.D

If we get $A = \hat{U}\hat{\Sigma}V^*$ (Reduced S.V.D factorization), we can rewrite P as $P = \hat{U}\hat{U}^*$, because $\hat{U}$ is rectangular with orthonormal columns and $C(A) = C\left(\hat{U}\right)$, so project orthogonally onto $C(A)$ is the same as project orthogonally onto $C\left(\hat{U}\right)$, analogues of the QR method, we have:

$$\hat{U}\hat{\Sigma}V^*x = \hat{U}\hat{U}^*b \Leftrightarrow \hat{\Sigma}V^*x = \hat{U}^*b$$

Look that we can get a new formula for A^+, that is $A^+ = V\hat{\Sigma}^{-1}\hat{U}^*$

1. Compute the reduced S.V.D $A = \hat{U}\hat{\Sigma}V^*$
2. Compute the vector $\hat{U}^*b$
3. Solve the diagonal system $\hat{\Sigma}w = \hat{U}^*b$ for w
4. Set $x = Vw$

8. Lecture 12 - Conditioning and Condition Numbers

This is the part where the things start get confusing, so we'll need to get through it very calmly!

8.1. Condition of a Problem

First of all, what is a problem?

Definition 8.1.1: A *problem* $f : X \to Y$ is a function from a normed vector space X of **data** to a normed vector Y of **solutions**

Why did I define a problem like that? Because throughout the next lectures, we want to identify and measure what happens when I give a slightly different data to this problem, given that I have a certain data and pass it to the problem and it gives me a solution, the solution differs a lot? Just a bit? Is the same solution?

Definition 8.1.2: A **well-conditioned** problem is one that all small perturbations of $x \in X$ gives small differences on $f(x) \in Y$, that means if we make perturbations in x, the solutions of the problem won't change so much

Definition 8.1.3: A **ill-conditioned** problem is one that a small perturbation of $x \in X$ give me a large difference in the solutions $f(x) \in Y$, that is, changing the data, even if just slightly, will give me completely different answers to the problem

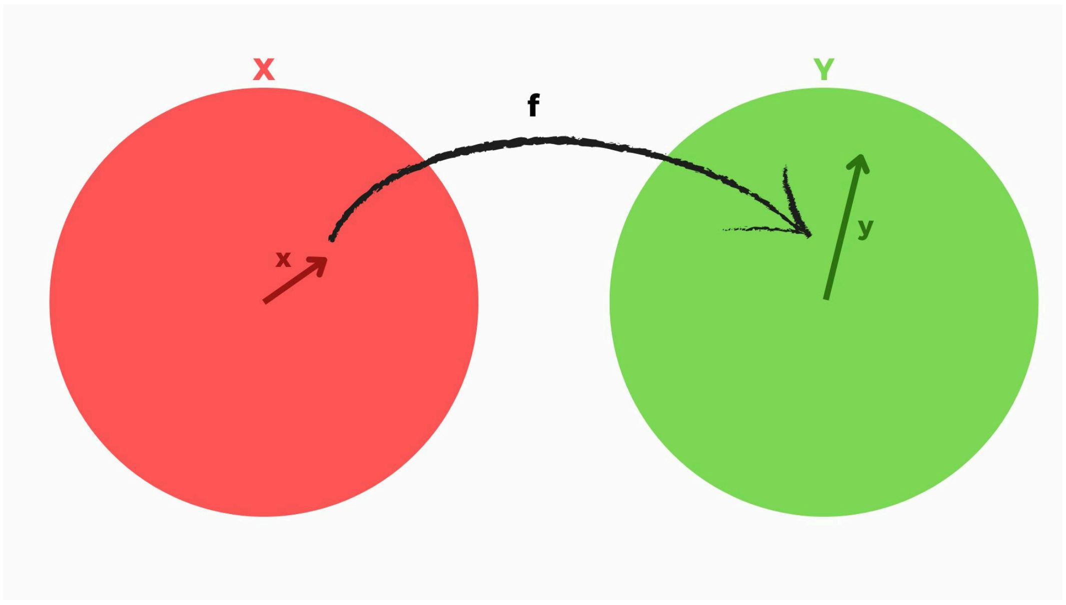
The meaning of **small** and **large** depends on what context I'm looking. And how can I measure this type of changes? How do I quantify those "**small**" and "**large**" changes? I'll define it here and show to you how we could visualize it

Definition 8.1.4 (Absolute Condition Number): Let Δx denote a small perturbation of x and write $\Delta f = f(x + \Delta x) - f(x)$. The **absolute condition number** of the problem f is defined as

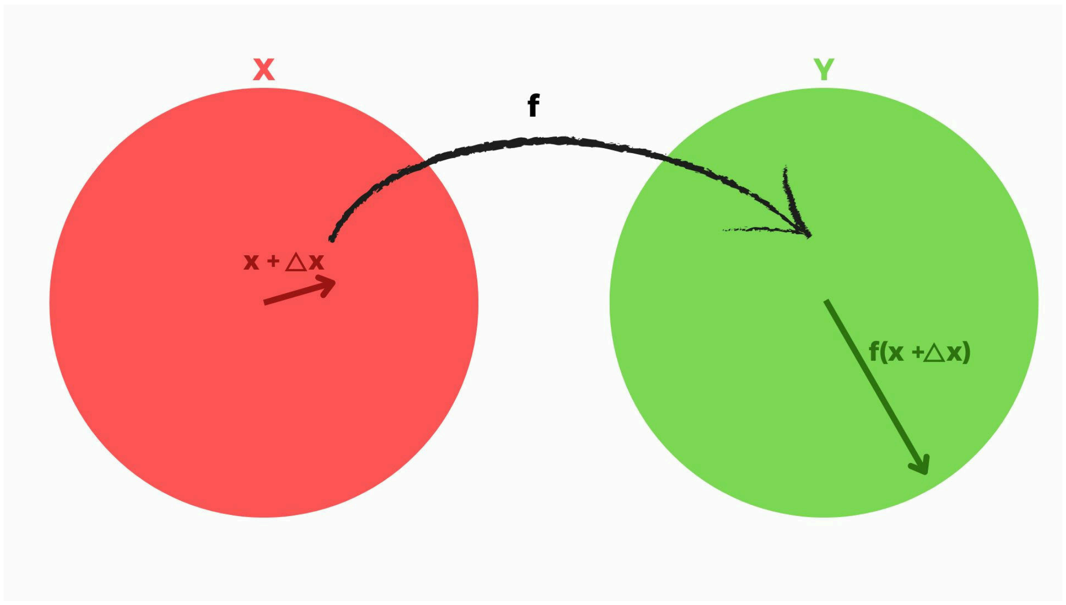
$$\hat{\kappa} = \lim_{\Delta \to 0} \sup_{\|\Delta x\| < \Delta} \left(\frac{\|\Delta f\|}{\|\Delta x\|} \right)$$

For better understanding, we can rewrite it as $\hat{\kappa} = \sup_{\Delta x} \frac{\|\Delta f\|}{\|\Delta x\|}$, that is, the supreme over all infinitesimal perturbations on x (Understanding that Δx and Δf are infinitesimal)

Dude, I didn't understand ANYTHING! Hold on, let's try to draw it here:



First, I have the normed spaces X and Y and how the problem f applies a transformation on x. Then, let's make just a tiny little adjustment on x, having Δx, this new vector is a infinitesimal perturbation on x, almost the same thing, now let's have a look on how f affects Δx:



JESUS, notice how a slightly perturbation on x changed the solution A LOT? That means this is a *ill-conditioned* problem, and the **larger** of this perturbations is the **absolute condition number** of f

We are talking about problems in **normed** spaces, therefore, in its heart, we could say problems are functions of subspaces of $\mathbb{C}^m$ to $\mathbb{C}^n$, that means if a problem f is *differentiable* (It is a function, so it could or couldn't be differentiable), it has a *Jacobian*. There is a statement that says, *"If $f : X \to Y$ is differentiable, then* $f(x + \Delta x) \approx$

$f(x) + J(x)\Delta x$ *when* $\Delta x \to 0$", well, we are working with $\Delta x \to 0$, so what happens if we substitute $f(x + \Delta x)$ for $f(x) + J(x)\Delta x$:

$$\hat{\kappa} = \lim_{\Delta \to 0} \sup_{\|\Delta x\| < \Delta} \left(\frac{\|\Delta f\|}{\|\Delta x\|} \right) = \lim_{\Delta \to 0} \sup_{\|\Delta x\| < \Delta} \left(\frac{\|f(x + \Delta x) - f(x)\|}{\|\Delta x\|} \right) = \lim_{\Delta \to 0} \sup_{\|\Delta x\| < \Delta} \left(\frac{\|f(x) + J(x)\Delta x - f(x)\|}{\|\Delta x\|} \right)$$

$$\lim_{\Delta \to 0} \sup_{\|\Delta x\| < \Delta} \left(\frac{\|J(x)\Delta x\|}{\|\Delta x\|} \right) \leq \lim_{\Delta \to 0} \sup_{\|\Delta x\| < \Delta} \left(\frac{\|J(x)\| \, \|\Delta x\|}{\|\Delta x\|} \right) = \|J(x)\|$$

That means the supreme value over all infinitesimal variations of x will be $\|J(x)\|$, that means

$$\hat{\kappa} = \|J(x)\|$$

Definition 8.1.5 (Relative Condition Number): Let Δx denote a small perturbation of x and write $\Delta f = f(x + \Delta x) - f(x)$. The **relative condition number** of the problem f is defined as

$$\kappa = \lim_{\Delta \to 0} \sup_{\|\Delta x\| \leq \Delta} \frac{\frac{\|\Delta f(x)\|}{\|f(x)\|}}{\frac{\|\Delta x\|}{\|x\|}}$$

Why we defined a "relative" condition number? Because I'm trying to measure it relatively.

Wow! You just repeated yourself... Hold on. Imagine that an engineer is making a 1000m rocket, and he measures an error of 1m, it is a little mistake right? But what if the rocket measures 2m? Is a COLOSSAL error right? That's the point, we are trying to measure the error caused by Δx based on the size of x and its solution $f(x)$

Well, remember we could represent the absolute condition number as

$$\hat{\kappa} = \|J(x)\|$$

we can do something similar with κ, let's see:

$$\kappa = \lim_{\Delta \to 0} \sup_{\|\Delta x\| \leq \Delta} \frac{\frac{\|\Delta f(x)\|}{\|f(x)\|}}{\frac{\|\Delta x\|}{\|x\|}} = \lim_{\Delta \to 0} \sup_{\|\Delta x\| \leq \Delta} \frac{\|\Delta f(x)\|}{\|f(x)\|} \frac{\|x\|}{\|\Delta x\|} = \lim_{\Delta \to 0} \sup_{\|\Delta x\| \leq \Delta} \frac{\|\Delta f(x)\|}{\|\Delta x\|} \frac{\|x\|}{\|f(x)\|} = \|J(x)\| \frac{\|x\|}{\|f(x)\|}$$

8.2. Conditioning Matrices and Vectors

Now, an important concept for stability and conditioning is how vector multiplication and matrices are conditioned, vector multiplications are *well-conditioned*? *ill-conditioned*? How matrices are conditioned? Let's see

8.2.1. Condition of Matrix-Vector Multiplication

Theorem 8.2.1.1: Given $A \in \mathbb{C}^{m \times n}$ fixed, the problem of calculating Ax with x in the normed space of data, the condition problem of the matrix is

$$\kappa = \|A\| \, \frac{\|x\|}{\|Ax\|}$$

If A is squared and invertible:

$$\kappa \leq \|A\| \, \|A^{-1}\|$$

Proof: Fix $A \in \mathbb{C}^{m \times n}$ and the problem of calculating Ax, with x being the data, that is, we'll calculate the condition of this problem based on perturbations of x, not A, A will be fixed the entire time.

$$\kappa = \sup_{\Delta x} \left(\frac{\|f(x + \Delta x) - f(x)\|}{\|f(x)\|} \frac{\|x\|}{\|\Delta x\|} \right) = \sup_{\Delta x} \left(\frac{\|A(x + \Delta x) - Ax\|}{\|Ax\|} \frac{\|x\|}{\|\Delta x\|} \right) = \sup_{\Delta x} \left(\frac{\|A\Delta x\|}{\|Ax\|} \frac{\|x\|}{\|\Delta x\|} \right)$$

$$\kappa \leq \sup_{\Delta x} \left(\frac{\|A\| \ \|\Delta x\|}{\|Ax\|} \frac{\|x\|}{\|\Delta x\|} \right) = \|A\| \frac{\|x\|}{\|Ax\|}$$

If A is squared and invertible, we can use the fact that $\frac{\|x\|}{\|Ax\|} \leq \|A^{-1}\|$ (We'll prove it later), to express κ as:

$$\kappa \leq \|A\| \ \|A^{-1}\| \lor \kappa = \alpha \ \|A\| \ \|A^{-1}\|$$

□

Corollary 8.2.1.1.1: Let $A \in \mathbb{C}^{m \times n}$ be non-singular and consider the equation $Ax = b$. The problem of computing b given x has condition number

$$\kappa = \|A\| \frac{\|x\|}{\|b\|}$$

Theorem 8.2.1.2:

$$\frac{\|x\|}{\|Ax\|} \leq \|A^{-1}\|$$

Proof: Write x as $x = A^{-1}(Ax)$, that means

$$\|x\| = \|A^{-1}Ax\| \leq \|A^{-1}\| \ \|Ax\| \Leftrightarrow \frac{\|x\|}{\|Ax\|} \leq \|A^{-1}\|$$

□

8.2.2. Condition Number of a Matrix

What? Why can we give a condition number to a matrix? Because they are **functions**! Why? Remember we can represent every **linear transformation** as a **matrix multiplication**? This implies, if a matrix A is in $\mathbb{C}^{m \times n}$, then it could be represented as $A : \mathbb{C}^n \to \mathbb{C}^m$! Now we understood why they can have a condition number, let's define it

Definition 8.2.2.1: Given $A \in \mathbb{C}^{m \times m}$ non-singular, the condition number of A, denoted as $\kappa(A)$ is:

$$\kappa(A) = \|A\| \ \|A^{-1}\|$$

If A is singular

$$\kappa(A) = \infty$$

If A is rectangular

$$\kappa(A) = \|A\| \ \|A^{+}\|, \ \left(A^{+} = (A^{*}A)^{-1}A\right)$$

8.2.3. System of Equations' Condition

Theorem 8.2.3.1: Given a system $Ax = b$, let us hold b fixed and consider the problem $A \mapsto x = A^{-1}b$, the condition number of this problem is:

$$\kappa(A)$$

Proof: If we perturb A and x, we do:

$$(A + \Delta A)(x + \Delta x) = b$$

$$\Leftrightarrow Ax + A(\Delta x) + (\Delta A)x + (\Delta A)(\Delta x) = b$$

$$\Leftrightarrow b + A(\Delta x) + (\Delta A)x + (\Delta A)(\Delta x) = b$$

$$\Leftrightarrow A(\Delta x) + (\Delta A)x + (\Delta A)(\Delta x) = 0$$

We know that $(\Delta A)(\Delta x)$ is doubly infinitesimal and both are going to 0, so we can discard it, and we obtain

$$A(\Delta x) + (\Delta A)x = 0 \Leftrightarrow \Delta x = -A^{-1}(\Delta A)x$$

This equation implies that

$$\|\Delta x\| \leq \|A^{-1}\| \, \|\Delta A\| \, \|x\| \Leftrightarrow \|\Delta x\| \, \|A\| \leq \|A^{-1}\| \, \|A\| \, \|\Delta A\| \, \|x\|$$

$$\frac{\|\Delta x\|}{\|x\|} \frac{\|A\|}{\|\Delta A\|} \leq \|A^{-1}\| \, \|A\|$$

If we make $\Delta x \to 0$ and $\Delta A \to 0$, we have

$$\kappa = \|A\| \, \|A^{-1}\| = \kappa(A)$$

□

9. Lecture 13 - Floating Point Arithmetic

When we are looking into algorithms and computers, we have a **really** big problem. Computers are discrete machines, that means, when we are talking about real numbers, they can't represent **all** of them, there is a finite amount of numbers they can represent depending on how these computers are built. Most of computers use a binary system to represent real numbers, but they could use other systems. There are two big problems on the representation of real numbers:

1. **Underflow & Overflow**: As I said, a computer can represent a finite number of real numbers, that means there is a maximum and a minimum in this set. If I try to represent a number bigger than this maximum, I'll have an **overflow** error, therefore, trying to represent a smaller number, I'll have an **underflow** error. Now days this isn't a really big problem, most of computers are capable of storing really large and tiny numbers, sufficient for the problems we're going to work with
2. **Gap**: When we try to represent real numbers, there is a problem, because between two real numbers, there are infinite other real numbers, that lead us to the **gap** problem, because, if the set of numbers the computer can represent finite, we can count them, if we can count them, we can get an infinity of other real numbers between them. The **gap** problem isn't a really a **PROBLEM**, but when we are making algorithms, we want them to pre as precise as possible, because a unstable algorithm can lead us to large rounding errors

9.1. Floating Point Set

The set of numbers a computer can understand and represent is called a **Floating Point set**. The book gives us a really hard definition, so I'll give you a simpler one and, later, understand the definition of the book:

Definition 9.1.1 (Simple definition): A **Floating Point Set** is defined as:

$$F = \{\pm 0, d_1 d_2 ... d_t * \beta^e / e \in \mathbb{Z}\}$$

Where $0, d_1 ... d_t$ is the **mantissa**, $t \in \mathbb{N}^*$ is the mantissa's **precision**, $\beta \in \mathbb{N}(\beta \geq 2)$ is the base, $d_j \left(0 \leq d_j < \beta\right)$ are the mantissa's digits and $e \in \mathbb{Z}$ is the **exponent**. In computers, **we** define a range for t, e and a specific base β

Let's go through everything. First of all, we defined everything, but what and why I defined those things?

9.1.1. Mantissa

Is the core of a number, a number in this set has only one mantissa, but a mantissa can be attached at various numbers, for example, we could represent 1 in the decimal system, as $0,1 * 10$ and we can represent 10 as $0,1 * 10^2$, that means the same mantissa can represent a lot of numbers. It's important to say that the mantissa is always wrote in base β, we'll see what this means now

9.1.2. Base and Precision

Definition 9.1.2.1: If we have $x \in \mathbb{R}$, write x in base β means choose coefficients $..., \alpha_{-1}, \alpha_0, \alpha_1, ...$ with $0 \leq \alpha_j < \beta$ and α_j are integers such that:

$$x = ... + \alpha_2\beta^2 + \alpha_2\beta^1 + \alpha_0\beta^0 + \alpha_{-1}\beta^{-1} + \alpha_{-2}\beta^{-2} + ...$$

Then we write x in base β as $...\alpha_2\alpha_1\alpha_0, \alpha_{-1}\alpha_{-2}..._{\beta}$, where each α_j is a digit

This is a way of representing every single real number, but the computer is limited to a certain amount of α_j, he can't store like 1 billion α_j, there's where the precision enters, it tells the computer how many digits the mantissa is able to represent!

Wait... but we saw the mantissa only stores numbers after the ",", so how do the set represents numbers grater then 1? That's where the **exponent** enters

9.1.3. Exponent

The exponent tells the order of our number, for example, with a mantissa $0, d_1d_2d_3$, we can represent 4 different numbers: $0, d_1d_2d_3$, d_1, d_2d_3, d_1d_2, d_3, $d_1d_2d_3$, they have exponent 0, 1, 2 and 3 respectively. But, if I have a mantissa $0, d_1...d_t$, why multiplying it by β^e moves the comma by e digits? (Just to do a parallel, that is exactly how our decimal system works, if you have $0, 1$, multiply it by 10 gives you 1)

Theorem 9.1.3.1: If you have x written in base β, multiply it by β^e will move the comma, in base β notation, e digits to the right

Proof:

$$x = ... + \alpha_2\beta^2 + \alpha_1\beta^1 + \alpha_0\beta^0 + \alpha_{-1}\beta^{-1} + \alpha_{-2}\beta^{-2} + ...$$

$$\Leftrightarrow \beta^e x = (... + \alpha_2\beta^2 + \alpha_1\beta^1 + \alpha_0\beta^0 + \alpha_{-1}\beta^{-1} + \alpha_{-2}\beta^{-2} + ...)\beta^e$$

$$\Leftrightarrow \beta^e x = ... + \alpha_2\beta^2\beta^e + \alpha_1\beta^1\beta^e + \alpha_0\beta^0\beta^e + \alpha_{-1}\beta^{-1}\beta^e + \alpha_{-2}\beta^{-2}\beta^e + ...$$

$$\Leftrightarrow \beta^e x = ... + \alpha_2\beta^{e+2} + \alpha_1\beta^{e+1} + \alpha_0\beta^e + \alpha_{-1}\beta^{e-1} + \alpha_{-2}\beta^{e-2} + ...$$

Notice how all α_j moved e digits to the left, that means the comma moves e digits to the right □

Lets go through an example to understand it better:

Example: I created a machine that represents my numbers by a floating point set F with decimal base ($\beta = 10$), precision 3 and $e \in [-5, 5]$, answer this questions:

1. What is the smallest positive number F can represent?

   Well, if F has precision 3, my mantissa is like this:

$$0, d_1d_2{d_3}_{\beta}$$

   So if we want the minimum, we need to make d_j as tiny as possible but still different from 0, that means we make the last digit (d_3) equal to 1 and the rest equal to 0, that means the smallest mantissa we can have is:

$$0,001$$

But we can still represent a larger number using the exponent, the smaller exponent we have is -5, that means the smallest positive number this set can represent is

$$0,001 * 10^{-5} = 0,00000001$$

2. What is the largest positive number F can represent?

   Following the same logic, the largest mantissa we can have is:

$$0,999$$

   Using the largest exponent possible, the largest number our set can represent is:

$$0,999 * 10^{5} = 99900$$

3. Is −3921 in the set?

   Let's test it out, if we decompose it:

$$-3921 = -0,3921 * 10^{4}$$

   The exponent is in the range $[-5, 5]$, but the mantissa is of precision 4, that means -3921 **IS NOT ON THE SET *F***

4. Is 738000000 in the set?

   Decomposing, we have: $738000000 = 0,738 * 10^{9}$, as we can see, the mantissa has the desired precision, but the exponent is greater than the given range, that means 738000000 **IS NOT ON THE SET *F***

Now that we understood this floating point set definition, let's see the book's definition:

Definition 9.1.3.1 (Book's definition): Being $F \subset \mathbb{R}$, we define it as:

$$F = \left\{ \pm \left(\frac{m}{\beta^{t}} \right) \beta^{e} \right\}$$

Where t, e and β means the same thing with the same restrictions as the previous definition

What is the difference and why the book defines it as this? There is just 1 big difference here: Why is he defining the **mantissa** as $\frac{m}{\beta^t}$? Remember when we had proven that, if we write x in base β and multiply x by β^e, the comma moves e digits to the right? The same applies if $e < 0$, but the comma goes to the left, and why am I saying that? Because, what the book doesn't tell us, is that we write m **IN BASE β**, and this division by β^t makes that we only get the first t digits of the number, meaning we only get the digits we desire to, JUST THIS (Yes, the book explains it poorly)

9.2. Numbers not in F

When we try to represent a number that is not in F, the computer can do 2 things:

1. **Round**: We get $t + 1$ digits of the number, and we check if the $(t + 1)$-th digit is greater or equal then $\left\lceil \frac{\beta}{2} \right\rceil$, if it is, we exclude the $(t + 1)$-th digit and sum 1 to the t-th digit. If the $(t + 1)$-th digit is less then $\left\lceil \frac{\beta}{2} \right\rceil$, then we just exclude the $(t + 1)$-th digit

   Example: Round 10324 knowing F has precision 4 and $e \in [-\infty, +\infty]$ and $\beta = 10$.

   Converting into the mantissa and exponent notation: $10324 = 0,10324 * 10^{5}$, we have 5 digits, so lets see the 5-th one. $4 \geq \left\lceil \frac{10}{2} \right\rceil \Leftrightarrow 4 \geq 5$? No, so the rounded number will be $0,1032 * 10^{5}$
2. **Truncate**: If the number's mantissa pass t digits, we remove all of the digits after the t-th one

Knowing that, we can finally understand what is $\varepsilon_{\text{machine}}$ is

9.3. Epsilon Machine

Let's see the first definition of the book, that is:

$$\varepsilon_{\text{machine}} = \frac{1}{2}\beta^{1-t}$$

But what does this mean? Why he defined like that? First of all, $\varepsilon_{\text{machine}}$ is the number that, if we make this operation in F, it will be valid

$$1 + \varepsilon_{\text{machine}} > 1$$

That means, if we sum 1 with a number less then $\varepsilon_{\text{machine}}$, even if by an infinitesimal difference, the number returned will be rounded or truncated to 1 in F. The book says this definition is the distance between 2 representable numbers in F, but why is that?

Theorem 9.3.1: The distance between 2 representable numbers in F is β^{1-t}

Proof: If we have x wrote in base β notation with precision t, we write it as:

$$x = 0, d_1 d_2 ... d_{t_\beta}$$

If we want to increment something in this number, but without making it go out of the possible precision and making the smallest increment possible, we can add 1 to d_t. So let's do it and see what happens, writing x:

$$x = d_1\beta^{-1} + d_2\beta^{-2} + ... + d_t\beta^{1-t}$$

If we sum 1 to d_t:

$$\alpha = d_1\beta^{-1} + d_2\beta^{-2} + ... + (d_t + 1)\beta^{1-t}$$

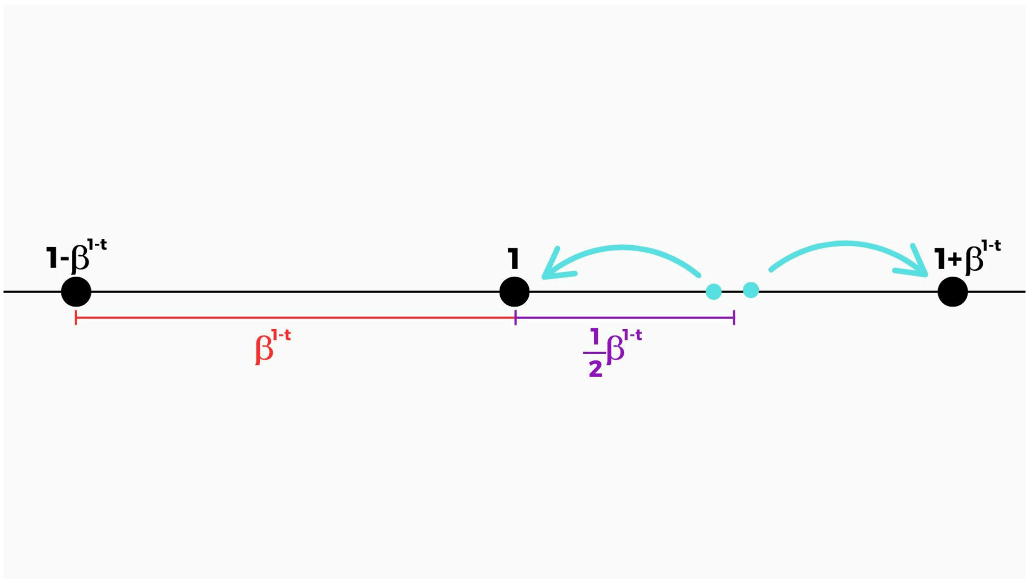
$$\Leftrightarrow \alpha = d_1\beta^{-1} + d_2\beta^{-2} + ... + d_t\beta^{1-t} + \beta^{1-t}$$

But notice how we can rewrite this as

$$\alpha = x + \beta^{1-t}$$

That means α (The next representable number), is $x + \boldsymbol{\beta^{1-t}}$, that is, the distance between them □

Now we can visualize it as a line, where we have the representable numbers and, if we try to represent a number that is in the gap of them, the computer will round it based on $\varepsilon_{\text{machine}}$



The cyan blue dots represent real numbers that can't be represented entirely by F, and the arrows shows where the computer rounds it. We'll change this definition later, and you'll understand why later.

The book shows us an inequality that every $\varepsilon_{\text{machine}}$ must hold, but that inequality can be rewritten:

Definition 9.3.1: Let F be a floating point set. $\text{fl} : \mathbb{R} \to F$ is a function that returns the rounded approximation of the input x in the set F

Theorem 9.3.2 (Floating Point Conversion): $\forall x \in \mathbb{R}$, there exists ε with $|\varepsilon| \leq \varepsilon_{\text{machine}}$ such that:

$$\text{fl}(x) = x(1 + \varepsilon)$$

What does this mean? It means, whenever I round a real number to fit it in F, the rounded number is equivalent to multiply x to $1 +$ a very tiny number, you can visualize it looking at the line representation of F I showed before

9.4. Floating Point Arithmetic

We need to make operations with numbers right? But we have the same problem, computers need to round because they can't understand all numbers in an interval, so how can we make the operations as precise as possible? We construct a computer based on this principle (Some computers can have more principles on its core, so some operations can be even preciser, but let's focus only on this one):

Definition 9.4.1 (Fundamental Axiom of Floating Point Arithmetic): Given that $+$, $-$, $\times$ and $\div$ represents operations in $\mathbb{R}$, consider $\oplus$, $\ominus$, $\otimes$ and $\oplus\!\!\!\!\div$ being operations in F. Let $\circledast$ define any of the previous operations in F, then we define a computer that makes the operation $x \circledast y$ as

$$x \circledast y = \text{fl}(x * y) = (x * y)$$

That means we make a computer following that $\forall x, y \in F$, there exists ε with $|\varepsilon| \leq \varepsilon_{\text{machine}}$ such that

$$x \circledast y = (x * y)(1 + \varepsilon)$$

In other words, every operation on F have an error with size **at most** $\varepsilon_{\text{machine}}$

9.5. More on Epsilon Machine

Now we can redefine $\varepsilon_{\text{machine}}$! But why? Well, we want to make it as low as possible, and sometimes increase the precision is not an option:

Definition 9.5.1: $\varepsilon_{\text{machine}}$ is the lowest value such that Theorem 9.3.2 and Definition 9.4.1 are valid

This implies that, for some computers, $\varepsilon_{\text{machine}}$ can be even lower then $\frac{1}{2}\beta^{1-t}$, wich is a **really** good thing!

10. Lecture 14 and 15 - Stability

When we're talking about **stability**, we are trying to see if an algorithm has fidelity on the computer! That means, if the roundings the computer make on the inputs and outputs won't change the result to something a lot different from the original. But first, we need to define mathematically what is an algorithm!

Definition 10.1: Let a problem $f : X \to Y$ and a computer with floating point system that satisfies Definition 9.4.1 be fixed. The algorithm of f, $\tilde{f} : X \to F^n \subset Y$ is a function that represent a series of steps and its implementations on the given computer with the objective of solving the problem f

Note: $F^n \subset Y$ just represents that I have a vector of numbers that can be represented by F and this vector is in Y

Well, we know that, if we pass $x \in X$ for this algorithm, the result $\tilde{f}(x)$ can be affected by rounding errors! In most cases, $\tilde{f}$ is not a continuous function, but the algorithm needs to approximate $f(x)$ as good as possible.

Let's make a definition for a **stable** algorithm too! First, let's define the **precision** of an algorithm

Definition 10.2 (Algorithm's Precision): An algorithm $\tilde{f}$ is precise if:

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = O(\varepsilon_{\text{machine}})$$

WHAT? WTF DOES $O(\varepsilon_{\text{machine}})$ EVEN MEEEEEANS??? Wait wait, I'll make a formal definition later, for now, you can understand that an algorithm is precise if the error between the algorithm's output and the original output doesn't pass $\varepsilon_{\text{machine}}$

In a bad-conditioned algorithms, the equality shown on the definition is too ambitious, because rounding errors are inevitable, in this kind of algorithms this rounding could cause big errors, overpassing the desired errors we wanted

Now we can define a **stable** algorithm

Definition 10.3 (Stable Algorithm): An algorithm $\tilde{f}$ for a problem f is stable if

$$\forall x \text{ is valid that } \frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = O(\varepsilon_{\text{machine}})$$

$$\text{for a } \tilde{x} \text{ with } \frac{\|\tilde{x} - x\|}{\|x\|} = O(\varepsilon_{\text{machine}})$$

Wait what? What does this definition means?

It means that, all similar data to my original data passed for my algorithm will return outputs very similar to the right solutions (This is shown with $\varepsilon_{\text{machine}}$, that is, the difference between the solutions given by the algorithm and the real solutions won't surpass $\varepsilon_{\text{machine}}$)

There is another type of **stability**, very powerful:

Definition 10.4 (Backwards Stability): An algorithm $\tilde{f}$ for f is **backwards stable** if:

$$\forall x \in X \text{ is valid that } \exists \tilde{x} \text{ with } \frac{\|\tilde{x} - x\|}{\|x\|} = O(\varepsilon_{\text{machine}}) \text{ such that } \tilde{f}(x) = f(\tilde{x})$$

That means if I pass the data into the algorithm, I can find a really small perturbation $\tilde{x}$ such that the solution of the problem if I pass this perturbation is the **same** as if I pass the original data into the algorithm!

Example: Given the data $x \in \mathbb{C}$, check if the algorithm $x \oplus x$ to compute the problem of summing two equal numbers (Solution is $2x$) is backwards stable:

We have that

$$f(x) = x + x \wedge \tilde{f}(x) = x \oplus x$$

That means

$$\tilde{f}(x) = (2x)(1 + \varepsilon)$$

Let's check if this algorithm is **stable**. First, define $\tilde{x} = x(1 + \varepsilon)$, we know that $\varepsilon = O(\varepsilon_{\text{machine}})$, let's check if the relative error between x and $\tilde{x}$ is $O(\varepsilon_{\text{machine}})$:

$$\frac{\|\tilde{x} - x\|}{\|x\|} = \frac{\|x(1 + \varepsilon) - x\|}{\|x\|} = \frac{\|x(1 + \varepsilon - 1)\|}{\|x\|} = |\varepsilon| = O(\varepsilon_{\text{machine}})$$

So $x(1 + \varepsilon)$ is a **valid** definition for $\tilde{x}$, making this clear, let's check if $\tilde{f}$ is stable

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = \frac{\|2x(1 + \varepsilon) - 2x(1 + \varepsilon)\|}{\|f(\tilde{x})\|} = 0 = O(\varepsilon_{\text{machine}})$$

That means $\tilde{f}$ is stable, but it is **backwards stable**? We need a definition of $\tilde{x}$ that stills hold the condition of x set in Definition 10.3. Let's check if our definition holds it, we already checked that the condition is valid, but it holds $f(\tilde{x}) = \tilde{f}(x)$?

$$\tilde{f}(x) = 2x(1 + \varepsilon)$$

$$f(\tilde{x}) = 2x(1 + \varepsilon)$$

That means this algorithm **IS** indeed **backwards stable**

10.1. Formal Definition of $O(\varepsilon_{\text{machine}})$

I'll write the definition here and explain what it means right after it

Definition 10.1.1: Given the functions $\varphi(t)$ and $\psi(t)$ the sentence

$$\varphi(t) = O(\psi(t))$$

means that $\exists C > 0$ such that $\forall t$ close enough to a known limit (e.g $t \to 0$, $t \to \infty$), is valid that:

$$\varphi(t) \leq C\psi(t)$$

What does this mean? It means that, if we write $\varphi(t) = O(\psi(t))$, and we know where t is going, there exists $C > 0$ such that the values of $\varphi(t)$ will never be greater then $C\psi(t)$. Most of the times I don't even care what is C, I only care about its existence!

Talking about how we are talking about $\varepsilon_{\text{machine}}$, the implicit limit here is $\varepsilon_{\text{machine}} \to 0$, and write that $\varphi(t) = O(\varepsilon_{\text{machine}})$ means that we have a constant that limits the error over an amount of $\varepsilon_{\text{machine}}$, that means, the error will never be greater then, for example, 3 times $\varepsilon_{\text{machine}}$, 2 and a half times $\varepsilon_{\text{machine}}$.

We can make a stronger (But more confusing) formal definition for O notation

Definition 10.1.2: Given $\varphi(s, t)$, we have that:

$$\varphi(s, t) = O(\psi(t)) \text{ uniformly in } s$$

makes sure that $\exists! C > 0$ such that:

$$\varphi(s, t) \leq C\psi(t)$$

and that is valid for any s I choose

Is a similar definition, I'm only adding a variable that I can choose and it will change nothing on it.

In real computers, $\varepsilon_{\text{machine}}$ is a fixed number, so when we're working with the implicit limit $\varepsilon_{\text{machine}} \to 0$, we're selecting an **ideal** family of computers!

10.2. Dependency on m and n

In practice, when we are talking about rounding errors, the stability of algorithms involving a matrix A depends not on A itself, but on m and n (Its dimensions). We can see that looking at the following problem:

Suppose I have an algorithm for resolving a non-singular system $m \times m$ $Ax = b$ for x and we make sure that the solution $\tilde{x}$ given by the algorithm satisfies

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\kappa(A)\varepsilon_{\text{machine}})$$

That means a constant C exists and satisfies

$$\|\tilde{x} - x\| \leq C\kappa(A)\varepsilon_{\text{machine}} \|x\|$$

This shows that, even C depending neither on A or b, it ends up depending on the dimensions of A because, if we change m or n, the data passed to the problem changes, that means we'll have a **new** problem because we're changing its domain and $k(A)$ will change too if we change its dimensions!

10.3. Independence of the Norm

You might have noticed that, when we are defining things in stability with norms, we denote $\| \cdot \|$ as *any* type of norm, but, if we choose a certain norm, the definition might not hold right? Like, it might hold for $\| \cdot \|_2$ but not for $\| \cdot \|_3$ right? Actually wrong! We can show that **precision**, **stability** and **backwards stability** hold its properties for **any** kind of norm! That means, when we're choosing a norm, we can choose one that makes the calculations easier!

Theorem 10.3.1: For problems f and its algorithms $\tilde{f}$ in finite dimensional normed spaces X and Y, the properties of **precision**, **stability** and **backwards stability** all hold or fail to hold independently on what norm I choose to make the analysis

Proof: If we prove that, if $\| \cdot \|$ and $\| \cdot \|'$ are two norms on X and Y, then $\exists c_1, c_2$ such that:

$$c_1\|x\| \leq \|x\|' \leq c_2\|x\|$$

then the theorem shown before is valid, because that shows:

1. If a sequence converges or is really small in a norm, it will be in all other norms too
2. Small errors in a norm will be small in all other norms too

But we need to prove the previous statement right? Let's do it! (The book only says that it is easy, lol)

First of all, lets reduce the problem to a unitary sphere of the norms, lets define:

$$S = \{x \in \mathbb{C}^n / \ \|x\| = 1\}$$

this set is **closed** because the norm is **continuous** and it's **limited** (A sphere, lol). Now let's define the function $f(x) = \|x\|'$, because $f(x)$ is continuous in $\mathbb{C}^n$ and S is closed and limited, we can find the **maximum** and **minimum** value of f in S, lets define:

1. $m = \min_{x \in S}\|x\|'$
2. $M = \max_{x \in S}\|x\|'$

$m > 0$ because $0 \notin S$. Now, let's try to generalize it in $\mathbb{C}^n$. If we want to generalize for all $x \neq 0 \in \mathbb{C}^n$, let's write:

$$x = \|x\|\left(\frac{x}{\|x\|}\right)$$

You can clearly see that $\frac{x}{\|x\|} \in S$ because $\|\frac{x}{\|x\|}\| = 1$. So, lets see what happens if we take $\|x\|'$:

$$\|x\|' = \| \ \|x\| \left(\frac{x}{\|x\|} \right) \ \|' = \|x\| \ \|\frac{x}{\|x\|}\|'$$

If you look it closely, $\frac{x}{\|x\|}$ is a vector in S, that means $\|\frac{x}{\|x\|}\|' \in [m, M]$ and, because of $\|x\|$ (Positive scalar number), we can see that

$$\|x\| \ \|\frac{x}{\|x\|}\|' \in [\|x\|m, \|x\|M]$$

We can rewrite this as

$$m\|x\| \leq \|x\|' \leq M\|x\|$$

That means this two constants exists and proves the theorem set before □

10.4. Floating Point Arithmetic Stability

Theorem 10.4.1: The operations $\oplus$, $\ominus$, $\otimes$ and $\oslash$ are **backwards stable**

Proof: Define $\circledast$ as any of the 4 operations shown before. Given a problem $f : X \to Y$ that is calculating $x_1 * x_2$, the algorithm $\tilde{f}$ for solving this problem is $\tilde{f}(x) = \text{fl}(x_1) \circledast \text{fl}(x_2)$ where $x = \binom{x_1}{x_2}$.

We have that:

$$\begin{aligned} \tilde{f}(x) &= \text{fl}(x_1) \circledast \text{fl}(x_2) \\ &= (\text{fl}(x_1) * \text{fl}(x_2))(1 + \varepsilon_3) \\ &= (x_1(1 + \varepsilon_1) * x_2(1 + \varepsilon_2))(1 + \varepsilon_3) \\ &= x_1(1 + \varepsilon_1)(1 + \varepsilon_3) * x_2(1 + \varepsilon_2)(1 + \varepsilon_3) \\ &= x_1(1 + \varepsilon_4) * x_2(1 + \varepsilon_5) \end{aligned}$$

Where $\varepsilon_4 = O(\varepsilon_{\text{machine}}) \land \varepsilon_5 = O(\varepsilon_{\text{machine}})$. We calculated $\tilde{f}(x)$, now let's see $f(\tilde{x})$. First, let's define:

$$\tilde{x} = \begin{pmatrix} x_1(1 + \varepsilon_4) \\ x_2(1 + \varepsilon_5) \end{pmatrix}$$

If we define $\tilde{x}$ like this, we can clearly see that

$$f(\tilde{x}) = x_1(1 + \varepsilon_4) + x_2(1 + \varepsilon_5) = \tilde{f}(x)$$

But the condition $\frac{\|\tilde{x}-x\|}{\|x\|} = O(\varepsilon_{\text{machine}})$ is satisfied?

$$\frac{\|\tilde{x} - x\|}{\|x\|} = \frac{\|\binom{x_1(1+\varepsilon_4)}{x_2(1+\varepsilon_5)} - \binom{x_1}{x_2}\|}{\|\binom{x_1}{x_2}\|} = \frac{\|\binom{x_1\varepsilon_4}{x_2\varepsilon_5}\|}{\|\binom{x_1}{x_2}\|}$$

Using the 1-norm

$$\frac{x_1\varepsilon_4 + x_2\varepsilon_5}{x_1 + x_2} = \frac{x_1 O(\varepsilon_{\text{machine}}) + x_2 O(\varepsilon_{\text{machine}})}{x_1 + x_2} = \frac{(x_1 + x_2) O(\varepsilon_{\text{mahcine}})}{x_1 + x_2} = O(\varepsilon_{\text{machine}})$$

This shows that $\oplus$, $\ominus$, $\otimes$ and $\oslash$ are **backwards stable** □

10.5. Precision of a Backwards Stable Algorithm

We say condition numbers before stability, let's try to associate both of them!

Theorem 10.5.1: Suppose a backwards stable algorithm $\tilde{f}$ is applied for a problem $f : X \to Y$ with condition number κ in a computer that satisfies Theorem 9.3.2 and Definition 9.4.1, so, the relative error satisfies:

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = O(\kappa(x)\varepsilon_{\text{machine}})$$

Proof: By definition, we have $\tilde{f}(x) = f(x + \delta x)$ with $\frac{\|\delta x\|}{\|x\|} = O(\varepsilon_{\text{machine}})$. Using Definition 8.1.5 (Relative Condition Number), we have that:

$$\kappa(x) = \lim_{\delta x \to 0} \left(\frac{\|f(x + \delta x) - f(x)\|}{\|f(x)\|}\right)\left(\frac{\|x\|}{\|\delta x\|}\right)$$

$$\kappa(x) = \lim_{\delta x \to 0} \left(\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} \frac{\|x\|}{\|\delta x\|}\right)$$

Using some formal definitions (Even I don't understand, so if I try to explain it here, I'll just lose time, lol), we can rewrite this as:

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} \leq (\kappa(x) + o(1))\frac{\|\delta x\|}{\|x\|}$$

Where $o(1) \to 0$ when $\varepsilon_{\text{machine}} \to 0v$ □

THE END.