Chapitre 10 Résolution numérique d'un système linéaire

1. Python et le calcul matriciel

Le module numpy contient les éléments indispensables à la modélisation des vecteurs, matrices et plus généralement des tableaux multidimensionnels. Pour cela, numpy fournit le type *ndarray*, qui, bien que très proche sur le plan syntaxique du type *list* , diffère de ce dernier sur plusieurs points importants :

- la taille des tableaux numpy est fixée au moment de la création et ne peut plus être modifiée par la suite [1] ;
- les tableaux numpy sont *homogènes*, c'est-à-dire constitués d'éléments de même type.

En contrepartie, l'accès aux éléments d'un tableau numpy est incomparablement plus rapide, ce qui justifie pleinement leur usage pour manipuler des matrices de grandes tailles.

Par la suite, nous supposerons avoir écrit au début de chacun des scripts de ce chapitre l'instruction :

```
import numpy as np
```

ce qui nous permettra de visualiser aisément les functions de ce module : elles seront préfixées par `.np`.

1.1 Création de tableaux

On utilise en général la fonction `array` pour former un tableau à partir de la liste de ses éléments (ou de la liste des listes pour une matrice bi-dimensionnelle). Par exemple, pour créer une matrice 3×4 à partir de ses éléments on écrira :

```
>>> a = np.array([[3, 6, 2, 0], [2, -3, 5, -1], [0, 6, 1, -1]])
>>> a
array([[ 3,  6,  2,  0],
       [ 2, -3,  5, -1],
       [ 0,  6,  1, -1]])
```

La matrice que nous venons de créer est une matrice à coefficients entiers car tous ses éléments sont des entiers ; si on change l'un des coefficients, le nouveau coefficient sera au préalable converti en entier, ce qui peut provoquer une confusion :

```
>>> a[0, 0] = 1.25
>>> a
array([[ 1,  6,  2,  0],
       [ 2, -3,  5, -1],
       [ 0,  6,  1, -1]])
```

Si la liste des éléments est hétérogène, certains seront automatiquement convertis. Par exemple, si la liste des coefficients contient des éléments de type *float* et de type *int*, ces derniers seront convertis en flottants :

```
>>> a = np.array([[3., 6, 2, 0], [2, -3, 5, -1], [0, 6, 1, -1]])
>>> a
array([[ 3.,  6.,  2.,  0.],
       [ 2., -3.,  5., -1.],
       [ 0.,  6.,  1., -1.]])
```

Aussi, pour éviter toute ambiguïté il est préférable de préciser lors de la création le type des éléments souhaités avec le paramètre optionnel `dtype` (pour *data type*) :

```
>>> b = np.array([1, 7, -1, 0, -2], dtype=float)
>>> b
array([ 1.,  7., -1.,  0., -2.])
```

1. On peut tout au plus les *redimensionner*, comme cela sera expliqué plus loin.

Les types autorisés sont les suivants :

`bool` (booléens), `int` (entiers), `float` (flottants), `complex` (complexes)

plus un certain nombre de types dont nous n'auront que peu ou pas d'usage : entiers signés sur 8-16-32-64 bits, entiers non signés sur 8-16-32-64 bits, flottants sur 8-16-32-64 bits, ...

Remarque. Attention, en réalité le data type *int* utilisé par NUMPY ne correspond pas au type *int* de PYTHON ; il s'agit du type *int64* des entiers signés sur 64 bits (codés par complémentation à deux). Autrement dit, les entiers NUMPY sont restreints à l'intervalle $[\![-2^{63}, 2^{63}-1]\!]$.

Pourquoi tant de types différents ?
Prenons le cas de la représentation matricielle d'une image non compressée 1600×1200, chaque pixel étant représenté par un triplet RGB permettant de reconstituer une couleur par synthèse additive. Autrement dit, avec NUMPY une image est modélisée par un tableau tri-dimensionnel $1600 \times 1200 \times 3$.
Chaque composante primaire est décrite par un entier non signé codé sur 8 bits (= 1 octet), autrement dit un entier de l'intervalle $[\![0, 2^8-1]\!] = [\![0, 255]\!]$. Avec le data type *uint8* (entier non signé codé sur 8 bits) l'espace mémoire utilisé pour représenter cette image vaut $1600 \times 1200 \times 3 = 5\,760\,000$ octets soit 5,5 Mio. Si on utilisait pour chacune des composantes des entiers codés sur 64 bits (= 8 octets) l'espace mémoire nécessaire serait huit fois plus important, soit 44 Mio.

Dorénavant, et sauf mention explicite du contraire, nous supposerons les tableaux NUMPY remplis à l'aide du type *float* (correspondant sur les machines actuelles au data type *float64* des nombres flottants représentés sur 64 bits).

1.2 Coupes

On accède à un élément d'un tableau NUMPY exactement de la même façon que pour une liste : à partir de son indice.

```
>>> b[0]
1.0
```

Pour les tableaux multidimensionnels, outre la syntaxe usuelle `a[i][j]` il est aussi possible d'utiliser la syntaxe `a[i, j]` :

```
>>> a[0, 0]
3.0
```

Le slicing suit la même syntaxe que pour les listes PYTHON. On retiendra surtout la syntaxe pour obtenir une vue d'une colonne ou d'une ligne d'une matrice :

```
>>> a[2]                        # 3e ligne de la matrice a
array([ 0.,  6.,  1., -1.])

>>> a[:, 2]                     # 3e colonne de la matrice a
array([ 2.,  5.,  1.])
```

Attention cependant, à la différence des listes PYTHON, *une coupe d'un tableau* NUMPY *ne crée pas un nouveau tableau* (la terminologie officielle parle dans ce cas de « vue » plutôt que de coupe). Pour copier un tableau NUMPY il est donc indispensable d'utiliser la méthode `copy`.

```
>>> a1 = a[2]                    # a1 est une vue de la 3e ligne de a

>>> a2 = a[2].copy()             # a2 est une copie de la 3e ligne de a

>>> a[2, 0] = 7

>>> a1                           # la modification de a se répercute sur a1
array([ 7.,  6.,  1., -1.])

>>> a2
array([ 0.,  6.,  1., -1.])     # la modification de a ne se répercute pas sur a2
```

Cette différence avec les listes Python peut s'avérer problématique lorsqu'il s'agit d'effectuer des opération élémentaires sur les lignes ou les colonnes d'une matrice. En particulier, *la syntaxe* `a[i], a[j] = a[j], a[i]` *n'échange pas les lignes* $(i+1)$ *et* $(j+1)$.

```
>>> a[0], a[1] = a[1], a[0]

>>> a
array([[ 2., -3.,  5., -1.],
       [ 2., -3.,  5., -1.],
       [ 7.,  6.,  1., -1.]])
```

Il est donc sage lorsqu'on manipule des tableaux numpy de s'interdire l'affectation simultanée et lui préférer une syntaxe telle que :

```
b = a[0].copy()
a[0] = a[1]
a[1] = b
```

voire, pour éviter tout coût spatial, à permuter terme par terme chaque élément des lignes à échanger :

```
for j in range(a.shape[1]):
    a[0, j], a[1, j] = a[1, j], a[0, j]
```

« fancy indexing »

On peut néanmoins accéder au contenu d'un tableau numpy tant en lecture qu'en écriture en utilisant le *fancy indexing*. Ici les positions dans un tableau ne procèdent plus nécessairement par des coupes mais peuvent être données dans un ordre quelconque : si `a` est un tableau et `l` une liste d'indices, alors `a[l]` renvoie le tableau formé des éléments $a[i]$ où i est dans `l`. Il est donc possible d'échanger les lignes i et j du tableau a en écrivant :

```
a[[i, j]] = a[[j, i]]
```

et d'inverser les colonnes i et j de ce tableau par :

```
a[:, [i, j]] = a[:, [j, i]]
```

1.3 Redimensionnement d'un tableau

La méthode `shape` permet de connaître les dimensions d'un tableau de type *ndarray* :

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], dtype=float)
>>> a.shape
(3, 4)
```

Modifier cet attribut permet de redimensionner le tableau, à condition bien sur que le nombre total d'éléments du tableau reste inchangé. Les 12 éléments qui composent le tableau `a` peuvent donc être ré-ordonnés pour former un tableau de taille 2×6, voire un vecteur de taille 12 :

```
>>> a.shape = 2, 6
>>> a
array([[  1.,   2.,   3.,   4.,   5.,   6.],
       [  7.,   8.,   9.,  10.,  11.,  12.]])

>>> a.shape = 12
>>> a
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.])
```

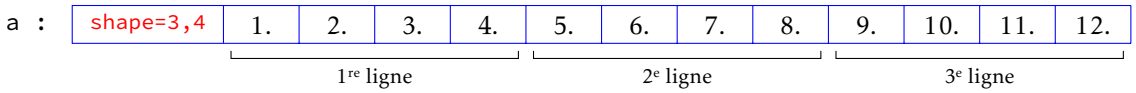
C'est l'occasion d'observer que numpy différencie les tableaux uni-dimensionnels (les vecteurs) des tableaux bi-dimensionnels, *même lorsque ces derniers ne comportent qu'une seule ligne ou une seule colonne* :

```
>>> a = np.array([1, 2, 3], dtype=float)
>>> a                                          # a est un vecteur
array([ 1.,  2.,  3.])

>>> a.shape = 1, 3
>>> a                                          # a est une matrice 1 x 3
array([[ 1.,  2.,  3.]])

>>> a.shape = 3, 1
>>> a                                          # a est une matrice 3 x 1
array([[ 1.],
       [ 2.],
       [ 3.]])
```

On notera que redimensionner une matrice est une opération de coût constant. L'explication est simple : quelle que soit la forme de la matrice, ses coefficients sont stockés en mémoire dans des cases *contiguës* et le format de la matrice dans un emplacement spécifique. Par exemple, le script suivant :

```
>>> a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], dtype=float)
>>> a.shape = 3, 4
>>> a
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
```

crée un espace mémoire de 12 cases des 64 bits chacune ainsi qu'un espace mémoire supplémentaire contenant différentes informations, en particulier que a est un tableau bi-dimensionnel de 3 lignes et de 4 colonnes.



Connaître la case dans laquelle se trouve l'élément de rang (i, j) n'est dès lors que le résultat d'un simple calcul arithmétique : pour une matrice de n lignes et p colonnes il s'agit de la case de rang $i \times p + j$, et modifier les dimensions d'un tableau consiste tout simplement à changer cette formule, pas les emplacements en mémoire des éléments.

Remarque. Il existe aussi une méthode `reshape` qui crée une nouvelle matrice (les éléments sont donc copiés) en la redimensionnant si nécessaire (cette méthode a donc un coût proportionnel au nombre d'éléments de la matrice).

• Création de tableaux spécifiques

La fonction `zeros` permet de former des tableaux dont tous les coefficients sont nuls : le premier et seul argument obligatoire est un tuple qui précise la dimension du tableau [2] et un argument facultatif (par défaut égal à `float`) permet de fixer le type de données.

```
>>> np.zeros((2, 3), dtype=int)
array([[0, 0, 0],
       [0, 0, 0]])
```

La fonction `identity` construit la matrice d'identité d'ordre n :

```
>>> np.identity(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

La fonction `diag` appliquée à une liste renvoie la matrice diagonale formée à partir des coefficients de cette liste :

2. Notez que les parenthèses pour enclore ce tuple sont indispensables si le tableau est multi-dimensionnel.

```
>>> np.diag([1., 2., 3.])
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
```

1.4 Opérations élémentaires sur les lignes et les colonnes

Les fonctions usuelles qui sont définies au sein de NUMPY peuvent être appliquées à des tableaux : dans ce cas, chacun des coefficients de ce dernier se voient appliquer la fonction. C'est aussi le cas des opérateurs binaires : si $a = (a_{ij})$ et $b = (b_{ij})$ sont deux matrices $n \times p$ et $\oplus$ un opérateur binaire alors $a \oplus b$ calcule la matrice $(a_{ij} \oplus b_{ij})$.
Ainsi, pour calculer l'addition de deux matrices a et b il suffira d'écrire `a + b`. De même, pour calculer le produit d'un scalaire t par une matrice a il suffira d'écrire `t * a`. En revanche, *on se gardera bien de calculer le produit matriciel en écrivant* `a * b` car ce qui est ainsi calculé est le produit terme à terme de chacun des coefficients de ces deux matrices. Pour calculer le produit matriciel, il faut utiliser la fonction `dot(a, b)`.
On notera que si a est une matrice et x un vecteur on calcule le produit ax à l'aide de la fonction `dot`.

Ces opérations vont nous permettre d'appliquer aisément des opérations élémentaires sur les lignes d'une matrice :

– L'opération $L_i \leftarrow L_i + tL_j$ (ajout de tL_j à la ligne L_i) s'écrit :

```
a[i] = a[i] + t * a[j]
```

– L'opération $L_i \leftarrow tL_i$ (multiplication de L_i par t) s'écrit :

```
a[i] = t * a[i]
```

– L'opération $L_i \leftrightarrow L_j$ (permutation des lignes L_i et L_j) s'écrit [3] :

```
a[[i, j]] = a[[j, i]]
```

On prendra bien garde au fait que chacune de ces trois opérations élémentaires modifie physiquement la matrice a.

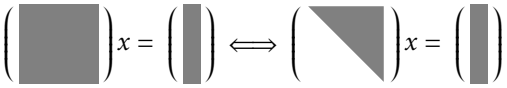
2. Méthode du pivot de GAUSS

La méthode du *pivot de* GAUSS est une méthode générale de résolution d'un système linéaire de la forme : $Ax = b$, où A est une matrice inversible. Elle repose sur l'observation faite qu'appliquer une opération élémentaire (O_i) sur les *lignes* d'une matrice équivaut à multiplier cette dernière *à gauche* par une certaine matrice inversible U_i.
Compte tenu de l'équivalence :

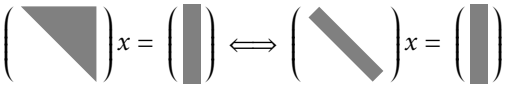
$$Ax = b \iff U_i Ax = U_i b$$

on constate qu'on ne modifie pas l'ensemble des solutions d'une équation linéaire en appliquant *les mêmes opérations élémentaires sur les lignes* de A et de b.
La méthode du pivot de GAUSS comporte trois étapes :

1. une première étape dite de *descente* qui consiste à transformer la matrice inversible A en une matrice A′ *triangulaire supérieure* tout en effectuant les mêmes opérations sur b ;



2. une deuxième étape dite de *remontée* qui consiste à transformer la matrice inversible A′ en une matrice A″ *diagonale* tout en effectuant les mêmes opérations sur b ;



3. une troisième et dernière étape de résolution du système linéaire, devenu trivial car diagonal.

3. Relire le paragraphe consacré au *fancy indexing*.

2.1 L'étape de descente

Celle-ci consiste à transformer progressivement la matrice A par opérations élémentaires sur les lignes tout en maintenant l'invariant :

$$\text{à l'entrée de la } j^{\text{e}} \text{ boucle,} \qquad \mathrm{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & \cdots & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & \cdots & \cdots & a_{2n} \\ \vdots & \ddots & \ddots & & & \vdots \\ 0 & \cdots & 0 & a_{jj} & \cdots & a_{jn} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{nj} & \cdots & a_{nn} \end{pmatrix}.$$

Dans le cas où la matrice A est de la forme ci-dessus, on procède en deux temps :

1. on recherche un pivot *non nul*[4] a_{ij} parmi $a_{jj}, a_{j+1,j}, \ldots, a_{nj}$ puis on permute les lignes L_i et L_j : $\mathrm{L}_i \leftrightarrow \mathrm{L}_j$;
2. à l'issue de cette première étape nous sommes assurés que désormais $a_{jj} \neq 0$; on l'utilise comme pivot pour remplacer tous les termes a_{ij} pour $i > j$ par des zéros à l'aide de l'opération élémentaire :

$$\forall i \in [\![j+1, n]\!], \qquad \mathrm{L}_i \leftarrow \mathrm{L}_i - \frac{a_{ij}}{a_{jj}} \mathrm{L}_j.$$

• Le choix du pivot

Mathématiquement, tous les pivots non nuls se valent. Il n'en est pas de même du point de vue numérique : diviser par un pivot dont la valeur absolue est trop faible par rapport aux autres coefficients du système conduit à des erreurs d'arrondi importantes. Nous allons illustrer ce phénomène à l'aide d'un exemple numérique.

Exemple. Supposons que les calculs sont effectués en virgule flottante dans le système décimal avec une mantisse de quatre chiffres et considérons le système :

$$\begin{cases} 0{,}003\,x + 59{,}14\,y = 59{,}17 \\ 5{,}291\,x - \ 6{,}13\,y = 46{,}78 \end{cases} \iff \begin{pmatrix} 3{,}000 \cdot 10^{-3} & 5{,}914 \cdot 10^{1} \\ 5{,}291 \cdot 10^{0} & -6{,}130 \cdot 10^{0} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5{,}917 \cdot 10^{1} \\ 4{,}678 \cdot 10^{1} \end{pmatrix}$$

dont la solution exacte est $x = 10$, $y = 1$.

Choisir $0{,}003$ pour pivot conduit à effectuer l'opération élémentaire : $\mathrm{L}_2 \leftarrow \mathrm{L}_2 - \lambda \mathrm{L}_1$ avec

$$\lambda = \frac{5{,}291}{0{,}003} = 1\,763{,}666\cdots \approx 1{,}764 \cdot 10^{3}$$

et conduit à résoudre le nouveau système :

$$\begin{pmatrix} 3{,}000 \cdot 10^{-3} & 5{,}914 \cdot 10^{1} \\ 0 & -1{,}043 \cdot 10^{5} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5{,}917 \cdot 10^{1} \\ -1{,}044 \cdot 10^{5} \end{pmatrix}$$

En effet, $1{,}764 \cdot 10^{3} \times 5{,}914 \cdot 10^{1} \approx 1{,}043 \cdot 10^{5}$ et $-6{,}130 \cdot 10^{0} - 1{,}043 \cdot 10^{5} \approx -1{,}043 \cdot 10^{5}$

$1{,}764 \cdot 10^{3} \times 5{,}917 \cdot 10^{1} \approx 1{,}044 \cdot 10^{5}$ et $4{,}678 \cdot 10^{1} - 1{,}044 \cdot 10^{5} \approx -1{,}044 \cdot 10^{5}$

L'étape de remontée qui sera expliquée plus loin consiste à effectuer l'opération élémentaire $\mathrm{L}_1 \leftarrow \mathrm{L}_1 - \mu \mathrm{L}_2$ pour obtenir un système diagonal, avec

$$\mu = \frac{5{,}914 \cdot 10^{1}}{-1{,}043 \cdot 10^{5}} \approx -5{,}670 \cdot 10^{-4}$$

et conduit à résoudre le système diagonal :

$$\begin{pmatrix} 3{,}000 \cdot 10^{-3} & 0 \\ 0 & -1{,}043 \cdot 10^{5} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -2{,}000 \cdot 10^{-2} \\ -1{,}044 \cdot 10^{5} \end{pmatrix}$$

car $5{,}670 \cdot 10^{-4} \times 1{,}044 \cdot 10^{5} \approx 5{,}919 \cdot 10^{1}$ et $5{,}917 \cdot 10^{1} - 5{,}919 \cdot 10^{1} \approx -2{,}000 \cdot 10^{-2}$.

Ainsi, la solution numérique obtenue est : $x \approx -6{,}667$ et $y \approx 1{,}001$. Même si l'erreur faite sur y est très faible (de l'ordre de $0{,}1\%$), sa propagation a un effet désastreux sur x (une erreur de l'ordre de 167%), à cause d'un pivot très faible.

4. L'inversibilité de la matrice A nous assure de son existence (voir votre cours de mathématiques).

Reprenons ces calculs, mais en choisissant cette fois 5,291 pour pivot, ce qui conduit à débuter par la permutation $L_1 \leftrightarrow L_2$ des deux lignes du système :

$$\begin{pmatrix} 5{,}291\cdot 10^{0} & -6{,}130\cdot 10^{0} \\ 3{,}000\cdot 10^{-3} & 5{,}914\cdot 10^{1} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4{,}678\cdot 10^{1} \\ 5{,}917\cdot 10^{1} \end{pmatrix}$$

On effectue cette fois l'opération $L_2 \leftarrow L_2 - \lambda L_1$ avec

$$\lambda = \frac{0{,}003}{5{,}291} \approx 5{,}670\cdot 10^{-4}$$

ce qui conduit à résoudre le système triangulaire :

$$\begin{pmatrix} 5{,}291\cdot 10^{0} & -6{,}130\cdot 10^{0} \\ 0 & 5{,}914\cdot 10^{1} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4{,}678\cdot 10^{1} \\ 5{,}914\cdot 10^{1} \end{pmatrix}$$

et enfin, l'opération $L_1 \leftarrow L_1 - \mu L_2$ avec $\mu = \dfrac{-6{,}130\cdot 10^{0}}{5{,}914\cdot 10^{1}} \approx -1{,}037\cdot 10^{-1}$ conduit au système diagonal :

$$\begin{pmatrix} 5{,}291\cdot 10^{0} & 0 \\ 0 & 5{,}914\cdot 10^{1} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5{,}291\cdot 10^{1} \\ 5{,}914\cdot 10^{1} \end{pmatrix}$$

qui fournit finalement les valeurs $x \approx 1{,}000\cdot 10^{1}$ et $y \approx 1{,}000\cdot 10^{0}$ qui coïncident avec les résultats théoriques.

Pivot partiel
Ces considérations nous amènent à choisir pour pivot *le plus grand en module* des coefficients $a_{jj}, a_{j+1,j}, \dots, a_{nj}$; c'est le choix du *pivot partiel* de Gauss.

Remarque. La méthode du pivot total consiste à cherche le plus grand en module des coefficients du bloc rectangulaire

$$\begin{bmatrix} a_{jj} & \cdots & a_{jn} \\ \vdots & & \vdots \\ a_{nj} & \cdots & a_{nn} \end{bmatrix}$$

Cependant, cela induit une difficulté algorithmique supplémentaire car pour amener ce pivot à l'emplacement (j, j) il est souvent nécessaire d'effectuer une permutation entre deux colonnes qui modifie l'ordre des inconnues pour le système linéaire à résoudre. Il faut donc garder trace de ces multiples permutations afin de renvoyer celles-ci dans le bon ordre à la fin du calcul.
En pratique, on constate que le gain de stabilité apporté par la recherche du pivot total n'est en général pas significatif, alors qu'il alourdit la programmation. C'est donc la recherche du pivot partiel qui est le plus couramment utilisé, et c'est celui que nous allons implémenter.

• Programmation de l'étape de descente

a) Rédiger une fonction `recherche_pivot(A, b, j)` qui détermine le coefficient a_{ij} le plus grand en module parmi $a_{jj}, \dots, a_{nj}$ puis qui permute les lignes L_i et L_j de A et de b.

b) Rédiger une fonction `elimination_bas(A, b, j)` qui effectue les éliminations successives des coefficients situés sous a_{jj}, en supposant ce coefficient non nul. On effectuera en parallèle les mêmes opérations sur b.

c) En déduire une fonction `descente(A, b)` qui par opérations élémentaires sur les lignes des matrices A et b réalise l'étape de descente de la méthode du pivot de Gauss.

2.2 L'étape de remontée

Celle-ci intervient lorsque la matrice A est triangulaire supérieure, les coefficients de la diagonale étant non nuls. On transforme progressivement la matrice A en maintenant l'invariant :

à l'entrée de la $(n-j+1)^{e}$ boucle, $\quad A = \begin{pmatrix} a_{11} & \cdots & a_{1j} & 0 & \cdots & 0 \\ 0 & \ddots & \vdots & \vdots & & \vdots \\ \vdots & \ddots & a_{jj} & 0 & & \vdots \\ 0 & \cdots & 0 & a_{j+1,j+1} & \ddots & \vdots \\ \vdots & & \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \cdots & 0 & a_{nn} \end{pmatrix}$.

et en utilisant a_{jj} comme pivot pour éliminer les coefficients $a_{1j},\dots,a_{j-1,j}$.

• Programmation de l'étape de remontée

a) Rédiger une fonction `elimination_haut(A, b, j)` qui effectue les éliminations successives des coefficients situés au-dessus de a_{jj}, en supposant ce coefficient non nul. On effectuera en parallèle les mêmes opérations sur b.

b) En déduire une fonction `remontee(A, b)` qui par opérations élémentaires sur les lignes des matrices A et b réalise l'étape de remontée de la méthode du pivot de Gauss.

2.3 Résolution du système linéaire

La troisième et dernière étape de la méthode consiste à résoudre le système diagonal obtenu après les deux étapes précédentes.

• Programmation de la méthode du pivot de Gauss

a) Rédiger une fonction `solve_diagonal(A, b)` qui prend en arguments une matrice diagonale inversible A et un vecteur b et qui retourne l'unique vecteur x solution de l'équation $Ax = b$.

b) En déduire une fonction `gauss(A, b)` qui retourne l'unique solution d'un système de Cramer $Ax = b$. On travaillera sur des copies des matrices A et b pour éviter de modifier physiquement ces dernières.

Étude de la complexité temporelle de la méthode
Les différentes fonctions qui ont été écrites ont pour coûts temporels respectifs :

- `recherche_pivot` a un coût en $O(n)$;
- `elimination_bas` a un coût en $O(n^2)$;
- `descente` a un coût en $O(n^3)$;
- `elimination_haut` a un coût en $O(n^2)$;
- `remontee` a un coût en $O(n^2)$;
- `solve_diagonal` a un coût en $O(n)$;

La méthode de Gauss dans son ensemble a donc un coût temporel en $O(n^3)$, à quoi s'ajoute un coût spatial en $O(n^2)$ si on travaille sur des copies des matrices A et B.

2.4 Applications de la méthode du pivot de Gauss

Calcul du déterminant
Une fois l'étape de descente terminée, la matrice obtenue A' est triangulaire donc le calcul de son déterminant est chose aisée. De plus, son déterminant est égal à $(-1)^k \det A$ où k est le nombre d'échanges qui ont été effectués entre deux lignes. Ceci fournit donc une méthode de calcul du déterminant de A.

a) Rédiger une fonction `determinant(A)` qui calcule le déterminant de la matrice A. On travaillera sur une copie de A pour ne pas modifier la matrice initiale.

Calcul de l'inverse
La méthode du pivot conduit à passer de la matrice A à la matrice I_n par une succession d'opérations élémentaires sur les lignes. En appliquant la même succession d'opérations sur la matrice I_n on obtient la matrice A^{-1}.

b) Rédiger une fonction `inverse(A)` qui calcule l'inverse de la matrice A par la méthode du pivot. On travaillera sur une copie de A pour le pas modifier cette dernière.

2.5 Utilisation du module `numpy.linalg`

Le module `numpy.linalg` contient un certain nombre de fonctions dédiées à l'algèbre linéaire, en particulier une fonction `solve` qui joue le même rôle que la fonction `gauss` que nous venons d'écrire. Celle-ci va nous permettre de tester notre fonction pour résoudre le système :

$$\begin{pmatrix} 2 & 4 & -4 & 1 \\ 3 & 6 & 1 & -2 \\ -1 & 1 & 2 & 3 \\ 1 & 1 & -4 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ -7 \\ 4 \\ 2 \end{pmatrix}$$

```
>>> from numpy import linalg as la

>>> A = np.array([[2, 4, -4, 1],
                  [3, 6, 1, -2],
                  [-1, 1, 2, 3],
                  [1, 1, -4, 1]], dtype=float)

>>> b = np.array([0, -7, 4, 2], dtype=float)

>>> gauss(A, b)
array([ 1., -1., -0.,  2.])

>>> la.solve(A, b)
array([ 1., -1., -0.,  2.])
```

Le module `numpy.linalg` contient en outre une fonction `det` pour calculer le déterminant ainsi qu'une fonction `inv` pour calculer l'inverse :

```
>>> determinant(A)
-28.000000000000014

>>> la.det(A)
-28.000000000000021
```

```
>>> inverse(A)
array([[-3.5       ,  1.5       ,  0.75      ,  4.25      ],
       [ 1.5       , -0.5       , -0.25      , -1.75      ],
       [-0.78571429,  0.35714286,  0.25      ,  0.75      ],
       [-1.14285714,  0.42857143,  0.5       ,  1.5       ]])

>>> la.inv(A)
array([[-3.5       ,  1.5       ,  0.75      ,  4.25      ],
       [ 1.5       , -0.5       , -0.25      , -1.75      ],
       [-0.78571429,  0.35714286,  0.25      ,  0.75      ],
       [-1.14285714,  0.42857143,  0.5       ,  1.5       ]])
```

Annexe : la méthode du pivot partiel de Gauss

```
def recherche_pivot(A, b, j):
    p = j
    for i in range(j+1, A.shape[0]):
        if abs(A[i, j]) > abs(A[p, j]):
            p = i
    if p != j:
        b[[p, j]] = b[[j, p]]
        A[[p, j]] = A[[j, p]]

def elimination_bas(A, b, j):
    for i in range(j+1, A.shape[0]):
        b[i] = b[i] - (A[i, j] / A[j, j]) * b[j]
        A[i] = A[i] - (A[i, j] / A[j, j]) * A[j]

def descente(A, b):
    for j in range(A.shape[1] - 1):
        recherche_pivot(A, b, j)
        elimination_bas(A, b, j)

def elimination_haut(A, b, j):
    for i in range(j):
        b[i] = b[i] - (A[i, j] / A[j, j]) * b[j]

def remontee(A, b):
    for j in range(A.shape[1] - 1, 0, -1):
        elimination_haut(A, b, j)

def solve_diagonal(A, b):
    for k in range(b.shape[0]):
        b[k] = b[k] / A[k, k]
    return b

def gauss(A, b):
    U = A.copy()
    v = b.copy()
    descente(U, v)
    remontee(U, v)
    return solve_diagonal(U, v)
```

```
def determinant(A):
    U = A.copy()
    d = 1
    for j in range(U.shape[1] - 1):
        # recherche du pivot
        p = j
        for i in range(j+1, U.shape[0]):
            if U[i, j] > U[p, j]:
                p = i
        # permutation des deux lignes
        if p != j:
            d *= -1
            U[[p, j]] = U[[j, p]]
        # élimination
        for i in range(j+1, U.shape[0]):
                U[i] = U[i] - (U[i, j] / U[j, j]) * U[j]
    # produit des éléments diagonaux
    for k in range(U.shape[0]):
        d *= U[k, k]
    return d
```

```
def inverse(A):
    return gauss(A, np.identity(A.shape[0]))
```