31/03/20

1)

Calcul matriciel

Matrice de rotation centrée en (0,0)

ROTATION Θ

$$R = \begin{bmatrix} \cos(\Theta) & -\sin(\Theta) \\ \sin(\Theta) & \cos(\Theta) \end{bmatrix}$$

Rk: Appliquée à chaque point de la figure (a, b)

→ effectue une rotation de centre (0,0) et d'angle Θ
→ déterminant = 1
→ Matrice orthogonale: pas de déforma° de la forme (automorphisme orthogonal).

Une matrice orthogonale a par définiº la propriété suivante:
$R \cdot R^T = id$ $\quad R^T = R^{-1}$

↳ la matrice inverse de celle-ci est aussi sa transposée.

Rk: La matrice de rotation est une matrice orthogonale.

↳ En effet l'inverse de cette matrice est la matrice de rotation en (0,0) de angle −Θ.

Matrice symmetry horizontale

SYMMETRY AXIALE

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

$(a, b) \rightarrow (a, -b)$

⚠ Cela correspond à une symmetrie par rapport à l'axe $y = 0$.

Rk: Si on veut faire une symmétrie axiale par rapport à une droite qui passe par (0,0) il faut:
- faire une rotation pour mettre l'axe symmétrie à l'Horizontale.
- appliquée la symmétrie horizontale.
- faire la rotation inverse.

Cela revient au calcul matriciel suivant:

$$S = R_{-\alpha}^{-1} S_x R_{-\alpha} = R_\alpha S_x R_{-\alpha}$$

Rotation α · Sym. Horizontale · Rotation −α

Produit matriciel en python: @

Pb: translation n'est pas linéaire.
↳ donc on ne peut pas exprimer cette operation sous forme de produit matriciel. (pas non plus une operation isomérique).

TRANSLATION

isomérique: Une opération qui ne change pas la distance d'un point à un autre.

$T(x) = x + vt$ → simple addition

Lola Hack: on veut quand même pouvoir l'écrire sous forme de produit matriciel on va pour ça changer un peu la représentation de notre point.

$x=(x_1, x_2) \rightarrow x = (x_1, x_2, 1)$

la translation de x par le vecteur (v_1, v_2) devient :

$$T(x) = \begin{bmatrix} 1 & 0 & v_1 \\ 0 & 1 & v_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

Rk: And this time it is really linear!

La matrice inverse qui applique une translation $-v$ n'est pas la transposée de celle-ci du coup. T n'est donc pas isométrique d'un point de vue vectoriel alors que la translation l'est d'un point de vue géométrique.

↳ On appelle ça un isométrie d'un point de vue vectoriel, un automorphisme orthogonal.

→ automorphisme orthogonal : $\forall x, \|f(x)\| = x$
→ isométrie géométrique : $\forall a, b, \|f(a) - f(b)\| = \|a - b\|$

Matrice de passage (pour effectuer des changements de repère)

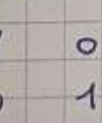
Il suffit pour ça d'exprimer notre nouvelle base dans notre ancienne

Si l'on veut effectuer par exemple une rotation dans notre nouveau repère.
Par exemple on veut créer une rotation d'un angle α :
- Retourner dans l'ancien repère. P^{-1}
- Appliquer la rotation de l'angle α. R
- Back to le nouveau repère. P

↳ On obtient donc l'operation suivante : $Q = PRP^{-1}$

for example in the base

$$B = \begin{bmatrix} \not{0}1 & 0 \\ 0 & 1 \end{bmatrix}$$



we want to change the base to

we express this in the previous Base:

$$D = \begin{bmatrix} 2 & -1,5 \\ 1 & 1 \end{bmatrix}$$

Passer d'un vecteur dans une base à une autre sans la translation
- $D(J-O)$
↳ calcul matriciel

→ Avec translation : on rajoute comme avant, une dimension à D:

$$T = \begin{bmatrix} 2 & -1,5 & 3 \\ 1 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

↳ Identité 3x3 puis P[2,-2] = D puis P[2,2] = A → 2 point vecteur de translation

Le monde 3D vu en 2D

Le but est de calculer l'image que génère une caméra en 2D en lui donnant :
- Une position dans l'espace. (C_x, C_y, C_z).
- Une direction dans laquelle regarder. (V_x, V_y, V_z)
- Une rotation C_0
- Une focale f. (~zoom)

Le but est de trouver la matrice P prenant tout cela en compte st : $\forall X$, point de mon espace 3D, on obtienne :

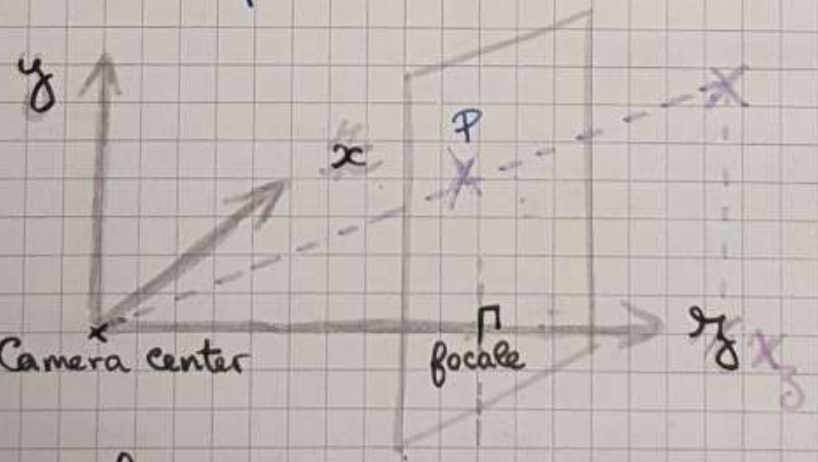
$$x = PX$$ with x sa position sur l'image.

X est un point dans la 3ème dimension ! MAIS ! Voulant pouvoir ajouter des translation, on doit rajouter une 4ème dimension - donc $X = (X_x, X_y, X_z, 1)$.

P est donc une matrice 3×4. since $x = PX$ (3D ; 3×4 ; $\times$ 4D)

Pour cela on va avoir besoin de 3 repères :
- Le repère du monde en 3D,
- Le repère de l'image en 2D,
- Le repère de la caméra axe z_3 → direction où elle regarde

la focale ici est un chiffre représentant la distance de la caméra sur l'axe z_3 où se trouve un plan orthogonal. C'est sur ce plan que l'on va projeter tous les points 3D.



We get $x = \frac{f}{X_{z_3}} X = f \frac{X}{X_{z_3}}$

Pour changer de repère pour passer du monde 3D au monde réel :

$P_{cam} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} X_{3D}$ on a donc : $P = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$F(f)$ la focale.

Pour translater la caméra en (c_x, c_y, c_z) du monde 3D :

$$T = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

On obtient $P = F(f) @ T(c)$

↳ j'applique ça à tout ce que je vois.

Pour rotate la caméra en 3D :

↳ 2 rotations possibles sur z et sur y (on considère que l'on ne tourne pas sur soi-même)
rotation horizontale ψ autour de z
rotation verticale φ autour de y.

$$D = \begin{bmatrix} \cos(\varphi) & 0 & \sin(\varphi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\varphi) & 0 & \cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\psi & -\sin(\psi) & 0 & 0 \\ \sin(\psi) & \cos(\psi) & 0 & \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation autour de y — rotation autour de z

⚠ prendre dans l'ordre de (y) (z).

↳ ne pas toucher à l'axe autour duquel on tourne. Commencer la matrice de rotation APRÈS l'axe exclu (d'où la matrice de y).

La dernière ~~ligne~~ colonne est pour les translation.

VECTEUR & VALEUR PROPRE

Vecteurs Propres

→ On remarque lorqu'on applique une application linéaire de nombreuses fois sur une forme, tous les points tendent vers les eigen ~~values~~ de l'application. La direction qui apparait est le eigen vector -> vectors.

On aura donc tous les point sur la direction imposée par le eigen vector le plus fort.

Si tu est un point dans la direction du vector propre le plus faible il va rester sur cette direction, sinon il va être attiré par le plus fort eigen vector.

→ Pour une matrice orthogonale, les eigen values son opposées, se are les vecteurs propres, les deux direction ayant la même "force" aucune n'attirera les points dans sa direction. Le phénomène de dilatation n'est pas visible ici.
(ex: rotation, symmétrie)

Pour une matrice diagonale, les eigenvalues sont sur la diagonale et les eigen vectores sont ceux de la base d'origine.

$\Rightarrow \exists P \mid A = P \Lambda P^{-1}$ with Λ cette fameuse matrice diag.

06/06/20 CAHA

2) Potentiellement, si on a une matrice correspondant à une application linéaire, On peut la remettre dans la base d'origine et elle sera diagonale.

P permet de changer de base. vec_propre @ np.diag(val_propre) @ lin.inv(vec_propre)

→ Pour inverser A (with $A = P \Lambda P^{-1}$) on a juste à inverser Λ which are all the value of the diagonal mais $\frac{1}{val}$ pour chaque valeur. Si une des eigen value est égal à 0, la matrice n'est pas diagonalisable.

Lorsque l'on a simplement un nuage de point et qu'on veut trouver une correlation entre eux, on peut essayer de trouver la matrice de covariance.

cov = np.cov(nuage)

Rk: quand une matrice est symmetrique, ses eigen values sont réelles et ses vecteurs propres sont orthogonaux.

en utilisant cov pour then trouver ses eigenvectors, on trouve le vecteur directeur

val, vec = lin.eig(cov)

Plus le premier vecteur est important par rapport au second, plus la correlation entre x et y est forte.

MATRICE DE COVARIANCE

Matrice de covariance. Relation linéaire entre les variables.

→ indique à quelle point les variables sont liées entre elles.
→ C'est une matrice de dim x dim où la covariance entre chaque éléments est "testée". Pour 2dim on a une matrice:

$$Cov(nuage2D) = \begin{bmatrix} cov(x,x) & cov(x,y) \\ cov(y,x) & cov(y,y) \end{bmatrix} \text{ with cov: } cov(x,y) = \frac{1}{N}\sum_{i=1}^{N}(x_i-\bar{x})(y_i-\bar{y})$$

→ plus la cov(.,.) est haute plus les variables sont liées entre elles. Une variance à 0 veut dire soit que les variables ne sont PAS liées, soit qu'elles ont une relation + complexe que linéaire.
→ deux variables non liées ont une covariance à 0.

$$(PDP^{-1})^n = PD^nP^{-1}$$

% timeit command
↳ Only on jupyter

(TP vecteur et valeur propre)

Si on veut la pente principale on peut simplement regarder le premier eigen vector (le plus fort).

Si on veut simplement la pente: $\frac{vect[1]}{vect[0]}$ (ici y et x). Si on veut enlever une composante: $y = ax^2 - bx + c \rightarrow y - ax^2$ et voilà c'est linéaire.

SYSTÈME D'ÉQUATION

Normalement pour résoudre un système de n inconnue n équations on inverse A :

$$AX = B \rightarrow X = A^{-1}B.$$

Des fois pas le meilleur moyen car A est trop grande pour être inversée de manière efficace.

3 méthodes de résolution
- ↳ Pivot de Gauss (rendre la matrice triangulaire)
- ↳ Décomposition LU
- ↳ Méthode de Gauss-Jordan (rendre la matrice diagonale)

PIVOT DE GAUSS

On a $AX = B$, on veut rendre A triangulaire : on peut faire disparaître (mettre à 0) les premiers terme d'une ligne en faisant $E_1 A$ (ne pas oublier $E_1 B$) with E_1 :

$$E_1 = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ -\frac{a_{21}}{a_{11}} & 1 & 0 & \cdot & 0 \\ -\frac{a_{31}}{a_{11}} & 0 & 1 & \cdot & 0 \\ \dots & 0 & 0 & \cdot & 0 \\ -\frac{a_{n1}}{a_{11}} & 0 & 0 & \cdot & 1 \end{bmatrix} A$$

Il faut que les termes $\frac{-a_{21}}{a_{11}}$ soient sous la diagonale

ex:

$$E_2 = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ 0 & -\frac{a_{32}}{a_{22}} & \dots & 0 \\ \dots & \dots & \dots & 0 \\ 0 & -\frac{a_{n2}}{a_{22}} & \dots & 1 \end{bmatrix}$$

Une fois que A est triangulaire, on peut résoudre en remontant progressivement de la dernière à la première ligne.

```
for i in range (len(A)-1)[::-1]:
    res[i] = (b[i] - A[i,i+1:] @ res[i+1:])/ A[i,i]
```

→ $\forall i \in [\![0, N]\!]$
- → On créé E_i en faisant identity (N) et ajoutant les coefs $\frac{-a_{ki}}{a_{ii}}$ dans la colonne i en dessous de la diagonale.
- → On multiplie E_i avec A apartir de la diagonale puisque le début ne changera pas la diagonale.
- → On multiplie B et E_i donc E_i @ B (on peut faire terme à terme pour gagner du temps.
- → On a donc A et B prêt.

→ On créé la matrice res vide avec simplement la dernière ligne à B[-1] / A[N-1, N-1] (comme si on avait $AX = B \rightarrow X = B/A$).

→ On part de la fin et on remonte pour remplir res de plus avant-dernière ligne jusqu'à la première since la dernière est déjà completed.

→ res est calculé en prenant B[i] et en lui soustrayant les lignes d'après, les coefficiant déjà trouvés. Et on divise par A[i,i].

→ You got it ☺.

```
for i in range (len(A)-1): # not the last line
    E = np.diag(np.array([1.,]*len(A)),dtype=A.dtype)
    coefs = -A[i+1:, i] / A[i,i]
    A[i:,i:] = E[i:, i:] @ A[i:, i:]
    B[i+1:] += coefs * B[i]
res = np.zeros(len(B),dtype=B.dtype) and res[-1] = B[-1]/A[-1,-1]
for i in range(len(A)-1)[::-1]:
    res[i] = (B[i]- A[i, i+1:] @ res[i+1:])/A[i,i]
```

Décomposition Lower Upper

Les calculs matriciels avec des matrices triangulaires sont beaucoup moins coûteux donc si l'on doit réutiliser la matrice A autant qu'on ait pas à la remodifier à chaque système.

On a $AX = B$ pivot de gauss $\underbrace{EA}X = EB$

matrice triangulaire supérieure.

Décomposition LU consiste a ne changer que la partie gauche de l'équation on a $E^{-1}EAX = B$ avec E^{-1} une matrice triangulaire inférieure.

→ Il est simple de faire E^{-1} et $E@A$ en même temps.

On aura plus qu'à résoudre $Ly = B$ puis $UX = y$.

E^{-1} c'est E mais avec des $+\frac{A[k,i]}{A[i,i]}$ à la place des $-\frac{A[k,i]}{A[i,i]}$.

Une fois qu'on a L et U on peut tout calculer en $\Theta(n^2)$ au lieu de $\Theta(\frac{n^3}{3})$ usuel. Le coup de la création des matrices L et U est rentabiliser si l'on utilise plusieurs fois A.

GAUSS-JORDAN

Gauss - Jordan

Le principe est de rendre la matrice diagonale plutôt que triangulaire ce qui rend les calculs encore + simple, le problème et que cette technique est un peu plus lente que les autres puisqu'il faut changer tout E et A à chaque fois - Ici on a E :

$$E = \begin{bmatrix} 1 & 0 & -\frac{a_{13}}{a_{33}} & 0 & 0 \\ 0 & 1 & -\frac{a_{23}}{a_{33}} & 0 & 0 \\ 0 & 0 & 1 & 0 \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \frac{-a_{n3}}{a_{33}} & 0 & 1 \end{bmatrix}$$

```
for i in range(len(A)):
    d1[i,i] = np.diag([1.,]*len(A))
    d1[:,i] = -A[:,i] / A[i,i]
    A = d1@A
    B = d1@B
return B / np.diag(A)
```

Pour éviter les erreurs d'arrondis, on peut réarranger les valeurs de la matrice pour ne pas avoir les petites valeurs sur la diagonale. En fait on veut la plus grosse valeur sur la diagonale comme ça on ne divise pas par une minuscule valeur.

CONDITIONNEMENT D'UNE MATRICE

Conditionnement d'une matrice → permet d'anticiper l'ordre de grandeur de l'erreur. Valeur max de l'erreur.

Si l'on a $A(x + \delta x) = b + \delta b$ alors on a $\|A^{-1}\|\|A\| = cond(A)$ le conditionnement de A.

→ `lin.norm(lin.inv(A)) * lin.norm(A)`

On va essayer de chercher une matrice qui permet de changer A et donc de minimiser l'erreur maximum possible lors du solve.

Une matrice conditionnée va donner potentiellement de grosses erreurs lors du solve.

Propriétés du conditionnement

$\text{cond}(A) \geq 1$ car $Id = AA^{-1}$ et donc $1 \leq \|A\|\|A^{-1}\|$

$\text{cond}(A) = \text{cond}(A^{-1})$ par définition du conditionnement

$\text{cond}_2(A) = \frac{\max_i |\lambda_i|}{\min_i |\lambda_i|}$ si la matrice est réelle où le 2 indique la norme 2 et λ_i sont les valeurs propres de A.

si A est unitaire ou orthogonale alors $\text{cond}_2(A) = 1$
↳ (une rotation ou symétrie ne va pas agrandir l'erreur. le conditionnement n'est pas modifié par transformation unitaire). (same pour symétrique).

```
vp = lin.eigvals(A)
vp.max() / vp.min()
```

Préconditionnement

Si le conditionnement n'est pas modifié par une transformation unitaire, il l'est par d'autres transformations. Ainsi
si $\text{cond}(A) \neq 1$, $\forall A, \exists B$ appelée matrice de préconditionnement such that $\text{cond}(BA) < \text{cond}(A)$

↳ Aussi au lieu de résoudre $AX = B$ on résoud $BCAX = CB$

Le plus dure est d'actually trouver cette matrice C.

MÉTHODES ITÉRATIVES

Méthodes itératives

Pour des énormes matrices, beaucoup trop de calculs à faire. on va alors prendre une nouvelle technique qui consiste à prendre une valeur plus ou moins random de départ et la faire converger petit à petit vers la solution. On arrête le calcul lorsqu'on estime qu'on est à une distance choisie de la solution (qu'on appelle erreur) plutôt que d'atteindre une solution exacte.

En calcul avec des matrices on peut faire:

```
while erreur > seuil:
    x = B @ x + c
    erreur = f(x)
```

On a donc un simple calcul matriciel ce qui ramène à du $\mathcal{O}(n^2)$!

Méthode de Jacobi

JACOBI

→ on découpe A en deux matrice M et N, M ne contenant que la diagonale et N le reste tel que

$$N = M - A \quad \Rightarrow A = M - N$$

formule itérative: $Mx^{k+1} = Nx^k + b$ (iteration $k+1$)
$\rightarrow x^{k+1} = M^{-1}(Nx^k + b)$

09/06/20 CAMA

3) La méthode de Jacobi ne fonctionnera pas pour toutes les matrices. Par exemple elle ne fonctionne pas si A possède des 0 sur la diagonale.

```
A = np.random.randint(10, size=(4,4)) # construit A
b = A.sum(axis=1) # makes sure toutes les solutions sont 1

M = np.diag(A)
N = np.diag(M) - A

X = np.random.random(4)
for i in range(20):
    x = (N @ x + b) / M # Jacobi Rare.
```

Si on reprend l'équation que nous avions au début :
$x = B@x + c$
On a dans notre implementation $B = M^{-1}N$ et $c = M^{-1}b$

Pour que cette méthode itérative fonctionne et converge il faut :
- $\rho(B) < 1$ avec ρ le rayon spectral (+ grande eigen valeur en absolute)
ou que :
$\|B\| < 1$ où la norme d'une matrice est subordonnée à une norme vectorielle :

$$\|B\| = \sup_v \frac{\|Bv\|}{\|v\|} = \sup_{v \text{ st } \|v\|=1} \|Bv\| = \sup_{v \text{ st } \|v\|\leq 1} \|Bv\|$$

On sait dans tous les cas que la méthode de Jacobi marchera si A est a diagonale dominante à savoir que chaque élément de la diagonale est + grand que tous les autres de la ligne ou colonne
Converge aussi si A symétrique, réelle et définie positive
($\forall x$, $x^T A x > 0$).

JACOBI & INERTIE

L'inertie va permettre de faire marcher la méthode de Jacobi sur plus de matrice. Elle permet de ralentir la convergence et les itérations assez pour permettre à + de matrice d'aller dans le droit chemin.
La formule de Jacobi devient alors :

$$x^{k+1} = M^{-1}(Nx^k + b) \quad \Rightarrow \quad x^{k+1} = \omega M^{-1}(Nx^k + b) + (1-\omega)x^k \quad 0 < \omega \leq 1$$

with $\omega = 1$ la méthode de Jacobi classique.
$\omega a + (1-\omega)b \rightarrow$ on obtient une valeur dans $[b, a]$

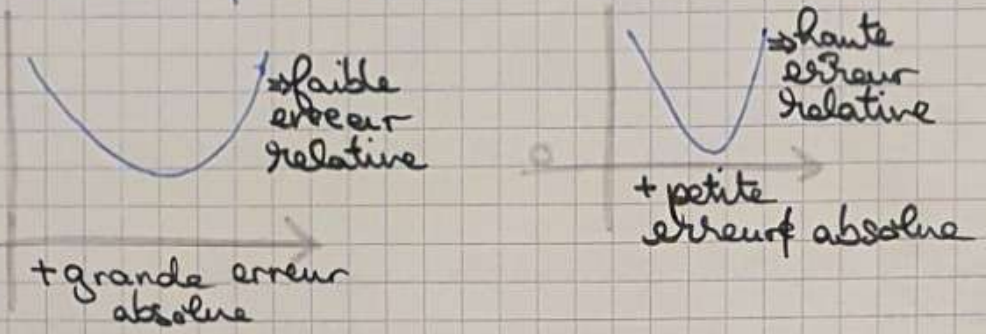
Cette méthode s'appelle la sur-relaxation :

Ça converge un moment puis diverge au lieu de diverger direct

↳ On peut étudier cela en regardant l'évolution de l'erreur absolue (par rapport à la véritable réponse) et plus communément l'erreur relative entre deux itérations. Globalement il y a des cas où ces erreur trouvent en même temps l'itération avec l'erreur minimale même si ça n'est pas toujours le cas. (argmin de la fonction d'erreur en fonction de l'itération).

On peut aussi calculer le résidu which is $\|Ax-b\|$ but c'est n² opéra°,

Le résidu et l'erreur relative donne la même courbes au final le problème est que l'on obtient pas forcément la meilleure précision en regardant la courbe de l'erreur relative car si la courbe d'erreur absolue contient des minimums locaux, la façon dont l'on converge vers ces minimums peut donner une erreur relative plus faible pour une erreur absolue plus haute



Courbe des erreurs absolue.

On peut smooth la courbe si elle est un peu "tremblante" en augmentant l'inertie.

Toute erreur est relative, on doit donc impérativement normaliser nos erreur pour se rendre compte du réelle pourcentage d'erreur par rapport au resultat que l'on veut obtenir

Normalisation d'erreur : NORMALISATION

Cas n°1 : On connait la valeur que l'on cherche.

$$\text{erreur absolue normalisée} = \frac{\|x^k - x\|}{\|x\|}$$

Cas n°2 : On utilise les erreur relatives :

$$\text{erreur relative normalisée} = \frac{\|x^{k+1} - x^k\|}{\|x^k\|}$$

Il arrive parfois que l'on ne converge pas vers le bon resultat si l'on a un conditionnement de notre machine très mauvais, à cause des erreurs de calcul répétées.

MÉTHODE DU GRADIENT

Problème d'optimisation : on cherche le minimum de J avec $J : \mathbb{R}^n \to \mathbb{R}$ which is $J(u) = \inf_{v \in \mathbb{R}^n} J(v)$

Problème d'optimisation avec contrainte → on cherche dans une partie de $\mathbb{R}^n$ au lieu de $\mathbb{R}^n$.

Méthode du gradient

On veut se diriger vers le point minimum de notre courbe en suivant une trajectoire opposée à celle du gradient.

1) On choisit un point au hasard : $p^0 = (x_0, y_0)$
2) On calcule son gradient en ce point :

$$\nabla J(x_0, y_0) = \begin{bmatrix} \frac{\partial J}{\partial x} \\ \frac{\partial J}{\partial y} \end{bmatrix} (x_0, y_0)$$

3) On prend la trajectoire opposée : $p^{k+1} = p^k - \mu \nabla J(p^k)$
4) From the top until $\|p^{k+1} - p^k\| < \varepsilon$

Algorithm

μ est le "learning rate" ya Fenou...

J(*x) gives the array's values as the arguments of J.

Si l'on veut faire des multiplications de matrices mais que l'on a des tenseurs il faut mieux utiliser des einsums.

↳ "indices dans la première ma, indices 2ème → indices res", ma1, ma2.

Si un des indices se répète dans le premier et le deuxième set d'indices, c'est qu'on va itérer dessus.
like np.einsum("ijk,ijk->ij", MA, M) sans être dans le res.

On peut aussi utiliser la fonction np.tensordot de numpy same same...

↳ np.tensordot(M, A, axes=(2,1)) * On somme sur l'axe 2 de M et sur l'axe 1 de A.

Gradient pour résoudre AX = B

Pour résoudre $AX = b$ on va chercher le minimum de la fonction :

$$J(x) = \frac{1}{2} x^T A x - bx$$

En ce minimum nous avons donc $J'(x) = 0$, et dans un certain cas, on a en plus la dérivée est $Ax - b$.

↳ $0 = Ax - b \rightarrow$ solve bien $AX = b$.

Pour ça il va aussi falloir calculer les dérivées.

la dérivée de $J(x)$ est $J'(x) = \frac{1}{2}(x^T A + Ax) - B$
↳ donc si A est symmétrique on a bien $J'(x) = Ax - B$.

Si en plus J a un minimum alors il est probable qu'on le trouve grâce à la méthode de gradient.

↳ J possède un minimum si J est définie positive et A symmétrique (ou à diagonale dominante).

Cela permet de savoir si la méthode du gradient va marcher.