10 Résolution numérique de systèmes linéaires

Le but de ce chapitre est l'étude de méthodes algorithmiques de résolutions de systèmes $AX = B$, où $A \in \mathcal{M}_{n,p}(\mathbb{R})$ et $B \in \mathcal{M}_{n,1}(\mathbb{R})$, l'inconnue X étant un élément de $\mathcal{M}_{p,1}(\mathbb{R})$. On peut bien sûr aussi travailler avec des coefficients complexes (en utilisant un langage manipulant des complexes, ce qui est le cas de Python), ou dans tout autre corps (si le langage connait les opérations dans ce corps, qu'elles soient implémentées initialement, ou qu'elles aient été programmées par la suite).
Savoir résoudre de tels systèmes est d'une importance capitale dans de nombreux domaines, par exemple en physique, en ingéniérie, ou encore dans toute l'imagerie vectorielle (l'image pixélisée est représentée par une matrice ; de nombreuses opérations sur les images nécessitent alors des résolutions de systèmes de taille importante). Dans ces domaines, les systèmes rencontrés sont souvent de très grande taille. Il est donc essentiel de savoir les résoudre de la façon la plus efficace possible. Nous nous contentons dans ce chapitre de rappeler la méthode du pivot, de tester l'aspect numérique, et d'étudier rapidement sa complexité. Nous introduisons également la décomposition $A = LU$ d'une matrice (Lower-Upper), donc en produit d'une matrice triangulaire inférieure et triangulaire supérieure : cela permet de ramener la résolution d'un système à la résolution de 2 systèmes triangulaires. Enfin, nous traitons de problèmes d'instabilité numériques, qu'on peut rencontrer dans le cas où le résultat d'un système est très sensible à une petite variation du second membre. Nous introduisons la notion de conditionnement, permettant de mesurer cette sensibilité, mais sans pour autant entrer dans une étude approfondie de cette notion.

I Structures de données adaptées en Python

Pour tout travail sur des données vectorielles et matricielles, nous utiliserons le module `numpy` de Python. Nous le supposerons importé sous l'alias `np`.
Le module `numpy` définit en particulier un type `array`, sur lequel sont définies certaines opérations matricielles usuelles. Nous renvoyons au chapitre 2 pour une description plus complète de ce module et des possibilités offertes. Nous renvoyons à l'aide de Python pour une description exhaustive.

L'instruction permettant de construire un tableau est `np.array`. Elle transforme une liste simple d'objets de même type numérique en tableau à une ligne.

```
>>> A = np.array([1,3,7,9])
>>> A
array([1, 3, 7, 9])
>>> type(A)
<class 'numpy.ndarray'>
>>> np.array([1,3,7,'a'])
```

```
array(['1', '3', '7', 'a'],
      dtype='<U1')
```

Le deuxième exemple ci-dessus montre la nécessité de l'homogénéité des éléments du tableau, qui doivent tous être de même type : les données numériques sont converties en chaînes de caractères (l'inverse n'étant pas possible). Le `dtype` indique un code pour le type des données (ici un texte en unicode)

Une liste de listes sera transformée en tableau bi-dimensionnel, chaque liste interne étant une des lignes de ce tableau. Il est évidemment nécessaire que toutes les listes internes aient la même taille. Dans le cas contraire, l'objet créé sera un tableau unidimensionnel, dont les attributs sont de type `list`. L'instruction `rank` donne la dimension spatiale du tableau (donc le degré d'imbrication) : 1 pour un tableau ligne, 2 pour une matrice, etc.

```
>>> A = np.array([[1,2,3],[3,6,7]])
>>> A
array([[1, 2, 3],
       [3, 6, 7]])
>>> type(A)
<class 'numpy.ndarray'>
>>> np.rank(A)
2
>>> B = np.array([[1,2,3],[3,6]])
>>> B
array([[1, 2, 3], [3, 6]], dtype=object)
>>> np.rank(B)
1
```

Le deuxième exemple illuste le fait que du point de vue de Python, B n'est pas bidimentionnel, mais est un tableau ligne, dont chaque donnée est une liste.

On peut bien sûr créer des objets de dimension plus grande :

```
>>> C =
np.array([[[[1,2],[3,4]],[[5,6],[7,8]]],[[[9,10],[11,12]],[[13,14],
[15,16]]]])
>>> C
array([[[[ 1, 2],
         [ 3, 4]],

        [[ 5, 6],
         [ 7, 8]]],


       [[[ 9, 10],
         [11, 12]],

        [[13, 14],
         [15, 16]]]])
>>> np.rank(C)
4
```

La dimension spatiale est suggérée par les sauts de ligne.
On peut récupérer le format (nombre de lignes, de colonnes, etc) grâce à la fonction `shape` :

```
>>> np.shape(A)
```

```
(2, 3)
>>> np.shape(B)
(2,)
>>> np.shape(C)
(2, 2, 2, 2)
```

Remarque 10.1.1 (tuple de taille 1)

Notez la syntaxe particulière pour le format de B. Cette syntaxe permet de différencier l'entier 2 du tuple $(2,)$ constitué d'une unique coordonnée. La virgule est nécessaire, car les règles de parenthésage ne permettent pas de différencier 2 et (2). Nous retrouverons cette syntaxe par la suite.

Comme les listes, les tableaux sont des objets mutables. Il faut donc faire attention à la dépendance possible entre plusieurs copies d'un tableau :

```
>>> D = A
>>> D
array([[1, 2, 3],
       [3, 6, 7]])
>>> A[1,1]= 8
>>> A
array([[1, 2, 3],
       [3, 8, 7]])
>>> D
array([[1, 2, 3],
       [3, 8, 7]])
```

On peut éviter ce désagrément en utilisant la fonction `np.copy`, équivalent de `deepcopy` pour les tableaux :

```
>>> D = np.copy(A)
>>> A[1,1]=10
>>> A
array([[ 1, 2, 3],
       [ 3, 10, 7]])
>>> D
array([[1, 2, 3],
       [3, 8, 7]])
```

Remarque 10.1.2 (Différences entre un tableau `numpy` et une liste)

- Toutes les entrées doivent être de même type
- Le format est immuable, et défini à la création du tableau : on ne peut pas modifier ce format en place (suppression ou ajout de ligne...), à moins de définir une nouvelle variable. En particulier, cela nécessite de définir dès le début un tableau de la bonne taille, par exemple rempli de 0 ou de 1. Il existe des fonctions permettant de le faire sans effort.

```
# Création d'un tableau de 0
>>> np.zeros(5)
array([ 0., 0., 0., 0., 0.])
>>> np.zeros((5,3))
array([[ 0., 0., 0.],
       [ 0., 0., 0.],
```

```
      [ 0.,  0.,  0.],
      [ 0.,  0.,  0.],
      [ 0.,  0.,  0.]])

# Création d'un tableau de 1
>>> np.ones(3)
array([ 1.,  1.,  1.])
>>> np.ones((3,8))
array([[ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
      [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
      [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]])

# Création de la matrice identité I (prononcez à l'anglaise!)
>>> np.eye(3)
array([[ 1.,  0.,  0.],
      [ 0.,  1.,  0.],
      [ 0.,  0.,  1.]])

# Création d'une matrice diagonale:
>>> np.diag([1,3,2,7])
array([[1, 0, 0, 0],
      [0, 3, 0, 0],
      [0, 0, 2, 0],
      [0, 0, 0, 7]])

# Création d'une matrice (f(i,j,k,...))
>>> def f(i,j):
...     return i+j
...
>>> np.fromfunction(f,(3,4))
array([[ 0.,  1.,  2.,  3.],
      [ 1.,  2.,  3.,  4.],
      [ 2.,  3.,  4.,  5.]])
>>> np.fromfunction(lambda x: x**2 ,(6,))
array([ 0.,  1.,  4.,  9., 16., 25.])
```

Remarquez dans la dernière fonction l'utilisation du tuple $(6,)$ pour désigner le format d'un tableau ligne. On peut aussi créer des tableaux de valeurs uniformément espacées :

```
# Création d'un tableau de valeurs régulièrement espacées entre a et b
# en imposant le nombre de valeurs
# Attention au fait que a et b sont comptabilisés dans les n valeurs.
>>> np.linspace(5,9,10)
array([ 5.       ,  5.44444444, 5.88888889, 6.33333333, 6.77777778,
       7.22222222, 7.66666667, 8.11111111, 8.55555556, 9.      ])
>>> np.linspace(5,9,11)
array([ 5. , 5.4, 5.8, 6.2, 6.6, 7. , 7.4, 7.8, 8.2, 8.6, 9. ])

# Création d'un tableau de valeurs régulièrement espacées entre a et b
# en imposant le pas
# Attention au fait que la borne b est prise au sens strict.
>>> np.arange(5,10,0.5)
```

```
array([ 5. , 5.5, 6. , 6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
>>> np.arange(5,10,0.7)
array([ 5. , 5.7, 6.4, 7.1, 7.8, 8.5, 9.2, 9.9])
```

De nombreuses opérations matricielles sont définies. Attention aux faux amis cependant. Comme on l'a vu, `np.rank` ne correspond pas au rang de la matrice, mais à sa dimension spatiale. Autre faux ami : le produit $A * B$ n'est pas le produit matriciel usuel, mais le produit de Schur (produit coordonnée par coordonnée). Le produit matriciel est donné par la fonction `np.dot`

```
>>> A = np.array([[1,2],[3,4]])
>>> B = np.array([[2,2],[0,3]])
# Produit de Schur
>>> A*B
array([[ 2, 4],
       [ 0, 12]])
# Produit matriciel
>>> np.dot(A,B)
array([[ 2, 8],
       [ 6, 18]])
# Autre syntaxe possible, par suffixe:
>>> A.dot(B)
array([[ 2, 8],
       [ 6, 18]])
```

Les fonctions les plus utiles sont décrites dans le chapitre 2. Retenons en particulier (puisque c'est ce qui nous intéresse dans ce chapitre) la fonction `np.linalg.solve` pour la résolution d'un système $AX = B$ (le vecteur B doit être donné sous forme d'un tableau ligne, ou d'une liste)

```
>>> np.linalg.solve(np.array([[1,3],[2,-3]]),[1,4])
array([ 1.66666667, -0.22222222])
```

Cette fonction ne résout que les systèmes de Cramer :

```
>>> np.linalg.solve(np.array([[1,3],[2,6]]),[1,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.3/site-packages/numpy/linalg/linalg.py",
  line 328, in solve
    raise LinAlgError('Singular matrix')
numpy.linalg.linalg.LinAlgError: Singular matrix
```

II Rappels sur la méthode du pivot de Gauss

La méthode du pivot est une méthode d'échelonnement d'une matrice donnée, par opérations admissibles (c'est-à-dire réversibles) sur les lignes. Ces opérations préservent le noyau, donc les solutions du système homogène. Si on effectue les mêmes opérations sur la matrice colonne B du second membre, on obtient le même espace affine de solutions.

On rappelle que les opérations admissibles sont :

- $L_i \leftrightarrow L_j$: l'échange de deux lignes (permutation)
- $L_i \leftarrow \lambda L_i$: la multiplication d'une ligne par un scalaire **non nul** (dilatation)
- $L_i \leftarrow L_i + \lambda L_j$: l'ajout à une ligne d'une autre, multipliée par un scalaire (transvection).

On rappelle que ces opérations correspondent matriciellement à la multiplication à gauche par certaines matrices de codage (voir le cours de mathématiques).

On rappelle les grandes lignes de l'algorithme d'échelonnement de Gauss :

Méthode 10.2.1 (Échelonnement par la méthode du pivot)

- On cherche un élément non nul dans la première colonne ;
- **Le choix du pivot est important**. Pour des raisons de précision numérique, il est judicieux de choisir le pivot de **valeur absolue maximale.** On fera aussi attention au test à 0, à faire sous forme d'une inégalité, et non d'une égalité stricte, si on ne veut pas avoir de surprises désagréables.
  - Si on n'en trouve pas, on passe à la colonne suivante, et on recommence de même.
  - Si on en trouve, on le positionne en haut par un échange. C'est notre premier pivot. On annule tous les autres coefficients de la première colonne à l'aide de celui-ci, par des opérations de transvection, puis on passe à la colonne suivante, mais en ne touchant plus à la première ligne (contenant le premier pivot). En particulier, la recherche d'un pivot suivant ne se fera pas sur la première ligne, et ce pivot ne sera remonté que sur la deuxième ligne.
- On continue de la sorte jusqu'à épuisement des colonnes de A. Les différents pivots se positionnent sur les lignes successives.
- Quitte à faire des opérations de dilatation, on peut s'arranger pour qu'à l'issue du calcul, tous les pivots soit égaux à 1.

Méthode 10.2.2 (Pivot remontant)

- Quitte à conserver en mémoire la liste des indices de colonne des pivots, on retrouve facilement la position et la valeur des pivots. On peut alors, par des opérations de tranvection, annuler tous les coefficients situés au dessus des pivots, de sorte à ce que les pivots soient les seuls composantes non nuls de leurs colonnes.
- On fera attention a être économe en calcul pour cette étape : commencer par le dernier pivot évite de manipuler trop de termes non nuls (et donc d'accumuler des erreurs sur des termes qui nous intéressent). Par ailleurs, il est inutile de faire les opérations sur les colonnes précédant le pivot (opérations triviales, mais qui prennent du temps informatiquement parlant).

Dans le cas de la résolution d'un système $AX = B$, en faisant en parallèle les mêmes opérations sur B, on obtient donc un système équivalent $A'X = B'$, où A' est échelonnée, de pivots tous égaux à 1, les éléments situés en-dessous, et au-dessus des pivots étant nuls. Notons $j_1 < j_2 < \cdots < j_r$ les colonnes des r pivots (on remarquera que r est le rang de A). On obtient alors facilement une solution particulière, ainsi qu'une base de l'espace des solutions de l'équation homogène. On note $B' = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$.

Méthode 10.2.3

- Si le système est compatible (c'est le cas ssi $b_i = 0$ pour tout $i > r$), une solution particulière est le vecteur $X_0 = (x_i)_{1 \leqslant i \leqslant p}$, tel que

$$x_i = \begin{cases} b_k & \text{si } i = j_k,\ k \in [\![1, r]\!] \\ 0 & \text{sinon} \end{cases}$$

- Une base est obtenue en énumérant les vecteurs X_j obtenus, pour $j \in [\![1, p]\!] \setminus \{j_1, \ldots, j_r\}$ par la construction suivante :
  - les coordonnées de X_j distinctes de j et des j_k sont nulles
  - la j-ième coordonnée de X_j est égale à 1
  - la colonne extraite de X_j en ne conservant que les lignes $j_1, \ldots, j_r$ est égale à $-C_j$, où C_j est la j-ième colonne de A', tronquée au-delà de la r-ième coordonnée.

Ainsi, en notant, pour un k-uplet $(\ell_1, \dots, \ell_k)$, $p_{\ell_1,\dots,\ell_k}$ la projection de $\mathbb{R}^n$ dans $\mathbb{R}^k$ définie par

$$p_{\ell_1,\dots,\ell_k}(x_1, \dots, x_n) = (x_{\ell_1}, \dots, x_{\ell_k}),$$

et en identifiant les vecteurs colonnes à des éléments de $\mathbb{R}^k$, le vecteur X_j est défini par :

$$p_j(X_j) = 1, \qquad p_{j_1,\dots,j_r}(X_j) = -C_j, \qquad \text{et} \qquad p_{[\![1,N]\!]\setminus\{j,j_1,\dots,j_r\}} = 0.$$

Théorème 10.2.4 (Complexité de la méthode du pivot)

La méthode du pivot sur une matrice de $\mathcal{M}(n,p)$ est en $O(np^2)$. En particulier, elle est cubique sur des matrices carrées.

◁ **Éléments de preuve.**

Le nombre de pivot est majoré par le nombre de colonnes p. Pour chaque colonne, on effectue au plus n opérations sur les lignes, chacune de ces opérations nécessitant p opérations réalisatbles en temps constant (pour traiter la totalité de la ligne). ▷

Le nombre de pivots étant aussi majoré par le nombre de lignes (et s'il n'y a pas de pivot, il n'y a rien à faire), on obtient aussi une complexité en $O(n^2p)$. Donc au final en $O(np\min(n,p))$.
Comme on l'a vu en cours de mathématiques, la méthode du pivot peut être adaptée de différentes manières :

Méthode 10.2.5 (Adaptations de la méthode du pivot)

- **Calcul du rang :** on se contente d'un premier passage (échelonnement simple), en comptant le nombre de pivots utilisés, ce qui donnera le rang.
- **Calcul de l'inverse :** on effectue un échelonnement double, en remarquant que dans la première étape (descente), si on ne trouve pas de pivot admissible sur une colonne, la matrice n'est pas inversible (on obtiendra au bout une matrice triangulaire dont certains coefficients diagonaux sont nuls) : on peut donc interrompre l'algorithme dans ce cas. L'échelonnement double de la matrice, avec normalisation des pivots, nous donne alors, en cas d'inversibilité, la matrice I_n. On effectue en parallèle les mêmes opérations sur la matrice initiale I_n, on obtient en fin d'algorithme la matrice A^{-1}.
- **Calcul de la matrice de passage** de l'échelonnement, c'est-à-dire de P telle que $PA = A'$: c'est une généralisation de la méthode de calcul de l'inverse, en suivant exactement le même principe : P est le produit des matrices de codage des opérations, donc obtenu en effectuant sur I_n les opérations correspondantes. On trouve donc P en effectuant en parallèle sur I_n les mêmes opérations que sur A pour obtenir A'.

III Décomposition LU

Définition 10.3.1 (Décomposition LU)

Une décomposition LU d'une matrice carrée A est une décomposition de A en un produit $A = LU$ d'une matrice triangulaire inférieure L (Lower) et d'une matrice triangulaire supérieure (Upper).

On peut obtenir, sous certaines conditions, une décomposition LU en adaptant la dernière variante de la méthode du pivot exposée ci-dessus. Au lieu de rechercher P tel que $PA = A'$, on recherche $Q \in \mathrm{GL}_n(\mathbb{K})$ tel que $A = QA'$. On pourrait se dire qu'il suffit d'inverser P, mais cela oblige à faire 2 pivots (un premier

pour réduire A, l'autre pour inverser P), ce n'est pas très optimal. On adapte alors la méthode précédente, en remarquant que si $P = P_k \cdots P_1$, où les P_i codent les opérations successives, alors

$$Q = P_1^{-1} \cdots P_k^{-1} = I_n P_1^{-1} \cdots P_k^{-1}.$$

Méthode 10.3.2 (Trouver Q telle que $A = QA'$)

On trouve Q en effectuant sur I_n les opérations sur les colonnes correspondant aux matrices inverses des matrices de codage des opérations sur les lignes effectuées sur A pour obtenir A'. Plus explicitement, à chaque opération sur les lignes de A, on effectue une opération correspondante sur les colonnes de I_n :

- $L_i \leftarrow \lambda L_i$ correspond à $C_i \leftarrow \frac{1}{\lambda} C_i$
- $L_i \leftrightarrow L_j$ correspond à $C_i \leftrightarrow C_j$
- $L_i \leftarrow L_i + \lambda L_j$ correspond à $C_j \leftarrow C_j - \lambda C_i$.

On obtient alors une décomposition LU en remarquant que beaucoup de matrices d'opérations sont triangulaires : il suffit donc de se limiter aux opérations pour lesquelles la matrice associée (donc son inverse) est triangulaire inférieure : c'est le cas des opérations de dilatation et de transvection $L_i \leftarrow L_i + \lambda L_j$ lorsque $i > j$. Or, s'il n'y a pas besoin de faire d'échanges de lignes, le pivot de Gauss peut s'effectuer avec ces seules opérations. De plus, les opérations de dilatation ne sont pas strictement nécessaires, et les matrices triangulaires en jeu ont alors toutes une diagonale de 1. On obtient alors, sous une hypothèse nous assurant qu'on n'aura pas d'échange de ligne à faire :

Théorème 10.3.3 (Décomposition LU)

Soit $A \in \mathrm{GL}_n(\mathbb{R})$, telle que pour tout $k \in [\![1, n]\!]$, la sous-matrice carrées obtenue par extraction des k premières lignes et des k premières colonnes soit inversible. Alors A admet une décomposition LU. On peut de plus imposer que L soit à diagonale égale à 1.

◁ **Éléments de preuve.**

Il suffit de montrer qu'on peut échelonner A par la méthode du pivot en n'utilisant que des transvections. Remarquons que l'hypothèse portant sur les mineurs principaux reste valable à chaque étape (puisqu'on ne modifie que les lignes sous le pivot : par restriction, les opérations effectuées fournissent des opérations admissibles sur les sous-matrices carrées calées en haut à gauche, qui restent donc inversibles).

Or, à l'issue du traitement des $k-1$ premières colonnes, la matrice carrée extraite d'ordre k calée en haut à gauche est triangulaire inversible, donc le coefficient (k, k) est non nul, et peut servir de pivot pour continuer. ▷

Cette décomposition se trouve explicitement par la méthode suivante :

Méthode 10.3.4 (Trouver une décomposition LU)

Réduire A en une matrice échelonnée U, en n'utilisant ni échange de ligne, ni dilatation (si on veut L de diagonale 1). Cette matrice échelonnée est triangulaire supérieure. On effectue sur I_n les opérations sur les colonnes correspondant aux opérations sur les lignes dans le sens décrit plus haut, cela fournit la matrice L.

Remarque 10.3.5 (Intérêt de la décomposition LU)

- Cette décomposition est intéressante notamment dans le cas où on a un grand nombre de systèmes $AX = B_i$ à résoudre, associés à la même matrice A. Plutôt que de refaire tout le calcul depuis le début pour chaque vecteur B_i, on fait le calcul de la décomposition une fois pour toutes : on est ramené à la résolution de 2 systèmes triangulaires $LUX = B$. On obtient d'abord UX par résolution du système de matrice L, puis X par résolution du système de matrice U.

- Il pourra être avantageux de programmer explicitement un algorithme spécifique de résolution des systèmes triangulaires inférieurs et supérieurs, afin d'optimiser au maximum les temps de calcul, en omettant les opérations avec les coefficients qu'on sait être nuls.
- Cette méthode est plus avantageuse que de calculer A^{-1} jusqu'au bout puis de faire les calculs de $A^{-1}B_i$. En effet, si les résolutions des systèmes linéaires triangulaires sont bien programmés, elles ne sont pas plus coûteuses que le calcul de $A^{-1}B_i$, cela permet donc d'éviter la phase finale du calcul de A^{-1}.

Cette méthode possède tout de même des inconvénient notables, comme pas exemple le fait que l'on ne peut pas choisir le pivot. Cela peut éventuellement diminuer la précision de calcul. On peut pallier à ce problème en décomposant A sous une forme un peu plus complexe.

Proposition 10.3.6 (Décomposition PLU, admis, donné uniquement pour la culture)
Toute matrice $A \in \mathrm{GL}(\mathbb{R})$ se décompose sous la forme PLU, où P est une matrice de permutation, L une matrice triangulaire inférieure à diagonale 1, U une matrice triangulaire supérieure.

IV Problèmes de conditionnement

On part de l'exemple suivant :

$$H_n = \left(\frac{1}{i+j-1}\right)_{1\leqslant i,j\leqslant n}.$$

Cette matrice est appelée matrice de Hilbert. On observe sur des exemples que la résolution de $H_nX = B$ est problématique :

```
>>> np.linalg.solve(H,[1,1,1,1])
array([ -4.,  60., -180., 140.])
>>> np.linalg.solve(H,[1.01,0.99,1.01,0.99])
array([ 1.16,  3.  , -43.8 , 51.8 ])
```

Cet exemple illustre la très grande variabilité du résultat en fonction du second membre B : une toute petite variation sur B fournit de très grosses variations sur la solution du système. Cela peut engendrer de grosses imprécisions sur les calculs, du fait que toute erreur d'arrondi sur B sera beaucoup amplifiée lors de la résolution du système.
Il n'y a pas grand chose à faire, sinon savoir détecter quelles sont les matrices qui vont nous donner des situations similaires, de sorte à pouvoir valider la précision des calculs dans les autres cas. Pour cela, on utilise une quantité permettant de mesurer l'amplification d'une variation sur B lors de la résolution du système.

Définition 10.4.1 (Norme matricielle)
La norme (euclidienne canonique) de $X \in \mathcal{M}_{n,1}(\mathbb{R})$ est définie par

$$\|X\| = \sqrt{x_1^2 + \cdots + x_n^2}, \text{ où } X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}.$$

On définit alors la norme d'une matrice $A \in \mathcal{M}_n(\mathbb{R})$ par :

$$\|A\| = \sup_{X\in\mathbb{R}^n\setminus\{0\}} \frac{\|AX\|}{\|X\|}.$$

Cette quantité est bien définie du fait que

$$\sup_{X\in\mathbb{R}^n\setminus\{0\}} \frac{\|AX\|}{\|X\|} = \sup_{X\in B(0,1)} \|AX\|,$$

(provenant de l'invariance par dilatation de X), en utilisant la compacité de $B(0,1)$ et la continuité de la norme d'un vecteur de $\mathbb{R}^n$.

Proposition 10.4.2

$A \mapsto \|A\|$ est une norme sur $\mathcal{M}_n(\mathbb{R})$:

(i) $\|A\| = 0$ *si et seulement si* $A = 0$

(ii) pour tout $\lambda \in \mathbb{R}$, $\|\lambda A\| = |\lambda| \cdot \|A\|$,

(iii) pour tout A, B, $\|A + B\| \leqslant \|A\| + \|B\|$ *(inégalité triangulaire)*

De plus, il s'agit d'une norme matricielle*, c'est-à-dire :*

(iv) pour tout $A, B \in \mathcal{M}_n(\mathbb{R})$, $\|AB\| \leqslant \|A\| \cdot \|B\|$.

◁ **Éléments de preuve.**

Les 3 premiers points sont classiques ; pour le troisième, utiliser l'ingalité triangulaire pour la norme euclidienne canonique de $\mathbb{R}^n$.

Pour le dernier point, on pourra utiliser le lemme qui suit. ▷

Lemme 10.4.3

Soit $A \in \mathcal{M}_n(\mathbb{R})$ et $X \in \mathcal{M}_{n,1}(\mathbb{R})$. On a :

$$\|AX\| \leqslant \|A\| \cdot \|X\|.$$

◁ **Éléments de preuve.**

C'est le fait que le sup est un majorant. ▷

On définit alors le conditionnement par :

Définition 10.4.4 (Conditionnement d'une matrice)

Soit $A \in \mathrm{GL}_n(\mathbb{R})$. Le conditionnement de A est :

$$\mathrm{cond}(A) = \|A\| \cdot \|A^{-1}\|.$$

Proposition 10.4.5

On a toujours $\mathrm{cond}(A) \geqslant 1$.

◁ **Éléments de preuve.**

Comparer à la norme de I_n. ▷

Nous allons voir que le conditionnement permet de contrôler les variations de la solution X du système en fonction des variations de B. Plus $\mathrm{cond}(A)$ est proche de 1, plus ce contrôle va être bon. Si $\mathrm{cond}(A)$ est grand, ce contrôle va être mauvais ; C'est ce qu'il se passe pour les matrices de Hilbert (on peut montrer que $\mathrm{cond}(H_4) = 15514$). Ainsi, un conditionnement proche de 1 est une garantie de fiabilité tdu résultat.

Théorème 10.4.6

Soit B et $B+\delta B$ deux colonnes de $\mathbb{R}^n$, et X et $X+\delta X$ les solutions obtenues pour le système de matrice A, associé à ces deux seconds membres. On a alors :

$$\frac{\|\delta X\|}{\|X\|} \leqslant \mathrm{cond}(A)\frac{\|\delta B\|}{\|B\|}.$$

◁ **Éléments de preuve.**

Utiliser le lemme 10.4.3 avec $B = AX$ et $\delta X = A^{-1}\delta B$ et combiner les deux inégalités. ▷

Ce théorème affirme que les variations relatives sur B sont amplifiées sur la solution du système par un facteur au plus égal à $\mathrm{cond}(A)$. Assez logiquement, on ne peut pas espérer un meilleur contrôle de l'erreur sur X que sur B (car $\mathrm{cond}(A) \geqslant 1$), et une grande valeur de $\mathrm{cond}(A)$ nous donne un très mauvais contrôle.

Définition 10.4.7 (Bon et mauvais conditionnement)

Si $\mathrm{cond}(A)$ n'est pas trop grand par rapport à 1, on dit que A est bien conditionné. Si A est grand par rapport à 1, on dit que A est mal conditionné.

C'est une notion qui reste très subjective, on ne quantifie pas la valeur seuil passant de l'une à l'autre des deux situations.