Résolution numérique d'un système linéaire

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

PYTHON et le calcul matriciel

Le module NUMPY contient les éléments indispensables à la modélisation des vecteurs, matrices et tableaux multidimensionnels. Pour cela, NUMPY fournit le type ndarray, qui diffère du type list sur plusieurs points :

- la taille des tableaux NUMPY est fixée au moment de la création ;

PYTHON et le calcul matriciel

Le module NUMPY contient les éléments indispensables à la modélisation des vecteurs, matrices et tableaux multidimensionnels. Pour cela, NUMPY fournit le type ndarray, qui diffère du type list sur plusieurs points :

- la taille des tableaux NUMPY est fixée au moment de la création ;
- les tableaux NUMPY sont homogènes.

Python et le calcul matriciel

Le module numpy contient les éléments indispensables à la modélisation des vecteurs, matrices et tableaux multidimensionnels. Pour cela, numpy fournit le type ndarray, qui diffère du type list sur plusieurs points :

- la taille des tableaux numpy est fixée au moment de la création ;
- les tableaux numpy sont homogènes.

En contrepartie, l'accès aux éléments d'un tableau numpy est incomparablement plus rapide.

PYTHON et le calcul matriciel

Le module NUMPY contient les éléments indispensables à la modélisation des vecteurs, matrices et tableaux multidimensionnels. Pour cela, NUMPY fournit le type ndarray, qui diffère du type list sur plusieurs points :

- la taille des tableaux NUMPY est fixée au moment de la création ;
- les tableaux NUMPY sont homogènes.

En contrepartie, l'accès aux éléments d'un tableau NUMPY est incomparablement plus rapide.

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

```
import numpy as np
```

Création de tableaux

La fonction `array` crée un tableau à partir de la liste de ses éléments :

```
>>> 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]])
```

Création de tableaux

La fonction `array` crée un tableau à partir de la liste de ses éléments :

```
>>> 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 crée est à coefficients entiers car tous ses éléments sont des entiers ; si on change l'un des coefficients le nouveau coefficient est au préalable converti en entier :

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

Création de tableaux

Si la liste des éléments est hétérogène, certains seront automatiquement convertis :

```
>>> 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.]])
```

Création de tableaux

Si la liste des éléments est hétérogène, certains seront automatiquement convertis :

```
>>> 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.]])
```

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

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

Création de tableaux

types autorisés

Les principaux types sont : `bool` `int` `float` `complex`

Création de tableaux

types autorisés

Les principaux types sont : **`bool`** **`int`** **`float`** **`complex`**

D'autres types existent :

- entiers signés sur 8-16-32-64 bits : `int8`, `int16`, `int32`, `int64` ;

Création de tableaux

types autorisés

Les principaux types sont : **`bool`** **`int`** **`float`** **`complex`**

D'autres types existent :

- entiers signés sur 8-16-32-64 bits : `int8`, `int16`, `int32`, `int64` ;
- entiers non signés : `uint8`, `uint16`, `uint32`, `uint64` ;

Création de tableaux

types autorisés

Les principaux types sont : **bool** **int** **float** **complex**

D'autres types existent :

- entiers signés sur 8-16-32-64 bits : int8, int16, int32, int64 ;
- entiers non signés : uint8, uint16, uint32, uint64 ;
- flottants sur 8-16-32-64 bits : float8, float16, float32, float64.

Création de tableaux

types autorisés

Les principaux types sont : **`bool`** **`int`** **`float`** **`complex`**
D'autres types existent :

- entiers signés sur 8-16-32-64 bits : `int8`, `int16`, `int32`, `int64` ;
- entiers non signés : `uint8`, `uint16`, `uint32`, `uint64` ;
- flottants sur 8-16-32-64 bits : `float8`, `float16`, `float32`, `float64`.

En réalité les types **`int`** et **`float`** sont automatiquement remplacés par `int64` et `float64` (sur les machines à processeur 64 bits).

Création de tableaux

types autorisés

Les principaux types sont : **bool** **int** **float** **complex**
D'autres types existent :

- entiers signés sur 8-16-32-64 bits : int8, int16, int32, int64 ;
- entiers non signés : uint8, uint16, uint32, uint64 ;
- flottants sur 8-16-32-64 bits : float8, float16, float32, float64.

En réalité les types **int** et **float** sont automatiquement remplacés par int64 et float64 (sur les machines à processeur 64 bits).

Exemple. Soit une image non compressée 1600×1200, chaque pixel étant représenté par un triplet RGB : avec NUMPY cette image est modélisée par un tableau tri-dimensionnel $1600 \times 1200 \times 3$.

Création de tableaux

types autorisés

Les principaux types sont : **bool** **int** **float** **complex**
D'autres types existent :

- entiers signés sur 8-16-32-64 bits : int8, int16, int32, int64 ;
- entiers non signés : uint8, uint16, uint32, uint64 ;
- flottants sur 8-16-32-64 bits : float8, float16, float32, float64.

En réalité les types **int** et **float** sont automatiquement remplacés par int64 et float64 (sur les machines à processeur 64 bits).

Exemple. Soit une image non compressée 1600×1200, chaque pixel étant représenté par un triplet RGB : avec NUMPY cette 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).

- Avec le data type *uint8* l'espace mémoire utilisé pour représenter cette image vaut $1600 \times 1200 \times 3 = 5\,760\,000$ octets soit 5,5 Mio ;
- avec le data type *int64* l'espace mémoire est 8 fois plus important, soit 44 Mio.

Coupes

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

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

Coupes

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

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

Pour les tableaux bi-dimensionnels `a[i, j]` est équivalente à `a[i][j]` :

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

Coupes

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

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

Pour les tableaux bi-dimensionnels `a[i, j]` est équivalente à `a[i][j]` :

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

Le slicing suit la même syntaxe que pour les listes PYTHON :

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

>>> 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.])
```

Coupes

Attention : à la différence des listes PYTHON, une coupe ne crée pas un nouveau tableau (la terminologie officielle parle dans ce cas de «vue»).

Coupes

Attention : à la différence des listes PYTHON, une coupe ne crée pas un nouveau tableau (la terminologie officielle parle dans ce cas de «vue»).

Pour copier un tableau NUMPY il faut 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 ne se répercute pas
```

Coupes

Attention : à la différence des listes PYTHON, une coupe ne crée pas un nouveau tableau (la terminologie officielle parle dans ce cas de «vue»).

`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.]])
```

Coupes

Attention : à la différence des listes PYTHON, une coupe ne crée pas un nouveau tableau (la terminologie officielle parle dans ce cas de «vue»).

`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 conseillé de s'interdire l'affectation simultanée et lui préférer :

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

Coupes

fancy indexing

On peut accéder au contenu d'un tableau NUMPY tant en lecture qu'en écriture en utilisant le fancy indexing : 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]]        # ou a[[i, j], :] = a[[j, i], :]
```

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

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

Redimensionnement d'un tableau

La méthode shape donne les dimensions d'un tableau :

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

Redimensionnement d'un tableau

La méthode shape donne les dimensions d'un tableau :

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

Modifier cet attribut redimensionne le tableau, à condition que le nombre total d'éléments du tableau reste inchangé :

```
>>> 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])
```

Redimensionnement d'un tableau

Notons que NUMPY différencie les tableaux uni-dimensionnels (les vecteurs) des tableaux bi-dimensionnels :

```
>>> 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 distingue les éléments de $\mathbb{R}^n$ de ceux de $\mathcal{M}_{1,n}(\mathbb{R})$ et de $\mathcal{M}_{n,1}(\mathbb{R})$.

Redimensionnement d'un tableau

Redimensionner une matrice est une opération de coût constant : les coefficients sont stockés en mémoire dans des cases *contiguës* et le format de la matrice dans un emplacement spécifique.

```
>>> 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.]])
```

a :

shape=3,4	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
	1re ligne				2e ligne				3e ligne			

Redimensionnement d'un tableau

Redimensionner une matrice est une opération de coût constant : les coefficients sont stockés en mémoire dans des cases *contiguës* et le format de la matrice dans un emplacement spécifique.

```
>>> 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.]])
```

a :	shape=3,4	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
		1$^{\text{re}}$ ligne				2$^{\text{e}}$ ligne				3$^{\text{e}}$ ligne			

Connaître la case dans laquelle se trouve l'élément de rang (i, j) n'est que le résultat d'un calcul arithmétique : pour une matrice de n lignes et p colonnes il s'agit de la case de rang $i \times p + j$.

Création de tableaux spécifiques

La fonction `zeros` crée un tableau dont tous les coefficients sont nuls :

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

Création de tableaux spécifiques

La fonction zeros crée un tableau dont tous les coefficients sont nuls :

```
>>> 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.]])
```

Création de tableaux spécifiques

La fonction zeros crée un tableau dont tous les coefficients sont nuls :

```
>>> 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 :

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

Opérations élémentaires

- `a + b` calcule la somme de deux matrices ;

Opérations élémentaires

- `a + b` calcule la somme de deux matrices ;
- `t * a` calcule le produit du scalaire t par la matrice a ;

Opérations élémentaires

- `a + b` calcule la somme de deux matrices ;
- `t * a` calcule le produit du scalaire t par la matrice a ;
- attention : `a * b` ne calcule pas le produit de deux matrices !

Opérations élémentaires

- `a + b` calcule la somme de deux matrices ;
- `t * a` calcule le produit du scalaire t par la matrice a ;
- attention : `a * b` ne calcule pas le produit de deux matrices !

On calcule le produit de deux matrices ou d'une matrice par un vecteur avec la fonction `dot`.

$L_i \leftarrow L_i + tL_j$ s'écrit :

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

$L_i \leftarrow tL_i$ s'écrit :

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

$L_i \leftrightarrow L_j$ s'écrit :

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

Opérations élémentaires

- `a + b` calcule la somme de deux matrices ;
- `t * a` calcule le produit du scalaire t par la matrice a ;
- attention : `a * b` ne calcule pas le produit de deux matrices !

On calcule le produit de deux matrices ou d'une matrice par un vecteur avec la fonction `dot`.

Opérations élémentaires sur les lignes et les colonnes

$L_i \leftarrow L_i + tL_j$ s'écrit :

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

$L_i \leftarrow tL_i$ s'écrit :

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

$L_i \leftrightarrow L_j$ s'écrit :

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

Opérations élémentaires

- `a + b` calcule la somme de deux matrices ;
- `t * a` calcule le produit du scalaire t par la matrice a ;
- attention : `a * b` ne calcule pas le produit de deux matrices !

On calcule le produit de deux matrices ou d'une matrice par un vecteur avec la fonction `dot`.

Opérations élémentaires sur les lignes et les colonnes

$L_i \leftarrow L_i + tL_j$ s'écrit :

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

$L_i \leftarrow tL_i$ s'écrit :

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

$L_i \leftrightarrow L_j$ s'écrit :

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

$C_i \leftarrow C_i + tC_j$ s'écrit :

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

$C_i \leftarrow tC_i$ s'écrit :

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

$C_i \leftrightarrow C_j$ s'écrit :

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

Méthode du pivot de Gauss

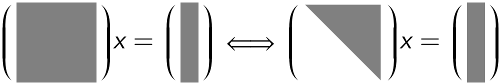
C'est une méthode de résolution d'un système linéaire : $Ax = b$, où A est une matrice inversible : on ne modifie pas l'ensemble des solutions d'une équation linéaire en appliquant les mêmes opérations élémentaires sur les lignes de A et de b.

Méthode du pivot de Gauss

C'est une méthode de résolution d'un système linéaire : $Ax = b$, où A est une matrice inversible : 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 de descente : on transforme la matrice A en une matrice A' *triangulaire supérieure* tout en effectuant les mêmes opérations sur b ;

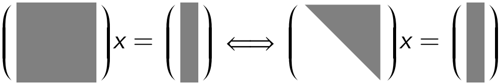


Méthode du pivot de Gauss

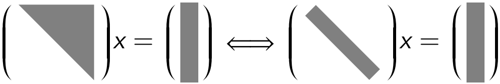
C'est une méthode de résolution d'un système linéaire : $Ax = b$, où A est une matrice inversible : on ne modifie pas l'ensemble des solutions d'une équation linéaire en appliquant les mêmes opérations élémentaires sur les lignes de A et de b.

La méthode du pivot de Gauss comporte trois étapes :

1. une première étape de descente : on transforme la matrice A en une matrice A' *triangulaire supérieure* tout en effectuant les mêmes opérations sur b ;



2. une deuxième étape de remontée : on transforme la matrice A' en une matrice A'' *diagonale* tout en effectuant les mêmes opérations sur b ;

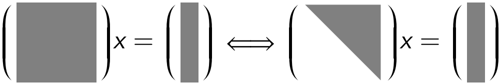


Méthode du pivot de Gauss

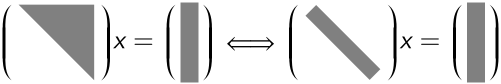
C'est une méthode de résolution d'un système linéaire : $Ax = b$, où A est une matrice inversible : on ne modifie pas l'ensemble des solutions d'une équation linéaire en appliquant les mêmes opérations élémentaires sur les lignes de A et de b.

La méthode du pivot de Gauss comporte trois étapes :

1. une première étape de descente : on transforme la matrice A en une matrice A' *triangulaire supérieure* tout en effectuant les mêmes opérations sur b ;



2. une deuxième étape de remontée : on transforme la matrice A' en une matrice A'' *diagonale* tout en effectuant les mêmes opérations sur b ;



3. une dernière étape de résolution du système linéaire diagonal obtenu.

L'étape de descente

On maintenant l'invariant :

à l'entrée de la j^{e} boucle, $$A = \begin{pmatrix} a_{11} & a_{12} & \multicolumn{3}{c}{\dotfill} & a_{1n} \\ 0 & a_{22} & \multicolumn{3}{c}{\dotfill} & a_{2n} \\ \vdots & \ddots & \ddots & & & \vdots \\ 0 & \cdots & 0 & {\color{red}a_{jj}} & \cdots & a_{jn} \\ \vdots & & \vdots & {\color{red}\vdots} & & \vdots \\ 0 & \cdots & 0 & {\color{red}a_{nj}} & \cdots & a_{nn} \end{pmatrix}.$$

L'étape de descente

On maintenant l'invariant :

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

1. on cherche un pivot *non nul* a_{ij}, $j \leqslant i \leqslant n$ puis on permute les lignes L_i et L_j : $L_i \leftrightarrow L_j$;
2. à l'issue de cette première étape on utilise a_{jj} comme pivot pour remplacer tous les termes a_{ij} pour $i > j$ par des zéros à l'aide de l'opération élémentaire :

$$\forall i \in [\![j+1, n]\!], \qquad L_i \leftarrow L_i - \frac{a_{ij}}{a_{jj}} L_j.$$

Le choix du pivot

Le choix d'un pivot trop petit conduit à des erreurs d'arrondi importantes.

Le choix du pivot

Le choix d'un pivot trop petit conduit à des erreurs d'arrondi importantes.

On suppose les calculs effectués en virgule flottante dans le système décimal avec une mantisse de quatre chiffres et on considère le système :

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

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

On choisit 0,003 pour pivot.

Le choix du pivot

Le choix d'un pivot trop petit conduit à des erreurs d'arrondi importantes.

On suppose les calculs effectués en virgule flottante dans le système décimal avec une mantisse de quatre chiffres et on considère 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 $x = 10$, $y = 1$.

On choisit 0,003 pour pivot.

On exécute $L_2 \leftarrow L_2 - \lambda L_1$ avec $\lambda = \dfrac{5{,}291}{0{,}003} = 1\,763{,}666\cdots \approx 1{,}764\cdot 10^{3}$:

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

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

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

Les termes de la seconde ligne (en rouge) ont été absorbés.

Le choix du pivot

Le choix d'un pivot trop petit conduit à des erreurs d'arrondi importantes.

On suppose les calculs effectués en virgule flottante dans le système décimal avec une mantisse de quatre chiffres et on considère le système :

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

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

On choisit 0,003 pour pivot.

On exécute $L_2 \leftarrow L_2 - \lambda L_1$ avec $\lambda = \dfrac{5{,}291}{0{,}003} = 1\,763{,}666\cdots \approx 1{,}764\cdot 10^{3}$:

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

On exécute $L_1 \leftarrow L_1 - \mu L_2$ avec $\mu = \dfrac{5{,}914\cdot 10^{1}}{-1{,}043\cdot 10^{5}} \approx -5{,}670\cdot 10^{-4}$:

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

Le choix du pivot

Le choix d'un pivot trop petit conduit à des erreurs d'arrondi importantes.

On suppose les calculs effectués en virgule flottante dans le système décimal avec une mantisse de quatre chiffres et on considère le système :

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

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

On choisit 0,003 pour pivot.

On exécute $L_2 \leftarrow L_2 - \lambda L_1$ avec $\lambda = \dfrac{5{,}291}{0{,}003} = 1\,763{,}666\cdots \approx 1{,}764\cdot 10^{3}$:

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

On exécute $L_1 \leftarrow L_1 - \mu L_2$ avec $\mu = \dfrac{5{,}914\cdot 10^{1}}{-1{,}043\cdot 10^{5}} \approx -5{,}670\cdot 10^{-4}$:

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

La solution numérique obtenue est : $x \approx -6{,}667$ et $y \approx 1{,}001$. L'erreur faite sur y est de l'ordre de $0,1\%$, l'erreur faite sur x est de l'ordre de 167%.

Le choix du pivot

Le choix d'un pivot trop petit conduit à des erreurs d'arrondi importantes.

On suppose les calculs effectués en virgule flottante dans le système décimal avec une mantisse de quatre chiffres et on considère le système :

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

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

On choisit 5,291 pour pivot.

Le choix du pivot

Le choix d'un pivot trop petit conduit à des erreurs d'arrondi importantes.

On suppose les calculs effectués en virgule flottante dans le système décimal avec une mantisse de quatre chiffres et on considère 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 $x = 10$, $y = 1$.

On choisit 5,291 pour pivot.

On permute L_1 et L_2 puis on exécute $L_2 \leftarrow L_2 - \lambda L_1$ avec $\lambda = \dfrac{0{,}003}{5{,}291} \approx 5{,}670\cdot 10^{-4}$:

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

Le choix du pivot

Le choix d'un pivot trop petit conduit à des erreurs d'arrondi importantes.

On suppose les calculs effectués en virgule flottante dans le système décimal avec une mantisse de quatre chiffres et on considère le système :

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

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

On choisit 5,291 pour pivot.

On permute L_1 et L_2 puis on exécute $L_2 \leftarrow L_2 - \lambda L_1$ avec $\lambda = \dfrac{0{,}003}{5{,}291} \approx 5{,}670\cdot 10^{-4}$:

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

On exécute $L_1 \leftarrow L_1 - \mu L_2$ avec $\mu = \dfrac{-6{,}130\cdot 10^{0}}{5{,}914\cdot 10^{1}} \approx -1{,}037\cdot 10^{-1}$:

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

Le choix du pivot

Le choix d'un pivot trop petit conduit à des erreurs d'arrondi importantes.

On suppose les calculs effectués en virgule flottante dans le système décimal avec une mantisse de quatre chiffres et on considère 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 $x = 10$, $y = 1$.

On choisit 5,291 pour pivot.

On permute L_1 et L_2 puis on exécute $L_2 \leftarrow L_2 - \lambda L_1$ avec $\lambda = \dfrac{0{,}003}{5{,}291} \approx 5{,}670\cdot 10^{-4}$:

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

On exécute $L_1 \leftarrow L_1 - \mu L_2$ avec $\mu = \dfrac{-6{,}130\cdot 10^{0}}{5{,}914\cdot 10^{1}} \approx -1{,}037\cdot 10^{-1}$:

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

La solution numérique obtenue est : $x \approx 1{,}000\cdot 10^{1}$ et $y \approx 1{,}000\cdot 10^{0}$ qui coïncident avec les résultats théoriques.

Le choix du pivot

Pivot partiel ou pivot total ?

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.

Le choix du pivot

Pivot partiel ou pivot total ?

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.

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

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

Le choix du pivot

Pivot partiel ou pivot total ?

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.

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

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

Cela induit une difficulté algorithmique supplémentaire non compensé par un gain de stabilité significatif.
Nous allons donc implémenter la méthode du *pivot partiel* de Gauss.

Programmation de l'étape de descente

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 permute les lignes L_i et L_j de A et de b.

Programmation de l'étape de descente

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 permute les lignes L_i et L_j de A et de b.

```
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:
        A[[p, j]] = A[[j, p]]
        b[[p, j]] = b[[j, p]]
```

Programmation de l'étape de descente

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

Rédiger une fonction `elimination_bas(A, b, j)` qui effectue les éliminations successives des coefficients situés sous a_{jj}, en effectuant en parallèle les mêmes opérations sur b.

Programmation de l'étape de descente

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 permute les lignes L_i et L_j de A et de b.

Rédiger une fonction `elimination_bas(A, b, j)` qui effectue les éliminations successives des coefficients situés sous a_{jj}, en effectuant en parallèle les mêmes opérations sur b.

```python
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]
```

Programmation de l'étape de descente

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

Rédiger une fonction `elimination_bas(A, b, j)` qui effectue les éliminations successives des coefficients situés sous a_{jj}, en effectuant en parallèle les mêmes opérations sur b.

Rédiger une fonction `descente(A, b)` qui par opérations élémentaires sur les lignes des matrices A et b réalise l'étape de descente.

Programmation de l'étape de descente

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 permute les lignes L_i et L_j de A et de b.

Rédiger une fonction `elimination_bas(A, b, j)` qui effectue les éliminations successives des coefficients situés sous a_{jj}, en effectuant en parallèle les mêmes opérations sur b.

Rédiger une fonction `descente(A, b)` qui par opérations élémentaires sur les lignes des matrices A et b réalise l'étape de descente.

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

L'étape de remontée

Une fois la matrice A triangulaire supérieure, on la transforme en matrice diagonale en maintenant l'invariant : à l'entrée de la $(n-j+1)^{\text{e}}$ boucle,

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1j} & 0 & \cdots & 0 \\ 0 & \ddots & \vdots & \vdots & & \vdots \\ \vdots & \ddots & a_{jj} & 0 & & \vdots \\ 0 & \cdots & 0 & a_{j+1,j+1} & \ddots & \vdots \\ \vdots & & \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \cdots & 0 & a_{nn} \end{pmatrix}$$

On utilise a_{jj} comme pivot pour éliminer les coefficients $a_{1j}, \ldots, a_{j-1,j}$.

L'étape de remontée

Programmation

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

L'étape de remontée

Programmation

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

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

On remarquera qu'il n'est pas nécessaire d'effectuer les opérations correspondantes sur la matrice A.

L'étape de remontée

Programmation

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

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

On remarquera qu'il n'est pas nécessaire d'effectuer les opérations correspondantes sur la matrice A.

Rédiger une fonction `remontee(A, b)` qui réalise l'étape de remontée de la méthode du pivot de Gauss.

L'étape de remontée

Programmation

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

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

On remarquera qu'il n'est pas nécessaire d'effectuer les opérations correspondantes sur la matrice A.

Rédiger une fonction `remontee(A, b)` qui réalise l'étape de remontée de la méthode du pivot de Gauss.

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

Méthode du pivot partiel de Gauss

Rédiger une fonction `solve_diagonal(A, b)` qui retourne l'unique vecteur x solution de l'équation $Ax = b$ lorsque A est diagonale.

Méthode du pivot partiel de Gauss

Rédiger une fonction `solve_diagonal(A, b)` qui retourne l'unique vecteur x solution de l'équation $Ax = b$ lorsque A est diagonale.

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

Méthode du pivot partiel de Gauss

Rédiger une fonction solve_diagonal(A, b) qui retourne l'unique vecteur x solution de l'équation $Ax = b$ lorsque A est diagonale.

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

Rédiger une fonction gauss(A, b) qui retourne l'unique solution d'un système de Cramer $Ax = b$. On travaillera sur des copies de A et b.

Méthode du pivot partiel de Gauss

Rédiger une fonction solve_diagonal(A, b) qui retourne l'unique vecteur x solution de l'équation $Ax = b$ lorsque A est diagonale.

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

Rédiger une fonction gauss(A, b) qui retourne l'unique solution d'un système de Cramer $Ax = b$. On travaillera sur des copies de A et b.

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

Étude de la complexité

Les différentes fonctions écrites ont pour coûts temporels respectifs :

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

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

Applications de la méthode du pivot

Calcul du déterminant

Rédiger une fonction `determinant(A)` qui calcule le déterminant de la matrice A. On travaillera sur une copie de A.

Applications de la méthode du pivot

Calcul du déterminant

Rédiger une fonction `determinant(A)` qui calcule le déterminant de la matrice A. On travaillera sur une copie de A.

```python
def determinant(A):
    U = A.copy()
    d = 1
    for j in range(U.shape[1] - 1):
        p = j
        for i in range(j+1, U.shape[0]):
            if U[i, j] > U[p, j]:
                p = i
        if p != j:
            d *= -1
            U[[p, j]] = U[[j, p]]
        for i in range(j+1, U.shape[0]):
            U[i] = U[i] - (U[i, j] / U[j, j]) * U[j]
    for k in range(U.shape[0]):
        d *= U[k, k]
    return d
```

Applications de la méthode du pivot

Calcul de l'inverse

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.

Applications de la méthode du pivot

Calcul de l'inverse

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.

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

Utilisation du module numpy.linalg

Le module numpy.linalg contient une fonction solve :

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

Utilisation du module numpy.linalg

Le module numpy.linalg contient une fonction solve :

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

Ainsi qu'une fonction det et une fonction inv :

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