# L1 - Calcul Scientifique

Youssef Chahir
youssef.chahir@unicaen.fr

Université de Caen Normandie



## Logique

- AND & : a & b
- OR | : a | b
- NOT ~ : ~a
- NAND : ~(a & b)
- XOR : a ^ b
- => : a >> b



## Algèbre de Boole

- Opérations logiques: ET & , OU | et NON, ~
- Représentation d'une fonction Booléenne :
  - $f = (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$

```
In [1]: from sympy import *
        init_printing()
        x,y,z=symbols('x,y,z')
        f=(x&y)|(y&z)|(z&x)
        f
```

Out[1]: $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$

- Une fois f définie, on peut chercher quelles sont les variables de f par la méthode **free_symbols** :

```
In [7]: f.free_symbols
```

Out[7]: $\{x, y, z\}$

```
In [2]: f.subs({x:True,y:True,z:False})
```

Out[2]: True



## Algèbre de Boole

- La fonction peut aussi être définie comme une Lambda-expression Sympy :

```
In [9]: g=Lambda([x,y,z],(x&y)|(y&z)|(z&x))
In [10]: g(True,True,False)
```

Out[10]: True

- Avec cette définition, les variables de g ne sont plus libres (elles sont liées par Lambda )

```
In [11]: g.free_symbols
```

Out[11]: {}

- Construire une table de vérité:

```
In [18]: for i in cartes([False,True],repeat=3):
             a,b,c=i
             print(i,'\t',g(a,b,c))

(False, False, False)     False
(False, False, True)      False
(False, True, False)      False
(False, True, True)       True
(True, False, False)      False
(True, False, True)       True
(True, True, False)       True
(True, True, True)        True
```

la fonction cartes de sympy nous permet de calculer directement le produit Cartésien qui nous engendre l'ensemble des valuations possibles des variables de la fonction

## Texte du titre

```
f=(p&q)|(q&r)|(r&p)
print("table de vérité ")
for (a,b,c) in [(i,j,k) for i in (True, False) for j in (True, False) for k in (True, False)]:
            print(a,b,c,f.subs({p:a,q:b,r:c}))
```

```
ble de vérité
ue True True True
ue True False True
ue False True True
ue False False False
lse True True True
lse True False False
lse False True False
lse False False False
```



## Fonction conjonctive / disjonctive

```
f=(p&q)|(q&r)|(r&p)
print("Forme conjonctive :", sp.to_cnf(f))
```

```
Forme conjonctive : (p | q) & (p | r) & (q | r) & (p | q | r)
```

```
print("Forme disjonctive :", sp.to_dnf(f))
```

```
Forme disjonctive : (p & q) | (p & r) | (q & r)
```



## Fonction simplifiée

```
f= ((p >>s) | (r & (~ p )))& (~(s &r))
print("Forme simplifiée :",sp.simplify(f))
```

```
orme simplifiée : (s & ~r) | (~p & ~s)
```



## Algèbre de Boole

- Vérifier les identités remarquables :
  - **Equivalent(A,B)** renvoie vraie si et seulement si A et B sont soit tous deux vrais soit tous deux faux.

# Un problème inverse

- Comment faire si on connait la table de vérité et qu'on veut en déduire une fonction Booléenne ?
- On fait appel à la fonction **SOPform** qui cherche une fonction Booléenne qui a la même valeur de vérité :
  - On définit d'abord les valuations qui rendent vraies la table de vérité (on les appelle des **mintermes**)
  - Ensuite, les valuations pour lesquelles les valeurs de vérité ne sont pas précisées (elles sont vraies ou fausses), on les appelle des **dontcares**
  - Toutes les autres valeur sont supposées à faux.
  - **Rq:** On n'est pas obligé de définir les don't care.



# Extraire une formule d'une table

- Il faut “déclarer” les mintermes

```
minterms = [[0, 1, 0], [0, 0, 1],[1,1,1]]

G = sp.SOPform(['p', 'q', 'r'], minterms)
display(G)
H = sp.POSform(['p', 'q', 'r'], minterms)
display(H)
```

$(p \wedge q \wedge r) \vee (q \wedge \neg p \wedge \neg r) \vee (r \wedge \neg p \wedge \neg q)$

$(q \vee r) \wedge (q \vee \neg p) \wedge (r \vee \neg p) \wedge (p \vee \neg q \vee \neg r)$



# Un problème inverse

```
In [21]: x,y,z=symbols('x,y,z')

In [22]: minterms=[[0,1,0]]

In [24]: dontcares=[[0,0,0],[0,0,1]]

In [28]: SOPform([x,y,z],minterms)
```

Out[28]: $y \wedge \neg x \wedge \neg z$

| $x$ | $y$ | $z$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | * |
| 0 | 0 | 1 | * |
| 0 | 1 | 0 | 1 |



Algèbre linéaire
Résolution d'équations

# Plan

1. Vecteurs / Matrices
2. Arithmétique matricielle
2. Propriétés des matrices
3. Vecteurs propres et valeurs propres
4. Exemples
5. Propriétés supplémentaires



# Matrice/Vecteur

- Un vecteur est défini comme une matrice à une seule colonne :
  - Sympy :
    - u=sp.Matrix([2,-3,1])
    - np shape(v) : (3,1)
  - Numpy
    - V1 = np.array([2, -3, 1])
    - np shape(v) : (3,)
  - Sympy/Numpy : print(v[0],v[1],v[2]) : 2 -3 1
  - np.shape(v) : Dimensions
  - len(v) : Nombre d'éléments

$$\begin{bmatrix} 2 \\ -3 \\ 1 \end{bmatrix}$$

- Une matrice :
  - Sympy :
    - A = sp.Matrix([[2, -3, 1],[1, -1, 2],[3, 1, -1]])
    - A=sp.Matrix(3,3, [2, -3, 1,1, -1, 2,3, 1, -1])
    - print(A[0,0],A[0,1],A[0,2])
  - Numpy :
    - A = np.array([[2, -3, 1],[1, -1, 2],[3, 1, -1]])
    - print(A[0][0],A[0][1],A[0][2])



# Algèbre Linéaire

- Matrice 0 : Ex:
  - zeros(4,6) ==> 0
  - ones(3,6) ==> 1
  - eye(5)
- Opérations de base : +,-,*,**
- Transposée de M : M.**transpose()**
- Accéder aux coefficients :
  - M[i,j]
  - M.row(i) : ligne i
  - M.col(j): colonne j



# Algèbre Linéaire

- Définir une matrice sympy:

```
Entrée [70]: Matrix([[2, 9, 3], [4, 5, 10], [2, 0, 3]])
```

Out[70]:

$$\begin{bmatrix} 2 & 9 & 3 \\ 4 & 5 & 10 \\ 2 & 0 & 3 \end{bmatrix}$$

```
Entrée [73]: N = Matrix(3,3,[2,9,3,4,5,10,-6,-1,-17]); N
```

Out[73]:

$$\begin{bmatrix} 2 & 9 & 3 \\ 4 & 5 & 10 \\ -6 & -1 & -17 \end{bmatrix}$$

```
Entrée [75]: v = Matrix([5,2,1]); v
```

Out[75]:

$$\begin{bmatrix} 5 \\ 2 \\ 1 \end{bmatrix}$$

## Matrice/Vecteur

- Norme d'un vecteur: np.linalg.norm(V1) ==>une valeur (pareil que np.sqrt(V1.dot(V1)))
- Transposée d'une matrice: np.transpose(A) ou A.T
- Matrices identité :np.eye(5)
- Déterminant d'une matrice : np.linalg.det(A) ==>7.999999999999998
  - Essayez de l'appliquer à une matrice qui n'est pas carrée ? Vous allez recevoir une `np.linalg.LinAlgError` !
- Matrices diagonale : np.diag(A)
- Trace d'une matrice : np.trace(A)=np.sum(np.diag(A))



## Matrice/Vecteur

- Inversion d'une matrice: np.linalg.inv(A)
  - Vérifions que le produit d'une matrice et son inverse fait bien l'identité :
    - np.dot(Ainv, A) == np.eye(n)
    - On préfère vérifier si on a des valeurs approchées: ss np.isclose( Ainv @ A, np.eye(n))
- Valeurs/vecteurs propres d'une matrice:
  - ls, vs = np.linalg.eig(M) # eigen_values, eigen_vectors
  - Premier vecteur propre
    - v0 = vs[:, 0] # toutes les lignes et la colonne 0
  - Première valeur propre
    - l0 = ls[0]
- x = np.linalg.solve(A, b) Si la matrice n'est pas inversible ou si elle n'est pas carrée, une erreur `np.linalg.LinAlgError` sera levée



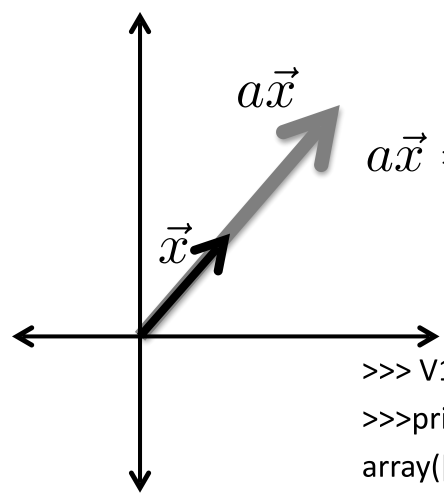
## Addition

- M1+M2

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}$$



## Vecteur scalaire




$$a\vec{x} = a \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} ax_1 \\ ax_2 \\ \vdots \\ ax_N \end{pmatrix}$$

```
>>> V1 = np.array([2, -3, 1])
>>>print(2*V1 )
array([ 4, -6,  2])
```

# Produit de 2 Vecteurs

Trois façons de se multiplier

- Élément par élément
- Produit intérieur
- Produit extérieur



# Produit élément par élément (produit Hadamard)

- Multiplication par élément (.)

$$\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_1 b_1 \\ a_2 b_2 \end{pmatrix}$$

```
>>> V1 = np.array([2, -3, 1])
>>>print(V1*V1 )
array([ 4, 9,  1])
```



## Multiplication:
## Produit scalaire (produit interne)

$$\vec{x} \cdot \vec{y} = (x_1 \quad x_2 \quad \cdots \quad x_N) \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} = x_1 y_1 + x_2 y_2 + \cdots + x_N y_N$$

$$= \sum_{i=1}^{N} x_i y_i$$

```
>>> V1 = np.array([2, -3, 1])
>>> V2 = np.array([[2], [-3], [1]])
>>>np.dot(V1, V2) # np.matmul(V1, V2)
>>> #Raccourci @ :  V1 @ V2
array([14])
```



## Multiplication:
## Produit scalaire (produit interne)

- 'les dimensions de la matrice interne doivent correspondre'

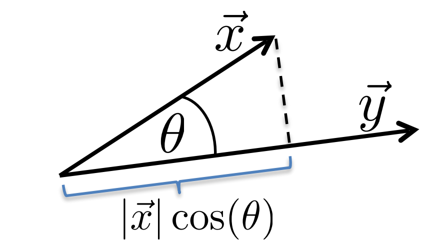
$$\vec{x} \cdot \vec{y} = (x_1 \quad x_2 \quad \cdots \quad x_N) \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} = x_1 y_1 + x_2 y_2 + \cdots + x_N y_N$$

```
>>> V1 = np.array([2, -3, 1])
>>>print(np.dot(V1, V2) )
14
```

1 X N   N X 1   1 X 1

Les dimensions extérieures donnent la taille de la matrice résultante

## Intuition géométrique du produit scalaire : "Chevauchement" de 2 vecteurs




$$\vec{x}\cdot\vec{y} = |\vec{x}||\vec{y}|\cos(\theta)$$



## Multiplication: Produit externe

```
>>> V1 = np.array([2, -3, 1])
>>> V2 = np.array([[2], [-3], [1]])
>>>V1*V2
array([[ 4, -6,  2],       [-6,  9, -3],       [ 2, -3,  1]])
```

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} \begin{pmatrix} y_1 & y_2 & \cdots & y_M \end{pmatrix} = \begin{pmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_M \\ x_2y_1 & x_2y_2 & \cdots & x_2y_M \\ \vdots & \vdots & \ddots & \vdots \\ x_Ny_1 & x_Ny_2 & \cdots & x_Ny_M \end{pmatrix}$$

- Remarque : chaque colonne ou chaque ligne est un multiple des autres.



## Matrice fois un vecteur: interprétation du produit interne

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_i \\ \vdots \\ y_M \end{pmatrix} = \begin{pmatrix} W_{11} & W_{12} & \cdots & W_{1N} \\ W_{21} & W_{22} & \cdots & W_{2N} \\ \vdots & \vdots & & \vdots \\ W_{i1} & W_{i2} & \cdots & W_{iN} \\ \vdots & \vdots & \ddots & \vdots \\ W_{M1} & W_{M2} & \cdots & W_{MN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}$$

- Règle: le $i^{\text{ème}}$ élément de **y** est le produit scalaire de la $i^{\text{ème}}$ ligne de W avec **x**



## Matrice fois un vecteur: interprétation du produit interne

$$\begin{array}{c} \vec{W}^{(1)} \\ \downarrow \end{array}$$

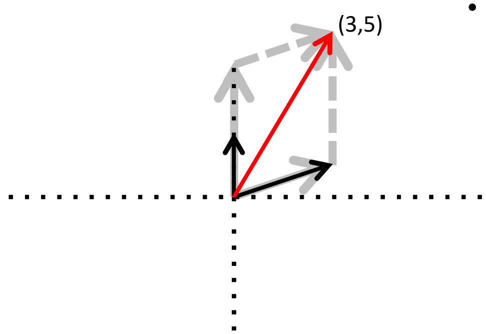
$$\begin{pmatrix} W_{11} & W_{12} & \cdots & W_{1N} \\ W_{21} & W_{22} & \cdots & W_{2N} \\ \vdots & \vdots & & \vdots \\ W_{M1} & W_{M2} & \cdots & W_{MN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = x_1\vec{W}^{(1)} + x_2\vec{W}^{(2)} + \cdots + x_N\vec{W}^{(N)}$$

- Le produit est une somme pondérée des colonnes de W, pondérée par les entrées de x

# Exemple de la méthode du produit externe

$$\overleftrightarrow{M}$$

$$\begin{pmatrix} 0 & 3 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} = 2 \begin{pmatrix} 0 \\ 2 \end{pmatrix} + 1 \begin{pmatrix} 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$




- Note: différentes combinaisons des colonnes de M peuvent donner n'importe quel vecteur dans le plan.

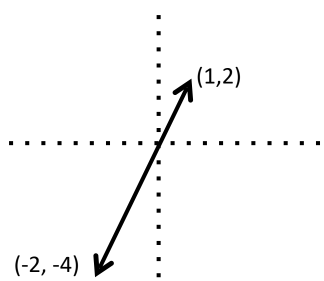
(on dit que les colonnes de **M** 'couvrent' le plan)



# Rang d'une matrice

- Existe-t-il des matrices spéciales dont les colonnes ne couvrent pas tout le plan ?

$$\begin{pmatrix} 1 & -2 \\ 2 & -4 \end{pmatrix}$$




- On ne peut obtenir que des vecteurs dans la direction (1,2) (c.à.d que les sorties sont à une dimension, donc on appelle la matrice de rang 1).



# Produit de 2 Matrices

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1P} \\ A_{21} & A_{22} & \cdots & A_{2P} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NP} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1M} \\ B_{21} & B_{22} & \cdots & B_{2M} \\ \vdots & \vdots & & \vdots \\ B_{P1} & B_{P2} & \cdots & B_{PM} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1M} \\ C_{21} & C_{22} & \cdots & C_{2M} \\ \vdots & \vdots & & \vdots \\ C_{N1} & C_{N2} & \cdots & C_{NM} \end{pmatrix}$$

N X P   P X M   N X M

- **Remarque** : la multiplication matricielle n'est pas commutative (généralement), **AB ≠ BA**



## Matrice fois Matrice : par produits internes

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1P} \\ A_{21} & A_{22} & \cdots & A_{2P} \\ \vdots & \vdots & & \vdots \\ A_{i1} & A_{i2} & \cdots & A_{iP} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NP} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1j} & \cdots & B_{1M} \\ B_{21} & B_{22} & \cdots & B_{2j} & \cdots & B_{2M} \\ \vdots & \vdots & & \vdots & & \vdots \\ B_{P1} & B_{P2} & \cdots & B_{Pj} & \cdots & B_{PM} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1M} \\ C_{21} & C_{22} & \cdots & C_{2M} \\ \vdots & \vdots & C_{ij} & \vdots \\ C_{N1} & C_{N2} & \cdots & C_{NM} \end{pmatrix}$$

$$C_{ij} = \sum_{k=1}^{P} A_{ik} B_{kj}$$

- $C_{ij}$ est le produit interne de la $i^{\text{ème}}$ ligne de **A** avec la $j^{\text{ème}}$ colonne de **B**

## Matrice fois Matrice : par produits externes

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1P} \\ A_{21} & A_{22} & \cdots & A_{2P} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NP} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1M} \\ B_{21} & B_{22} & \cdots & B_{2M} \\ \vdots & \vdots & & \vdots \\ B_{P1} & B_{P2} & \cdots & B_{PM} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1M} \\ C_{21} & C_{22} & \cdots & C_{2M} \\ \vdots & \vdots & & \vdots \\ C_{N1} & C_{N2} & \cdots & C_{NM} \end{pmatrix}$$

$$\overleftrightarrow{C} = \begin{pmatrix} \\ A^{c1} \\ \\ \end{pmatrix} \begin{pmatrix} & B^{r1} & \end{pmatrix} + \begin{pmatrix} \\ A^{c2} \\ \\ \end{pmatrix} \begin{pmatrix} & B^{r2} & \end{pmatrix} + \cdots + \begin{pmatrix} \\ A^{cP} \\ \\ \end{pmatrix} \begin{pmatrix} & B^{rP} & \end{pmatrix}$$

- **C** est une somme de produits externes des colonnes de **A** avec les lignes de **B**



## Partie 2 : Propriétés des matrices

- (Quelques) matrices spéciales
- Transformations matricielles et déterminant
- Matrices et systèmes d'équations algébriques

## Matrice diagonale

$$\overleftrightarrow{D} = \begin{pmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{pmatrix}$$

$$\overleftrightarrow{D}\,\vec{x} = \begin{pmatrix} d_1 x_1 \\ d_2 x_2 \\ \vdots \\ d_n x_n \end{pmatrix}$$

- Cela agit comme une multiplication scalaire



## Matrice identité

$$\overleftrightarrow{1} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

$$\text{for all } \overleftrightarrow{A}, \quad \overleftrightarrow{1}\,\overleftrightarrow{A} = \overleftrightarrow{A}\,\overleftrightarrow{1} = \overleftrightarrow{A}$$

## Matrice inverse

$$\overleftrightarrow{A}\,\overleftrightarrow{A}^{-1} = \overleftrightarrow{A}^{-1}\,\overleftrightarrow{A} = \overleftrightarrow{1}$$

- L'inverse existe-t-il toujours ?



## Résolution d'équations linéaires

- **NumPy** (**Num**erical **Py**thon)
- **SciPy** (**Sci**entific **Py**thon)
- **SymPy** (**Sym**bolic **Py**thon)

## Résolution d'équations linéaires

- Résolution avec 3 modules NumPy, SciPy et SymPy.
- Résoudre un système linéaire :
  - Une solution unique
  - Sans solution
  - Une infinité de solutions
- Exemple :

$$\begin{cases} 2x - 3y + z = -1 \\ x - y + 2z = -3 \\ 3x + y - z = 9 \end{cases}$$

## Représentation matricielle d'un système linéaire

$$\begin{cases} 2x - 3y + z = -1 \\ x - y + 2z = -3 \\ 3x + y - z = 9 \end{cases}$$

- Le système d'équations linéaires ci-dessus peut être représenté sous la forme d'une matrice spéciale appelée matrice augmentée :

$$\left[\begin{array}{ccc|c} 2 & -3 & 1 & -1 \\ 1 & -1 & 2 & -3 \\ 3 & 1 & -1 & 9 \end{array}\right]$$

# Représentation matricielle d'un système linéaire

$$\begin{cases} 2x - 3y + z = -1 \\ x - y + 2z = -3 \\ 3x + y - z = 9 \end{cases} \qquad \left[\begin{array}{ccc|c} 2 & -3 & 1 & -1 \\ 1 & -1 & 2 & -3 \\ 3 & 1 & -1 & 9 \end{array}\right]$$

- Il y a deux parties dans cette matrice augmentée :
  - La **matrice des coefficients** : Il s'agit d'un tableau rectangulaire qui contient uniquement les coefficients des variables. Dans notre exemple, il s'agit d'une matrice carrée de 3 x 3.

```
>>> import numpy as np
>>> A = np.array([[2, -3, 1],
              [1, -1, 2],
              [3, 1, -1]])
```

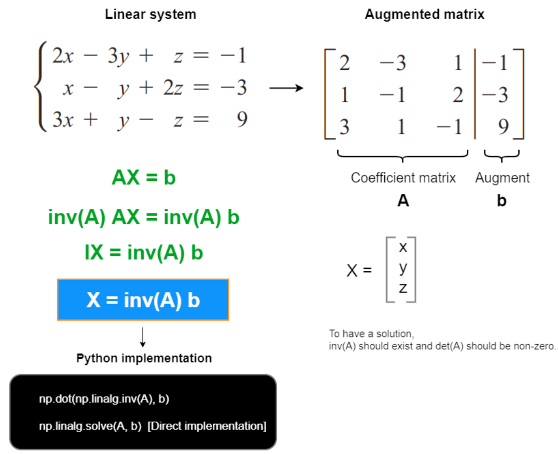
  - **Termes constants** : C'est un vecteur colonne à droite de la ligne verticale dans l'image ci-dessus. Il contient les constantes des équations linéaires. Dans notre exemple, il s'agit d'un vecteur

```
>>> b = np.array([-1, -3, 9])
```

# Comment cela fonctionne-t-il en interne ?

- Pour avoir une solution, l'inverse de A doit exister et le déterminant de A doit être non nul :

```
>>> np.linalg.det(A)
-19.000000000000004
```




```
>>> np.dot(np.linalg.inv(A), b)
array([ 2.,  1., -2.])
```

# Résolution de systèmes linéaires avec une solution unique

- Pour résoudre cette équation, on peut utiliser :
  - la fonction solve() du sous-paquetage **NumPy linalg**.
  - la fonction solve() du sous-paquetage **SciPy linalg**.

```
>>> A = np.array([[2, -3, 1],
              [1, -1, 2],
              [3, 1, -1]])
>>> b = np.array([-1, -3, 9])
>>> np.linalg.solve(A, b)
array([ 2.,  1., -2.])
```

```
>>> import scipy as sc
>>> A = np.array([[2, -3, 1],
              [1, -1, 2],
              [3, 1, -1]])
>>> b = np.array([-1, -3, 9])
>>> sc.linalg.solve(A, b)
array([ 2.,  1., -2.])
```

  - Le système linéaire ci-dessus a une solution unique :

x = 2   y = 1   z = -2

# Résolution de systèmes linéaires sans solution

$$\begin{cases} x - y + 4z = -5 \\ 3x + z = 0 \\ -x + y - 4z = 20 \end{cases}$$

```
>>> import numpy as np
>>> A = np.array([[1, -1, 4],
              [3, 0, 1],
              [-1, 1, -4]])
>>> b = np.array([-5, 0, 20])
>>> np.linalg.solve(A, b)
LinAlgError: Matrix is singular.
```

- Le message d'erreur indique que la matrice de coefficients (A) est singulière. Il s'agit donc d'une matrice non inversible dont le déterminant est nul. Vérifions-le avec :
  - la fonction det() du sous-paquetage **NumPy linalg**.

```
>>> np.linalg.det(A)
0.0
```

# Résolution de systèmes linéaires avec une infinité de solutions

$$\begin{cases} -x + \ y + 2z = \ \ 0 \\ \ x + 2y + \ z = \ \ 6 \\ -2x - \ y + \ z = -6 \end{cases}$$

```
>>> import numpy as np
>>> A = np.array([[-1, 1, 2],
              [1, 2, 1],
              [-2, -1, 1]])
>>> b = np.array([0, 6, -6])
>>> np.linalg.solve(A, b)
LinAlgError: Singular matrix
```

- Comment pouvons-nous distinguer les systèmes linéaires sans solution des systèmes linéaires avec une infinité de solutions ?

# Commen distinguer les systèmes linéaires avec ou sans solutions ?

- Forme réduite :(en utilisant SymPy)
  - Tout d'abord, créer la matrice augmentée,
  - Ensuite, utiliser la méthode rref().
- Exemple: avec une solution unique :

$$\begin{cases} 2x - 3y + \ z = -1 \\ \ x - \ y + 2z = -3 \\ 3x + \ y - \ z = \ \ 9 \end{cases}$$

```
>>> import sympy as sp
>>> A_augmente = sp.Matrix([[2, -3, 1, -1],
                      [1, -1, 2, -3],
                      [3, 1, -1, 9]])
>>> A_augmente.rref()[0]
```

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & -2 \end{bmatrix}$$

- Nous avons réussi ! Nous avons obtenu la forme réduite.
  - La 4e colonne est celle de la solution.
  - La solution est x=2, y=1 et z=-2, ce qui correspond à la solution précédente obtenue à l'aide de np.linalg.solve().

# Commen distinguer les systèmes linéaires avec ou sans solutions ?

- Mettre la matrice des coefficients sous la forme réduite qui a des 1 sur sa diagonale et des 0 partout ailleurs (matrice d'identité).
- Si on réussit alors le système a une solution unique. S
- Sinon, soit il n'y a pas de solution, soit il y a une infinité de solutions.
  - Dans ce cas, nous pouvons distinguer les systèmes linéaires sans solution des systèmes linéaires avec une infinité de solutions en regardant la dernière ligne de la matrice réduite.

# Commen distinguer les systèmes linéaires avec ou sans solutions ?

- Exemple: sans solution

$$\begin{cases} x - y + 4z = -5 \\ 3x \qquad + \ z = \ \ 0 \\ -x + y - 4z = 20 \end{cases}$$

```
>>> import sympy as sp
>>> A_augmente =  sp.Matrix([[1, -1, 4, -5],
                      [3, 0, 1, 0],
                      [-1, 1, -4, 20]])
>>> A_augmente.rref()[0]
```

$$\begin{bmatrix} 1 & 0 & \frac{1}{3} & 0 \\ 0 & 1 & -\frac{11}{3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Cette fois, nous n'avons pas réussi.
  - Nous n'avons pas obtenu la forme réduite ligne-échelon.
  - La 3ème ligne (équation) de cette forme est 0=1 ce qui est impossible ! Par conséquent, le système linéaire n'a pas de solution. Le système linéaire est incohérent.

## Commen distinguer les systèmes linéaires avec ou sans solutions ?

- Exemple: avec une une infinité de solutions :

```
>>> import sympy as sp
>>> A_augmente =  sp.Matrix([[-1, 1, 2, 0],
                             [1, 2, 1, 6],
                             [-2, -1, 1, -6]])

>>> A_augmente.rref()[0]
```

$$\begin{bmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{cases} -x + y + 2z = 0 \\ x + 2y + z = 6 \\ -2x - y + z = -6 \end{cases}$$

- Cette fois, nous n'avons pas réussi.
  - Nous n'avons pas obtenu la forme réduite ligne-échelon.
  - La 3ème ligne de cette forme est 0=0, ce qui est toujours vrai ! Cela implique que la variable z peut prendre n'importe quel nombre réel et que x et y peuvent être x-z = 2 (x = 2+z) et y+z = 2 (y = 2-z)

## Numpy : Matrice/Vecteur

- Transposée d'une matrice: np.transpose(A) ou A.T
- Matrices identité :np.eye(5)
- Déterminant d'une matrice : np.linalg.det(A) ==>7.999999999999998
  - Essayez de l'appliquer à une matrice qui n'est pas carrée ? Vous allez recevoir une `np.linalg.LinAlgError` !
- Matrices diagonale : np.diag(A)
- Trace d'une matrice : np.trace(A)=np.sum(np.diag(A))

## Matrices

```
>>> import numpy as np
>>> import scipy as sc
>>> import sympy as sp
```

```
>>> A =  sp.Matrix([[-1, 1, 2, 0],
                    [1, 2, 1, 6],
                    [-2, -1, 1, -6]])
>>> print(A,type(A))
Matrix([[-1, 1, 2, 0], [1, 2, 1, 6], [-2, -1, 1, -6]]) <class
'sympy.matrices.dense.MutableDenseMatrix'>

>>> B =  np.matrix([[-1, 1, 2, 0],
                    [1, 2, 1, 6],
                    [-2, -1, 1, -6]])
>>> print(B,type(B))

[[-1  1  2  0]
 [ 1  2  1  6]
 [-2 -1  1 -6]] <class 'numpy.matrix'>

>>> C = np.mat('[-1 1 2 0;1 2 1 6;-2 -1 1 -6 ]')
>>> print(C,

[[-1  1  2  0]
 [ 1  2  1  6]
 [-2 -1  1 -6]] <class 'numpy.matrix'>

>>> D =  np.array([[-1, 1, 2, 0], [1, 2, 1, 6], [-2, -1, 1, -6]])
>>> D = np.matrix(D)
```

## Numpy : Matrice/Vecteur

- Inversion d'une matrice: np.linalg.inv(A)
  - Vérifions que le produit d'une matrice et son inverse fait bien l'identité :
    - np.dot(Ainv, A) == np.eye(n)
    - On préfère vérifier si on a des valeurs approchées: ss np.isclose( Ainv @ A, np.eye(n))
- Valeurs/vecteurs propres d'une matrice:
  - ls, vs = np.linalg.eig(M) # eigen_values, eigen_vectors
  - Premier vecteur propre
    - v0 = vs[:, 0] # toutes les lignes et la colonne 0
  - Première valeur propre
    - l0 = ls[0]

## Résolution d'un système linéaire : Trouver les racines du système linéaire A . x = b

- x = np.linalg.solve(A, b)
- Si la matrice n'est pas inversible ou si elle n'est pas carrée, une erreur `np.linalg.LinAlgError` sera levée

Chapitre IV : Recherche des racines de fonctions non linéaires

Recherche de racines de systèmes d'équations

## Evaluation d'un système d'équations

$$\begin{cases} 2x + 3y + 1 = ? \\ -4x + 12z = ? \\ 23x + 4y - 42z - 4 = ? \\ 17z - 2 = ? \end{cases} \quad \text{pour } x = -11, y = 13 \text{ et } z = 16$$

$$A = \begin{bmatrix} 2 & 3 & 0 \\ -4 & 0 & 12 \\ 23 & 4 & 42 \\ 0 & 0 & 17 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 0 \\ -4 \\ -2 \end{bmatrix}, x = \begin{bmatrix} -11 \\ 13 \\ 16 \end{bmatrix}, Ax + b = ?$$

Solution numpy : A.dot(x)=b



## Résolution d'un système d'équations

$$\begin{cases} 2x + 3y + 1 = 1 \\ -4x + 12z = 2 \\ 23x + 4y - 42z - 4 = 3 \end{cases}$$

$$A = \begin{bmatrix} 2 & 3 & 0 \\ -4 & 0 & 12 \\ 23 & 4 & 42 \end{bmatrix}, y = \begin{bmatrix} -1 + 1 \\ 0 + 2 \\ 4 + 3 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 7 \end{bmatrix}, Ax = y, x = ?$$

Solution numpy : np.linalg.solve(A,y)

# Résolution d'un système d'équations (Ax=y)

$$2x_1 + 3x_2 = 4$$
$$5x_1 + 4x_2 = 3$$

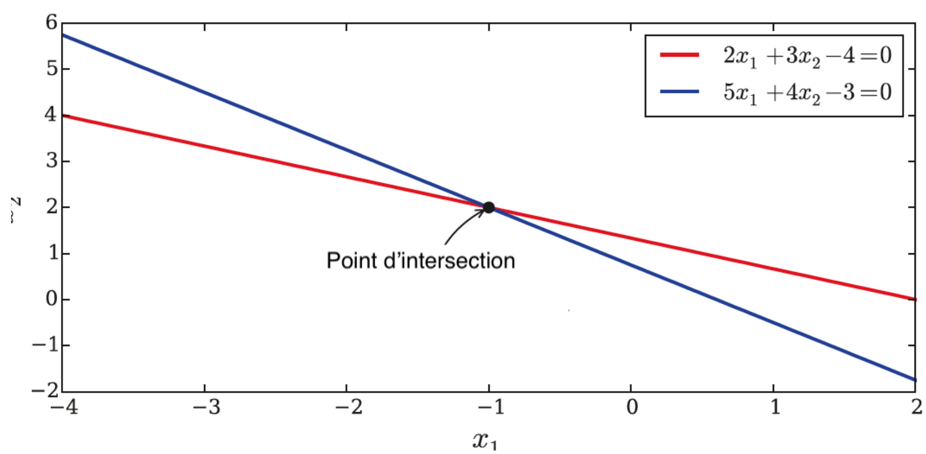
Solution numpy :  x,_,_,_ = np.linalg.lstsq(A,y)

np.linalg.lstsq : Renvoie la solution des moindres carrés d'un
équation matricielle linéaire.
Elle recherche la solution la plus proche possible.
Elle cherche à minimiser l'écart entre Ax et y .
Elle retourne le résultat de arg minx ‖Ax − y‖



# Exemple

$$2x_1 + 3x_2 = 4$$
$$5x_1 + 4x_2 = 3$$




# Solution avec sympy

```
In [8]: A = sympy.Matrix([[2, 3], [5, 4]])
In [9]: b = sympy.Matrix([4, 3])
In [10]: A.rank()
Out[10]: 2
In [11]: A.condition_number()
```

Out[11]: $\dfrac{\sqrt{27+2\sqrt{170}}}{\sqrt{27-2\sqrt{170}}}$

```
In [12]: sympy.N(_)
Out[12]: 7.58240137440151
In [13]: A.norm()
```

Out[13]: $3\sqrt{6}$



# Solution avec sympy

```
In [19]: A = sympy.Matrix([[2, 3], [5, 4]])
In [20]: b = sympy.Matrix([4, 3])
In [21]: L, U, _ = A.LUdecomposition()
In [22]: L
```

Out[22]: $\begin{bmatrix} 1 & 0 \\ 5/2 & 1 \end{bmatrix}$

```
In [23]: U
```

Out[23]: $\begin{bmatrix} 2 & 3 \\ 0 & -7/2 \end{bmatrix}$

```
In [24]: L * U
```

Out[24]: $\begin{bmatrix} 2 & 3 \\ 5 & 4 \end{bmatrix}$

```
In [25]: x = A.solve(b); x  # equivalent to A.LUsolve(b)
```

Out[25]: $\begin{bmatrix} -1 \\ 2 \end{bmatrix}$

# Solution avec numpy :np.linalg

```
In [14]: A = np.array([[2, 3], [5, 4]])
In [15]: b = np.array([4, 3])
In [16]: np.linalg.matrix_rank(A)
Out[16]: 2
In [17]: np.linalg.cond(A)
Out[17]: 7.5824013744
In [18]: np.linalg.norm(A)
Out[18]: 7.34846922835
```



# SciPy: Algèbre linéaire

**Inversion de la matrice :**

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')
>>> A
matrix([[1, 3, 5],
        [2, 5, 1],
        [2, 3, 8]])
>>> A.I
matrix([[-1.48,  0.36,  0.88],
        [ 0.56,  0.08, -0.36],
        [ 0.16, -0.12,  0.04]])
>>> from scipy import linalg
>>> linalg.inv(A)
array([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
```

# Solution avec scipy : scipy.linalg

import scipy.linalg as la

```
In [26]: A = np.array([[2, 3], [5, 4]])
In [27]: b = np.array([4, 3])
In [28]: P, L, U = la.lu(A)
In [29]: L
Out[29]: array([[ 1. ,  0. ],
                [ 0.4,  1. ]])
In [30]: U
Out[30]: array([[ 5. ,  4. ],
                [ 0. ,  1.4]])
In [31]: P.dot(L.dot(U))
Out[31]: array([[ 2., 3.],
                [ 5., 4.]])
In [32]: la.solve(A, b)
Out[32]: array([-1.,  2.])
```



# SciPy: Algèbre linéaire

**Résoudre :**

$$\begin{aligned} x+3y+5z &= 10 \\ 2x+5y+z &= 8 \\ 2x+3y+8z &= 3 \end{aligned}$$

S = $A^{-1}$ b où S=[ x y z] et b = [ 10 8 3]

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')
>>> b = mat('[10;8;3]')
>>> A.I*b
matrix([[-9.28],
        [ 5.16],
        [ 0.76]])
>>> linalg.solve(A,b)
array([[-9.28],
       [ 5.16],
       [ 0.76]])
```

# SciPy: Algèbre linéaire

**Trouver Determinant :**

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

$$\begin{aligned} |\mathbf{A}| &= 1\begin{vmatrix} 5 & 1 \\ 3 & 8 \end{vmatrix} - 3\begin{vmatrix} 2 & 1 \\ 2 & 8 \end{vmatrix} + 5\begin{vmatrix} 2 & 5 \\ 2 & 3 \end{vmatrix} \\ &= 1(5\cdot 8 - 3\cdot 1) - 3(2\cdot 8 - 2\cdot 1) + 5(2\cdot 3 - 2\cdot 5) = -25. \end{aligned}$$

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')
>>> linalg.det(A)
-25.000000000000004
```