Chapitre $\underline{1101}_2$
Résolution de systèmes linéaires

16 février 2018

1 Introduction motivante : l'équation de la chaleur

Intéressons-nous au problème monodimensionnel de la propagation de la chaleur dans une barre métallique longue de 1 m. Un point de cette barre est repéré par une abscisse x allant de 0 à 1. On note $T(x,t)$ la température de la barre au point d'abscisse x et au temps t. Le problème comporte également deux conditions dites *aux limites* : pour tout $t \in \mathbb{R}$, les températures $T(0,t)$ et $T(1,t)$ sont fixes et notées T_g et T_d. Il comporte également une condition initiale : pour $t = 0$, l'ensemble de la barre et à la température T_0, *i.e.* pour tout $x \in [0,1]$, $T(x,0) = T_0$.

La modélisation physique classique permet d'aboutir à une *équation aux dérivées partielles* (EDP) vérifiée par T, et appelée *équation de la chaleur* :

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \ ,$$

où α est la *diffusivité thermique*.

L'intervalle $[0,1]$ est subdivisé en N sous-intervalles de longueur constante $h_x = \frac{1}{N}$. Pour tout $i \in [\![0, N]\!]$, on note $x_i = ih$. On dit qu'on a *discrétisé* ou *maillé* l'intervalle $[0,1]$: les points x_i sont appelés les *nœuds* du maillage, et les intervalles $[x_i, x_{i+1}]$ en sont les *cellules*.

Le temps est également discrétisé en sous-intervalles de longueur h_t : h_t est donc le *pas temporel*, et h_x le *pas spatial*. Nous noterons alors T_i^n la température au point d'abscisse x_i et au temps $t = nh_t$. Par extension, pour toute fonction φ, φ_i^n désignera la valeur de φ au point d'abscisse x_i et au temps $t = nh_t$.

Comme pour la méthode d'Euler, nous allons approcher les dérivées intervenant dans l'équation de la chaleur en utilisant des taux d'acccroissement, ce qui permettra, en connaissant les T_i^n pour tout i et à un certain n, d'en déduire une relation de récurrence donnant les T_i^{n+1}, pour tout i.

En toute rigueur, les T_i^n ne désigneront pas la valeur exacte de $T(x_i, nh_t)$, que nous ne savons pas calculer, mais une approximation.

Là aussi, il existe deux approximations classiques des dérivées, menant à deux schémas numériques différents : le schéma *explicite* et le schéma *implicite*.

1.1 Le schéma explicite

Voici les approximations utilisées :

$$\left(\frac{\partial T}{\partial t}\right)_i^n = \frac{T_i^{n+1} - T_i^n}{h_t}$$

et

$$\left(\frac{\partial^2 T}{\partial x^2}\right)_i^n = \frac{\frac{T_{i+1}^n - T_i^n}{h_x} - \frac{T_i^n - T_{i-1}^n}{h_x}}{h_x} = \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{h_x^2}.$$

En posant $\lambda = \alpha\dfrac{h_t}{h_x^2}$, la température à l'itération $n+1$ est donnée, pour tout $i \in [\![1, N-1]\!]$, par :

$$T_i^{n+1} = \lambda T_{i-1}^n + (1-2\lambda)T_i^n + \lambda T_{i+1}^n$$

Ou encore, sous forme matricielle :

$$\begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ T_{N-2} \\ T_{N-1} \end{pmatrix}^{n+1} = \begin{pmatrix} 1-2\lambda & \lambda & 0 & \cdots & 0 \\ \lambda & 1-2\lambda & \lambda & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \lambda & 1-2\lambda & \lambda \\ 0 & 0 & 0 & \lambda & 1-2\lambda \end{pmatrix} \begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ T_{N-2} \\ T_{N-1} \end{pmatrix}^n + \lambda \begin{pmatrix} T_g \\ 0 \\ \vdots \\ 0 \\ T_d \end{pmatrix}.$$

1.2 Le schéma implicite

Voici les approximations utilisées :

$$\left(\frac{\partial T}{\partial t}\right)_i^{n+1} = \frac{T_i^{n+1} - T_i^n}{h_t}$$

et

$$\left(\frac{\partial^2 T}{\partial x^2}\right)_i^{n+1} = \frac{\frac{T_{i+1}^{n+1} - T_i^{n+1}}{h_x} - \frac{T_i^{n+1} - T_{i-1}^{n+1}}{h_x}}{h_x} = \frac{T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}}{h_x^2}.$$

En posant $\lambda = \alpha\dfrac{h_t}{h_x^2}$, la température à l'itération $n+1$ est donnée, pour tout $i \in [\![1, N-1]\!]$, par :

$$(1+2\lambda)T_i^{n+1} - \lambda(T_{i+1}^{n+1} + T_{i-1}^{n+1}) = T_i^n$$

Ou encore, sous forme matricielle :

$$\begin{pmatrix} 1+2\lambda & -\lambda & 0 & \cdots & 0 \\ -\lambda & 1+2\lambda & -\lambda & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & -\lambda & 1+2\lambda & -\lambda \\ 0 & 0 & 0 & -\lambda & 1+2\lambda \end{pmatrix} \begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ T_{N-2} \\ T_{N-1} \end{pmatrix}^{n+1} = \begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ T_{N-2} \\ T_{N-1} \end{pmatrix}^{n} + \lambda \begin{pmatrix} T_g \\ 0 \\ \vdots \\ 0 \\ T_d \end{pmatrix}.$$

Les choses ne sont donc pas aussi directes que lors de l'utilisation du schéma explicite : un simple produit matriciel ne suffit pas pour passer de la n^{e} itération à la $(n+1)^{\text{e}}$, il faut ici *résoudre un système linéaire*.

On pourrait être tenté d'écarter le schéma implicite pour éviter ce problème, mais il se trouve que ce schéma présente de qualité de *stabilité* que n'a pas le schéma explicite : il permet d'utiliser des pas de temps plus grands (donc de diminuer la taille de la matrice) sans trop altérer la qualité du résultat, et est plus tolérant face aux erreurs d'approximation. Il est donc très utilisé en pratique, en particulier pour étudier des phénomènes lents sur de grandes plages de temps.

Ici le problème est monodimensionnel, mais il existe des problèmes en dimension 2 ou 3. Par exemple en dimension 3, les dérivées partielles peuvent se faire selon x, y, z ou t. On rajoute un second et un troisième pas d'espace : h_y et h_z. Le domaine spatial d'étude est alors maillé par des parallélépipèdes. Mais le principe de base reste le même.

On peut également mailler le domaine par des triangles, des tétraèdres, ou toute autre forme géométrique (voir les figures 1 et 2 pour des illustrations).

Exemples :

— Diffusion de la chaleur dans un matériau ;

— Calcul du champ magnétique induit par une source électromagnétique ;

— Simulation de déformation d'une voiture en cas de choc ;

— Prévisions météorologiques ;

— ...

Dans le cas de prévisions météorologiques sur de grandes échelles d'espace et / ou de temps, le nombre de noeuds du maillage peut être énorme, parfois plusieurs milliers. D'où l'importance de disposer d'algorithmes efficaces de résolution de systèmes linéaires.

Comme sur l'exemple de l'équation de la chaleur, les matrices considérées sont souvent *creuses* (elles contiennent beaucoup plus de zéros qu'autre chose), ce qui peut être exploité.

2 Méthode de Gauss-Jordan

On l'appelle aussi « méthode du pivot de Gauss » : c'est celle qui a été vue en mathématiques ! Lorsqu'on résout un système « à la main », on ne la suit pas forcément

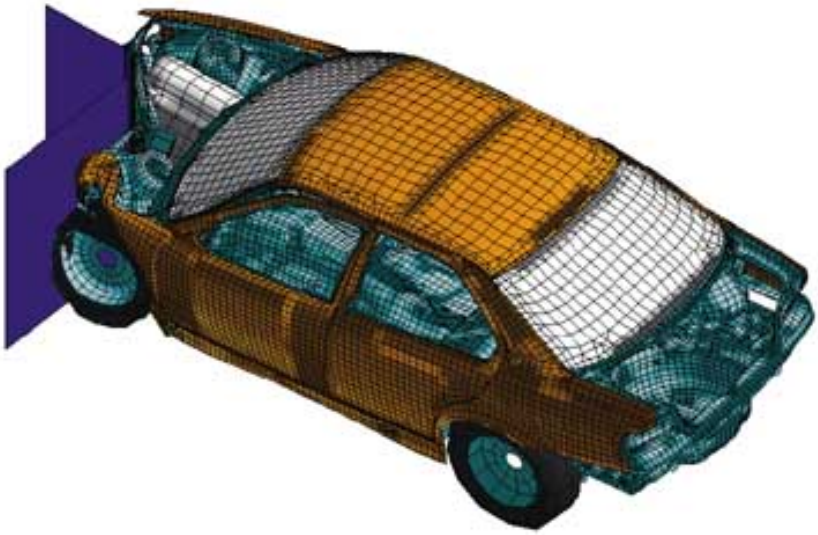

FIGURE 1 – Maillage utilisé pour simuler la déformation d'une voiture lors d'une collision.

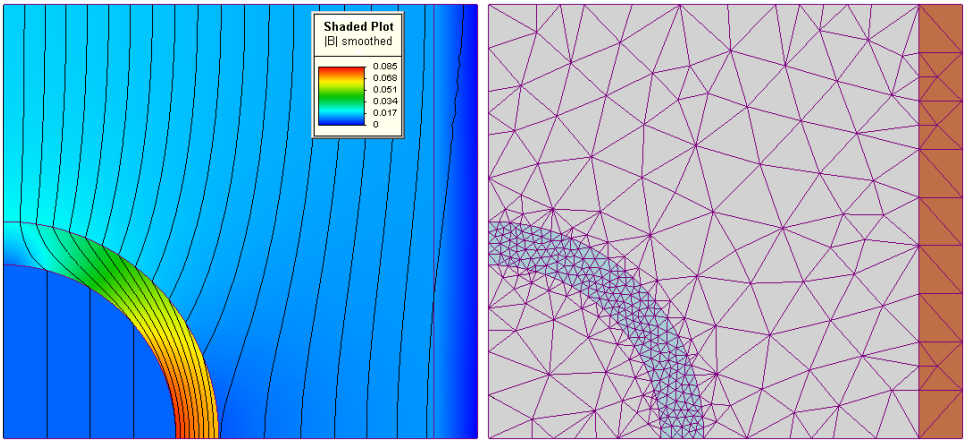

FIGURE 2 – Solution d'une équation magnétostatique. À droite, le maillage utilisé.

à la lettre. Nous allons la détailler d'un point de vue algorithmique.

On veut résoudre l'équation $Ax = b$, où $A = (a_{i,j}) \in \mathscr{M}_n(\mathbb{K})$ et $b = (b_i) \in \mathscr{M}_{n1}(\mathbb{K})$ sont donnés, d'inconnue $x = (x_i) \in \mathscr{M}_{n1}(\mathbb{K})$. Cette équation se voit naturellement comme le système linéaire suivant.

$$\left\{\begin{array}{ccccccccccc} a_{1,1}x_1 & + & \dots & + & a_{1,j}x_j & + & \dots & + & a_{1,n}x_n & = & b_1 \\ \vdots & & & & \vdots & & & & \vdots & & \vdots \\ a_{i,1}x_1 & + & \dots & + & a_{i,j}x_j & + & \dots & & a_{i,n}x_n & = & b_i \\ \vdots & & & & \vdots & & & & \vdots & & \vdots \\ a_{n,1}x_1 & + & \dots & + & a_{n,j}x_j & + & \dots & + & a_{n,n}x_n & = & b_n \end{array}\right. \qquad (\mathscr{S})$$

2.1 Cas où A est triangulaire

Si A est triangulaire (par exemple supérieure) et inversible (ce qui équivaut à $\forall k,\ a_{kk} \neq 0$), la résolution est facile. Le système $(\mathscr{S})$ s'écrit alors plus simplement.

$$\left\{\begin{array}{ccccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & \cdots & + & a_{1n}x_n & = & b_1 \\ & & a_{22}x_2 & + & \cdots & + & \cdots & + & a_{2n}x_n & = & b_2 \\ & & & \ddots & & & \vdots & & \vdots & & \vdots \\ & & & & a_{kk}x_k & + & \cdots & + & a_{kn}x_n & = & b_k \\ & & & & & \ddots & \vdots & & \vdots & & \vdots \\ & & & & & & & & a_{nn}x_n & = & b_n \end{array}\right.$$

Les solutions de $(\mathscr{S})$ se calculent de proche en proche de la manière suivante.

$$\begin{aligned} x_n &= \frac{1}{a_{nn}}\,(b_n) \\ x_{n-1} &= \frac{1}{a_{n-1,n-1}}\,(b_{n-1} - a_{n-1,n}x_n) \\ &\vdots \\ x_k &= \frac{1}{a_{k,k}}\,(b_k - a_{k,k+1}x_{k+1} - \cdots - a_{k,n}x_n) \end{aligned}$$

2.2 Cas général

On suppose que le système (*i.e.* A) est carré ainsi qu'inversible. On se ramène au cas triangulaire : pour cela on va effectuer des opérations sur les lignes du système $Ax = b$. Ces opérations sont

— des échanges de lignes,

— des ajouts à une ligne de combinaisons linéaires d'autres lignes.

On peut voir chacune de ces opérations comme une opération matricielle, via la multiplication à gauche de matrices d'échange et de transvection.

Exemple 2.2.1 (Matrice d'échange)**.** Soit

$$E_{1,3} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Soit B une matrice ayant 4 lignes L_1, L_2, L_3, L_4. Alors, $E_{1,3}B$ a les mêmes lignes que B sauf la 3^{e} ligne qui est L_1 et la première ligne qui est L_3. Multiplier B à gauche par $E_{1,3}$ revient donc à effectuer l'opération $L_1 \leftrightarrow L_3$.

Exemple 2.2.2 (Matrice de transvection)**.** Soit

$$T_\lambda = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \lambda & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Soit B une matrice ayant 4 lignes L_1, L_2, L_3, L_4. Alors, $T_{1,3,\lambda}B$ a les mêmes lignes que B sauf la 3^{e} ligne qui est $L_3 + \lambda L_1$.

Définition 2.2.3 (Matrice de transvection)**.** Soit $n \in \mathbb{N}^*$, $i, j \in [\![1, n]\!]$ avec $i \neq j$, soit $\lambda \in \mathbb{R}$. La matrice de transvection $T_{i,j,\lambda}$ est $I_n + \lambda E_{i,j}$, où $E_{i,j}$ est la matrice élémentaire nulle, hormis le coefficient d'indice (i, j) qui vaut 1.

Ainsi, $T_{i,j,\lambda}$ correspond à la matrice identité, sauf pour son coefficient d'indice (i, j) qui vaut λ.

Numérotons, suivant notre habitude pythonesque, les lignes de 0 à $n-1$ et les colonnes de 0 à $n-1$, notons L_i la i^{e} ligne de A pour chaque $i \in [\![0, n[\![$. L'algorithme du pivot est le suivant :

1. Phase de *descente* : pour chaque $i = 0, 1, \ldots, n-1$.
   a) Trouver j dans $[\![i, n[\![$ tel que a_{ji} soit non nul (c'est toujours possible, car le système est inversible).
   b) Échanger les lignes i et j dans A et dans b.
   c) Poser $p = a_{ii}$ (c'est le *pivot*).
   d) Pour j de $i+1$ à $n-1$:
      i. Remplacer b_j par $b_j - \frac{a_{ji}}{p} b_i$.
      ii. Remplacer la ligne L_j de A par $L_j - \frac{a_{ji}}{p} L_i$.
2. Phase de *remontée* : A est maintenant triangulaire, on peut calculer la solution avec la méthode précédente.

Remarque 2.2.4. Cet algorithme modifie la matrice A ainsi que le vecteur b. En pratique, on l'effectuera après avoir recopié les données.

Attention : se contenter de choisir un pivot non nul peut être problématique. Par exemple, un coefficient devrait être nul après un certain nombre de calculs. À cause d'erreurs d'arrondis, il est représenté par un coefficient non nul, et peut être choisi comme pivot. Ou bien, à cause d'erreurs d'arrondis, un coefficient qui devait-être non nul est représenté comme le flottant nul. Le système n'est alors plus inversible !

Pour palier cela, on utilise la méthode du *pivot partiel*. Il suffit de remplacer l'étape 1a) par la suivante.

Trouver j dans $[\![i, n[\![$ tel que $|a_{ji}|$ soit maximale.

3 Matrices avec `numpy`.

Nous représenterons les matrices par des tableaux bidimensionnels, en utilisant le type `array` de la bibliothèque `numpy`.

```
>>> from numpy import array
>>> M = array([[1.,2.,3.],[4.,5.,6.],[7.,8.,9.],[10.,11.,12.]])
>>> M
array([[  1.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
```

On remarquera notamment que les matrices sont décrites ligne par ligne. On peut accéder à un coefficient de la matrice par un double indice.

```
>>> M[0,0]
1.0
>>> M[1,2]
6.0
```

Il est aussi possible d'extraire une ou des lignes

```
>>> M[[0],:]
array([[ 1.,  2.,  3.]])
>>> M[[1,2],:]
array([[ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
```

De même, il est possible d'extraire une ou plusieurs colonnes d'une matrice.

```
>>> M[:,[1]]
array([[  2.],
       [  5.],
       [  8.],
       [ 11.]])
>>> M[:,[0,2]]
array([[  1.,   3.],
       [  4.,   6.],
       [  7.,   9.],
       [ 10.,  12.]])
```

Remarque 3.0.5. Il est aussi possible d'extraire une sous-matrice de M.

On peut obtenir les dimensions de d'une matrice par la méthode `shape`.

```
>>> M.shape
(4, 3)
```

Remarque 3.0.6. Attention, les commandes suivantes extraient bien une ligne ou une colonne d'une matrice, mais ne renvoient pas un résultat sous forme de matrice, mais de vecteurs.

```
>>> M[0,:]
array([ 1.,  2.,  3.])
>>> M[:,0]
array([  1.,   4.,   7.,  10.])
```

C'est toutefois très pratique pour effectuer des opérations sur les lignes et les colonnes de M.

```
>>> L = array([42.,42.,42.])
>>> M[0,:] = M[0,:] + L
>>> M
array([[ 43.,  44.,  45.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
```

Les opérations `+`, `-`, `*`, `/` (*etc.*) sont réalisées coefficient par coefficient. On peut aussi réaliser des produits matriciels par la méthode `.dot()`

```
>>> M = array([[1.,2.,3.],[4.,5.,6.]])
>>> M
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> X = array([[1.],[-1.],[0.]])
>>> X
```

```
array([[ 1.],
       [-1.],
       [ 0.]])
>>> N = array([[1.,2.],[3.,4.]])
>>> N
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> M.dot(X)
array([[-1.],
       [-1.]])
>>> N.dot(M)
array([[  9.,  12.,  15.],
       [ 19.,  26.,  33.]])
>>> N.dot(M).dot(X)
array([[-3.],
       [-7.]])
```

Enfin, il y a deux fonctions très pratiques de création de matrices.

```
>>> from numpy import zeros, eye
>>> zeros((4,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> eye(4,3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
```

Remarque 3.0.7. Attention, les données d'un objet de type `array` sont homogènes (*i.e.* elles doivent toutes être du même type).

```
>>> M = array([[1,2,3],[4,5,6],[7,8,9]])
>>> M[0,0] = M[0,0] / 2
>>> M
array([[0, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

On fera donc particulièrement attention à ne manipuler que des matrices et vecteurs de flottants.

4 Implantation

A et b sont des matrices, données sous forme d'un tableau bidimensionnel (type `array` de `numpy`). La structure `array` a plusieurs avantages :

— les opérations d'ajout de lignes et de multiplication d'une ligne par un scalaire sont déjà disponibles (sinon, il faut les programmer) ;

— les extractions de lignes sont faciles (on obtient alors un tableau unidimensionnel).

Remarque 4.0.8. Il existe aussi un type `matrix`. Adapté pour des manipulations de matrices mais pas très pratique pour l'extraction de lignes

On commence bien sûr par charger la bibliothèque `numpy`

```
from numpy import array, zeros
```

Pour chercher un pivot sur la colonne `i`, on effectue une recherche de maximum classique.

```
def cherche_pivot(A, i):
    """Cherche et renvoie un j tel que abs(A[j][i]) est maximal, avec j>=i"""
    n = len(A)
    best = i
    for j in range(i+1, n):
        # Inv : pour tout k <= j, abs(A[best][i]) >= abs(A[k][i])
        if abs(A[j,i]) > abs(A[best,i]):
            best = j
    return best
```

On réalise facilement l'échange de deux lignes.

```
def echange_lignes(A, i, j):
    """Échange les lignes i et j de la matrice A"""
    A[i,:],A[j,:] = A[j,:].copy(), A[i,:].copy()
    return None
```

Attention, la copie est nécessaire ici, sans quoi il y a une erreur due aux alias (les objets de type `array` sont mutables).

On peut réaliser la phase de descente.

```
def descente(A,b):
    """Phase de descente de la méthode du pivot pour résoudre Ax = b.
    Préconditions : A et b sont de type array,
                    A est inversible,
                    b a même nombre de lignes que A.
    Attention: cette fonction modifie A et b."""
```

```
    n = len(A)
    for i in range(n-1):
        ip = cherche_pivot(A, i)
        # on met en place la ligne du pivot :
        echange_lignes(A, i, ip)
        echange_lignes(b, i, ip)
        p = A[i, i] # le pivot
        for j in range(i+1, n):
            alpha = - A[j,i] / p # Coefficient multiplicateur
            b[j,:] = b[j,:] + alpha * b[i,:]
            A[j,:] = A[j,:] + alpha * A[i,:]
    return None
```

Remarque 4.0.9. Attention, une erreur fréquente est d'écrire la chose suivante.

```
        p = A[i, i] # le pivot
        for j in range(i+1, n):
            A[j,:] = A[j,:] - (A[j,i] / p) * A[i,:]
            b[j,:] = b[j,:] - (A[j,i] / p) * b[i,:]
```

Pourquoi ?

La phase de remontee se fait explicitement.

```
def remontee(U,B):
    """Résout le système UX = b.
    Préconditions: U triangulaire supérieure
                   b a autant de lignes que U."""
    n, m = B.shape
    X = zeros((n, m))
    for i in range(n):
        # Invariant X[n-i:] est correct
        s = U[n-1-i, n-i:].dot(X[n-i:])
        X[n-1-i] = (B[n-1-i] - s) / U[n-1-i, n-1-i]
    return X
```

Il n'y a plus qu'effectuer tout cela d'affilée.

```
def resout(A,b):
    """Applique la méthode du pivot pour résoudre Ax = b.
    Renvoie la solution x trouvée.
    Préconditions : A et b sont de type array,
                    A est inversible,
                    b a même nombre de lignes que A."""
```

```
    n = len(A)
    # On copie A et b
    A_, b_ = A.copy(), b.copy()
    descente(A_,b_)
    return remontee(A_,b_)
```

5 Complexité temporelle

Étudions le coût de l'algorithme du pivot.

Phase de remontée

Dans le cas d'une matrice triangulaire, le calcul de la dernière composante de la solution requiert une division, la précédente une division, une multiplication, une soustraction, ..., la première composante une division, $n-1$ multiplications et $n-1$ soustractions.

Au total : n divisions, $n(n-1)/2$ multiplications et $n(n-1)/2$ soustractions.

Ainsi, la phase de remontée a une complexité temporelle en $\Theta(n^2)$.

Phase de descente

Pour obtenir une matrice triangulaire, à l'étape i :

- on cherche le pivot ($n-i-1$ comparaisons, calculs de valeurs absolues, lectures dans un tableau *etc.*) ;
- on échange deux lignes (n flottants à échanger) ;
- pour chacune des $n-i-1$ dernières lignes, on effectue une multiplication de la ligne i avant de soustraire le résultat.

Il est clair que le nombre d'opérations avant d'arriver à une matrice triangulaire est un $O(n^3)$ et que c'est un $\Theta(n^3)$ (les $n/2$ premières étapes ont un coût supérieur à $n^3/8$).

Cas général

L'algorithme du pivot s'effectue en $\Theta(n^3)$ opérations et est donc relativement coûteux. Il est ici du même ordre que le produit (naïf) de deux matrices ($\Theta(n^3)$).

On verra plus loin des améliorations possibles.

6 Problèmes de stabilité numérique

Les calculs n'étant pas faits de façon exacte mais approchée, des problèmes peuvent survenir.

6.1 Incapacité à inverser une matrice

La matrice suivante est inversible :

$$\begin{pmatrix} 10^{20} & 10^{20} & 1 \\ 10^{19} & 1 & 0 \\ 10^{19} & 0 & 0 \end{pmatrix}.$$

Après une étape de pivot, on arrive à :

$$\begin{pmatrix} 10^{20} & 10^{20} & 1 \\ 0 & 1-10^{19} & -1 \\ 0 & -10^{19} & -1 \end{pmatrix}.$$

Mais, en machine, $1-10^{19}$ va être arrondi (à la même valeur que -10^{19}), et on aura donc la matrice

$$\begin{pmatrix} 10^{20} & 10^{20} & 1 \\ 0 & -10^{19} & -1 \\ 0 & -10^{19} & -1 \end{pmatrix}.$$

Résultat : les deux dernières lignes sont égales et la matrice n'est plus inversible.

6.2 Capacité à inverser des matrices non inversibles

Considérons une matrice de la forme

$$\begin{pmatrix} a & b \\ a & b \end{pmatrix},$$

avec $a \neq 0$.

Après une étape de pivot, on obtient :

$$\begin{pmatrix} a & b \\ 0 & b-(b/a)\times a \end{pmatrix}.$$

Si, en raison d'un arrondi, $b-(b/a)\times a$ ne donne pas 0, on obtient une matrice triangulaire sans zéro sur sa diagonale, donc inversible.

(en flottant ce problème se produit par exemple avec $b = 0,9999$ et $a = 1,9999$).

6.3 Conditionnement

Considérons la matrice $M = ((i+j-1)^{-1})_{1\leq i,j\leq 5}$ (matrice de Hilbert) et cherchons à résoudre $MX = B_1$ et $MX = B_2$ pour deux valeurs proches de B_1 et B_2.

```
n = 5
M = array([[ 1/(i+j+1.) for j in range(n)] for i in range(n)])
u0 = array([[-0.76785474],
       [-0.44579106],
```

```
       [-0.32157829],
       [-0.25343894],
       [-0.20982264]])
s0 = resout(M, u0)
u1 = array([[-0.76784856],
       [-0.44590775],
       [-0.32107213],
       [-0.25420613],
       [-0.20944639]])
s1 = resout(M, u1)
```

Résultat :

```
>>> s0
array([[-0.4900022],
       [-0.2844282],
       [-0.2054472],
       [-0.1613528],
       [-0.1340892]])
>>> u1 / u0
array([[ 0.99999195],
       [ 1.00026176],
       [ 0.99842601],
       [ 1.00302712],
       [ 0.99820682]])
>>> s1
array([[   1.3877308],
       [ -35.7756354],
       [ 153.7403826],
       [-233.496746 ],
       [ 114.2981532]])
>>> s1 / s0
array([[   -2.83209096],
       [  125.78090147],
       [ -748.32065172],
       [ 1447.1192691 ],
       [ -852.40387145]])
```

Le problème n'est pas dû au calcul numérique : il est inhérent à la matrice M.

Étant donné un vecteur x, on peut définir sa norme de plusieurs façons. Pour fixer les idées, pour $x = (x_1, \ldots, x_n) \in \mathbb{R}^{\mathbb{N}}$, introduisons sa *norme 2* :

$$\|x\| = \sqrt{x_1^2 + \ldots + x_n^2}.$$

Étant donné deux vecteurs $x \neq 0$ et x', on dit que l'erreur relative commise en prenant x' au lieu de x est $\frac{\|x-x'\|}{\|x\|}$.

On dit que M^{-1} est mal conditionnée car une petite erreur relative sur x peut se traduire par une grande erreur relative sur $M^{-1}x$.

Plus précisément, posons

$$E_M = \left\{ \frac{\|Mx\|}{\|x\|} \;\middle|\; x \neq 0 \right\} = \{\, \|Mx\| \mid \|x\| = 1 \,\}.$$

Le *conditionnement* c_M de la matrice M est la valeur

$$c_M = \frac{\sup E_M}{\inf E_M} = \frac{\max E_M}{\min E_M}$$

Remarque 6.3.1. On a aussi

$$c_M = \max E_M \times \max E_{M^{-1}} = c_{M^{-1}}$$

et $\max E_M$ est souvent noté $|||M|||$.

Alors, pour tout $x \neq 0$ et tout x' :

$$\frac{\|Mx - Mx'\|}{\|Mx\|} \leq c_M \times \frac{\|x - x'\|}{\|x\|}$$

et cette borne est serrée :

$$\exists x \neq 0,\ \exists x',\ \frac{\|Mx - Mx'\|}{\|Mx\|} = c_M \times \frac{\|x - x'\|}{\|x\|}$$

Donc si le conditionnement de M (donc de M^{-1}) est grand, une petite erreur relative sur x peut donner une grande erreur relative sur $M^{-1}x$.

7 Extensions aisées de l'algorithme de Gauss-Jordan

7.1 Résolution simultanée de plusieurs systèmes

On peut résoudre plusieurs systèmes $Ax = b_1$, $Ax = b_2$, … $Ax = b_p$ en utilisant l'algorithme avec un b matriciel dont les colonnes sont b_1, …, b_p. On aura alors pour solution $x \in \mathscr{M}_{n,p}(\mathbb{R})$, la colonne $j \in [\![1, n]\!]$ de x sera alors la solution du système $Ax = b_j$.

7.2 Inversion de matrices

Si on prend pour membre droit I_n, alors le x trouvé sera A^{-1}.

8 Méthode des moindres carrés

8.1 Cadre

Dans beaucoup de contextes scientifiques, il est fréquent qu'on soit dans la situation suivante.

— On modélise qu'une valeur b dépend linéairement de paramètres $(\alpha_1, \ldots, \alpha_q)$: il existe des constantes C_1, …, C_q telles que pour toute valeur des paramètres et toute valeur de b associée

$$b = \sum_{k=1}^{q} C_k \alpha_k = \begin{pmatrix} \alpha_1 & \cdots & \alpha_q \end{pmatrix} \times \begin{pmatrix} C_1 \\ \vdots \\ C_q \end{pmatrix}.$$

— On a mesuré des valeurs b_1, …, b_p et $(a_{11}, \ldots, a_{1q})$, …, $(a_{p1}, \ldots, a_{pq})$ les valeurs associées des paramètres.

— On aimerait « connaître » C_1, …, C_q.

On a dispose de p mesures pour estimer q paramètres.

On suppose $p \geq q$ (sinon on n'arrivera pas à connaître tous les paramètres sans effectuer d'hypothèses supplémentaires). En fait, on demande même plutôt $p > q$ (voire même de plusieurs facteurs), pour voir si ce modèle linéaire est réaliste, pour essayer de réduire l'effet des erreurs (aléatoires) de mesure *etc*.

Il s'agit de résoudre le système linéaire

$$\begin{pmatrix} a_{11} & \cdots & a_{1q} \\ \vdots & & \vdots \\ a_{p1} & \cdots & a_{pq} \end{pmatrix} \times \begin{pmatrix} C_1 \\ \vdots \\ C_q \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix},$$

d'inconnues C_1, …, C_q.

En pratique, on n'arrivera pas à résoudre ce système, pour plusieurs raisons.

1. Erreurs de mesure : il n'y a pas de solution alors que la relation était correcte.
2. La modélisation linéaire n'est en général qu'une approximation (on parle bien de « modèle »).

Pour s'en sortir, on change de point de vue :

Ne pas chercher à *résoudre*

$$Ax - b$$

mais plutôt à *minimiser*

$$\|Ax - b\|.$$

On prendra la norme 2 (c'est le plus facile et les résultats ont souvent une interprétation naturelle). Ainsi, on minimise le carré des écarts de Ax à b. D'où le nom : «Méthode des moindres carrés».

8.2 Résolution par projection orthogonale

Trouver x tel que $\|Ax - b\|$ soit minimale revient à trouver un élément de ImA dont la distance à b est minimale.

Or la distance de b à un sous-espace vectoriel en dimension finie, est atteinte en le projeté orthogonal de b sur ce sev. Notons

$$\hat{b} = Ax_0$$

le projeté orthogonal de b sur ImA. Alors pour tout x,

$$(Ax|b - \hat{b}) = 0,$$

soit

$$x^T A^T (b - Ax_0) = 0,$$

i.e.

$$A^T (b - Ax_0) = 0.$$

Ainsi, le vecteur x_0 recherché est la solution du système

$$A^T A x_0 = A^T b.$$

Sous des hypothèses raisonnables, $A^T A$ est inversible donc il existe une unique solution.

8.3 Exemple

On note t^C la température en degrés Celsius et t^F celle en degrés Farenheit. On sait qu'il existe α, β tels que $t^F = \alpha + \beta t^C$.

On dispose de deux thermomètres : un en degrés Celsius, l'autre en degrés Farenheit, et l'on veut déterminer α et β.

On effectue 14 relevés de tempéraure : $a = (t_1^C, \cdots, t_{14}^C)$ et $b = (t_1^F, \cdots, t_{14}^F)$.

On note A la matrice 14×2 dont la première colonne ne contient que des 1, et la seconde colonne est b.

En théorie

$$A \times \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = b,$$

mais la relation n'est pas exacte ici (à cause des erreurs de mesure, notamment).

On cherche donc $X = \begin{pmatrix} x \\ y \end{pmatrix}$ tel que $\|AX - b\|$ soit minimale. D'après la partie précédente, on sait qu'il convient de résoudre le système

$$A^T A \times X = A^T b.s$$

```
>>> from numpy import array, transpose
>>> A = array([[1. ,  -40.],
...            [1. ,  -35.5],
```

```
...             [1. ,   -30.5],
...             [1. ,   -25.5],
...             [1. ,  -20.5],
...             [1. ,   -15.5],
...             [1. ,   -10.5],
...             [1. ,  -5.5],
...             [1. ,   -0.5],
...             [1. ,   4.5],
...             [1. ,  19.5],
...             [1. ,   34.5],
...             [1. ,   44.5],
...             [1. ,   49.5]])
>>> b = array([[-39.67],
...             [-32.68],
...             [-23.81],
...             [-13.61],
...             [-3.76],
...             [5.38],
...             [12.50],
...             [24.28],
...             [32.57],
...             [38.78],
...             [66.65],
...             [93.18],
...             [111.88],
...             [121.52]])
>>> At = transpose(A)
>>> At
array([[  1. ,   1. ,   1. ,   1. ,   1. ,   1. ,   1. ,   1. ,   1. ,
          1. ,   1. ,   1. ,   1. ,   1. ],
       [-40. , -35.5, -30.5, -25.5, -20.5, -15.5, -10.5,  -5.5,  -0.5,
         4.5,  19.5,  34.5,  44.5,  49.5]])
>>> At.dot(A)
array([[    14.  ,    -31.5 ],
       [   -31.5 ,  11263.25]])
>>> At.dot(b)
array([[   393.21],
       [ 19215.61]])
```

Il nous reste à résoudre le système

$$\begin{pmatrix} 14 & -31.5 \\ -31.5 & 11236.25 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 393.21 \\ 19215.61 \end{pmatrix}.$$

Utilisons les outils de `numpy` pour résoudre ce système.

```
>>> from numpy.linalg import solve
>>> X = solve(At.dot(A),At.dot(b))
>>> X
array([[ 32.12719276],
       [  1.7958952 ]])
```

Ainsi, $t^F \approx 32.1 + 1.8t^C$ (voir figure 3).

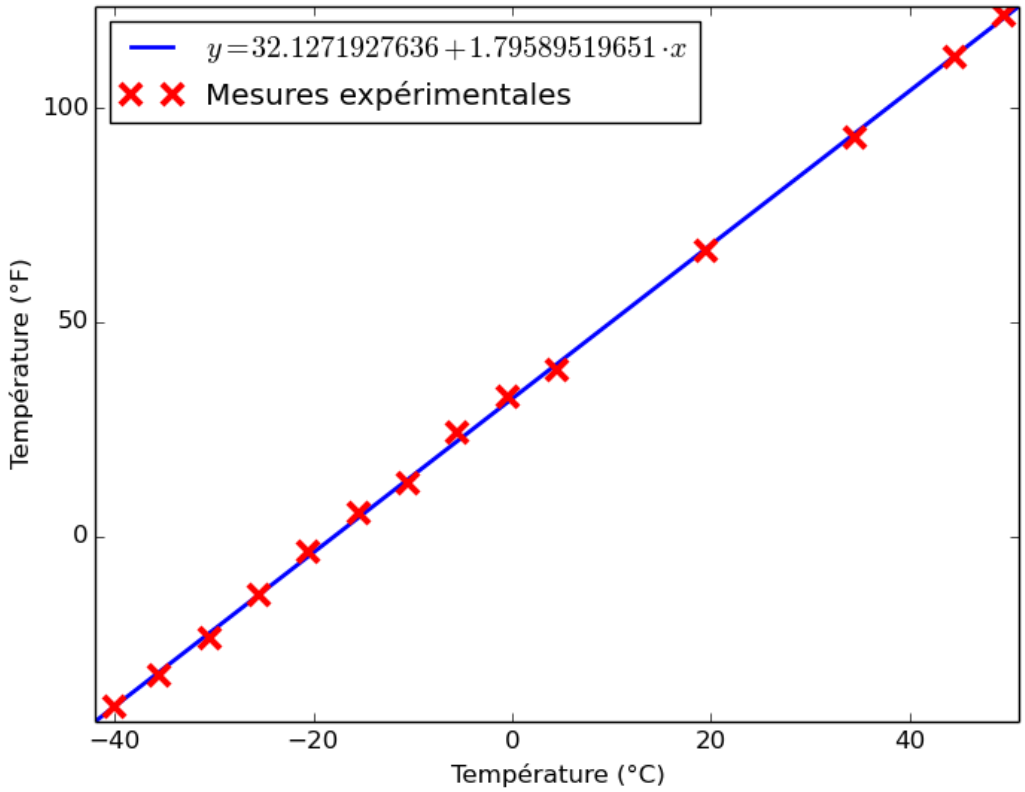

FIGURE 3 – Régression linéaire par moindres carrés entre les degrés Celsius et Farenheit.

8.4 Inconvénient

La résolution par transposition amplifie les erreurs : $(A + \delta A)^T(A + \delta A) = A^TA + \delta A^TA + A^T\delta A + \delta A^T\delta A \cdots$.

Il existe des méthodes plus stables : utilisation de matrices orthogonales, de pseudo-inverses, décomposition QR ...

9 Regard critique

9.1 Résumons

On travaille sur une matrice carrée A de taille n.

1. On veut parfois des valeurs de n élevées : plusieurs milliers.
2. La complexité de ces méthodes est en général de l'ordre de la méthode du pivot : $\Theta(n^3)$.
3. C'est l'ordre de grandeur du coût de la multiplication naïve.

Peut-on faire mieux ?

— oui, il y a des algorithmes un peu plus efficaces pour obtenir les mêmes décompositions (mais c'est délicat).

— mais bien souvent en informatique, il vaut mieux se poser la question de la pertinence de ce qu'on fait.

9.2 Cadre physique

1. interactions seulement entre des points proches
2. donc la matrice A est creuse (une majorité de zéros).

On peut améliorer la représentation des matrices et les algorithmes d'addition et de multiplication pour en tirer parti.

Problème : le pivot de Gauss, aussi bien que les méthodes « efficaces » sur des matrices quelconques, basées sur des décompositions des matrices (LU, PLU, QR, ...), ont tendance à remplir les matrices.

9.3 Méthodes plus adaptées aux matrices creuses

Pour résoudre $Ax = b$ pour une matrice creuse,
Utilisation de méthodes itératives pour les matrices creuses :

1. on part d'une solution approchée ;
2. on l'améliore ;
3. revenir au point précédent tant que nécessaire.

Intérêt de ces méthodes :

1. Pas besoin de calculer de nouvelles matrices (on reste sur des matrices creuses) ;
2. On améliore peu à peu une solution : impact des erreurs de calcul a priori plus faible.

10 Exercices

Dans chaque exercice, on demande une valeur approchée du résultat.

Exercice 10.0.1. Résoudre le système

$$\begin{pmatrix} 5 & 8 & -2 \\ 3 & 1 & 5 \\ 0 & -2 & 6 \end{pmatrix} \times X = \begin{pmatrix} 21 \\ 16 \\ 10 \end{pmatrix}.$$

Exercice 10.0.2. On pose

$$A = \begin{pmatrix} 3 & -2 & 5 \\ -4 & 1 & 1 \\ 2 & 3 & -2 \end{pmatrix}.$$

En effectuant un seul calcul, résoudre simultanément les systèmes :

$$AX = \begin{pmatrix} 20 \\ -2 \\ -7 \end{pmatrix}, \quad AX = \begin{pmatrix} -21 \\ 23 \\ -1 \end{pmatrix}, \quad AX = \begin{pmatrix} -12 \\ 17 \\ 4 \end{pmatrix}, \quad AX = \begin{pmatrix} 6 \\ -2 \\ 3 \end{pmatrix}.$$

Exercice 10.0.3. Inverser la matrice

$$A = \begin{pmatrix} 5 & -3 & 2 & 1 & -1 \\ 3 & 6 & 8 & 1 & -3 \\ 5 & 6 & 3 & 0 & 2 \\ 4 & 6 & 2 & 8 & 3 \\ -6 & 3 & 5 & -1 & -2 \end{pmatrix}.$$

Exercice 10.0.4. On considère les points de coordonnées

$$M_1 \begin{pmatrix} -5 \\ 11,67 \end{pmatrix}, \quad M_2 \begin{pmatrix} -2 \\ 4,52 \end{pmatrix}, \quad M_3 \begin{pmatrix} 1 \\ -0,15 \end{pmatrix}, \quad M_4 \begin{pmatrix} 2 \\ -3,31 \end{pmatrix}.$$

On cherche à ajuster une droite affine sur le nuage de points (M_1, M_2, M_3, M_4) par le critère des moindres carrés. Avec

$$A = \begin{pmatrix} 1 & -5 \\ 1 & -2 \\ 1 & 1 \\ 1 & 2 \end{pmatrix}, \quad X = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} 11,67 \\ 4,52 \\ -0,15 \\ -3,31 \end{pmatrix},$$

déterminer X minimisant la quantité

$$\|AX - B\|.$$

Produire une figure superposant les points (M_1, M_2, M_3, M_4) et la droite d'équation $y = \alpha + \beta x$.

Exercice 10.0.5. On considère les points de coordonnées

$$M_1\begin{pmatrix}-2\\7,62\end{pmatrix},\quad M_2\begin{pmatrix}-1\\3,87\end{pmatrix},\quad M_3\begin{pmatrix}0\\0,94\end{pmatrix},\quad M_4\begin{pmatrix}1\\1,56\end{pmatrix},\quad M_5\begin{pmatrix}2\\2,66\end{pmatrix}.$$

On cherche à ajuster une courbe polynomiale de degré 2 sur le nuage de points $(M_1, M_2, M_3, M_4, M_5)$ par le critère des moindres carrés. Avec

$$A=\begin{pmatrix}1&-2&4\\1&-1&1\\1&0&0\\1&1&1\\1&2&4\end{pmatrix},\quad X=\begin{pmatrix}\alpha\\\beta\\\gamma\end{pmatrix}\quad\text{et}\quad B=\begin{pmatrix}7,62\\3,87\\0.94\\1,56\\2,66\end{pmatrix},$$

déterminer X minimisant la quantité

$$\|AX-B\|.$$

Produire une figure superposant les points $(M_1, M_2, M_3, M_4, M_5)$ et la courbe d'équation $y=\alpha+\beta x+\gamma x^2$.