Module C2

Calcul matriciel avec Python 3
Nouvelle édition

Luc Baron, ing., Ph.D.
Professeur titulaire

24 janvier 2024

Département de génie mécanique
Polytechnique Montréal

Table des matières

Chapitre 1

Systèmes Algébriques Linéaires

Dans ce chapitre, nous rappelons quelques notions de base sur l'algèbre linéaire utiles pour décrire et résoudre des systèmes d'équations linéaires. La bibliothèque `numpy` de Python est utilisée pour manipuler des vecteurs, des matrices, et résoudre des systèmes d'équations linéaires.

1.1 Vecteurs

L'ensemble $\mathbb{R}^n$ est l'ensemble de tous les n-tuples des nombres réels. Par exemple, $\mathbb{R}^3$ est l'ensemble des triplets réels, soient les coordonnées (x, y, z) de l'espace Cartésien de dimension 3. Un vecteur $\mathbf{u}$ de $\mathbb{R}^n$ peut être écrit horizontalement $\mathbf{u} = [u_1, u_2, ..., u_n]$, c'est-à-dire une ligne où chacun des éléments u_i est disposé de gauche à droite. Un vecteur $\mathbf{v}$ de $\mathbb{R}^n$ peut aussi écrit verticalement $\mathbf{v} = [v_1, v_2, ..., v_n]^T$, c'est-à-dire une colonne où chacun des éléments v_i est disposé de haut en bas. Ainsi, nous avons

$$\mathbf{u} = [u_1, u_2, ..., u_n] = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}^T \quad \text{et} \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = [v_1, v_2, ..., v_n]^T \tag{1.1}$$

où l'exposant T indique la transposée du vecteur. Par exemple, $\mathbf{u}^T$ est un vecteur colonne. Lorsque le contexte d'un vecteur est ambigu, il est habituel de considérer que celui-ci est un vecteur colonne.

Script 1
Définition d'un vecteur ligne $\mathbf{u}$ et des vecteurs colonnes $\mathbf{v}$ et $\mathbf{w}$

```
import numpy as np
u = np.array([[1,2,3]])            #Vecteur ligne
v = np.array([[4],[5],[6]])        #Vecteur colonne
w = np.array([7,8,9]).reshape(3,1) #Transforme une liste simple en vecteur colonne
print(u.shape, v.shape, w.shape, u + 2*v.T + 3*w.T) #Dimension et combinaison lineaire
```

```
(1,3) (3,1) (3,1) [[30, 36, 42]]
```

Le script 1 défini le vecteur ligne $\mathbf{u}$ et les vecteurs colonnes $\mathbf{v}$ et $\mathbf{w}$ tous de $\mathbb{R}^3$. La méthode `reshape(3,1)` permet de transformer la liste simple initiale en vecteur colonne. La méthode `shape` permet d'obtenir les dimensions de l'objet `numpy`. Finalement, on calcul la somme des vecteurs lignes : $\mathbf{u} + 2 * \mathbf{v}^T + 3 * \mathbf{w}^T$ *élément-par-élément*.

La norme d'un vecteur $\mathbf{v}$ est une mesure de longueur dans $\mathbb{R}^n$. Bien qu'il existe plusieurs façons de définir cette longueur, la norme Euclidienne L_2 est celle-la plus communément utilisée parce qu'elle correspond à la longueur physique d'un vecteur dans l'espace $\mathbb{R}^3$, c'est-à-dire :

$$\text{Norme :} \quad \|\mathbf{v}\| = \sqrt{\sum v_i^2}, \quad fori = 1, ..., n, \quad \mathbf{v} \in \mathbb{R}^n \tag{1.2}$$

Le produit scalaire des vecteurs $\mathbf{u}$ et $\mathbf{v}$ est la somme des produits des éléments respectifs des deux vecteurs de même dimension. Ainsi, nous avons :

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^T \mathbf{v} = \sum_{i=1}^{n} u_i v_i, \quad \mathbf{u}, \mathbf{v} \in \mathbb{R}^n \tag{1.3}$$

De plus, le produit scalaire $\mathbf{u} \cdot \mathbf{v}$ est relié aux longueurs $\|\mathbf{u}\|$ et $\|\mathbf{v}\|$, ainsi qu'à l'angle θ entre ceux-ci

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos\theta \tag{1.4}$$

SCRIPT 2
Norme et produit scalaire de 2 vecteurs (sous forme de liste simple et de vecteur colonne)

```
1  import numpy as np
2  from numpy.linalg import norm
3  u = np.array([1,2,3])    #Liste simple u
4  v = np.array([7,8,9])    #Liste simple v
5  print('Liste: norm(u)=%.2f'%norm(u),' norm(v)=%.2f'%norm(v))
6  print('u.v=%.2f'%np.dot(u,v),'u^T.v=%.2f'%np.dot(u.T,v),' u.v^T=%.2f'%np.dot(u,v.T),
7        ' theta=%.2f'%(np.arccos(np.dot(u,v.T)/(norm(u)*norm(v)))*180/np.pi))
8  u = u.reshape(3,1)    #Vecteur colonne u
9  v = v.reshape(3,1)    #Vecteur colonne v
10 print('Vecteurs colonnes: norm(u)=%.2f'%norm(u),' norm(v)=%.2f'%norm(v))
11 print('u^T.v=%.2f'%np.dot(u.T,v),
12       ' theta=%.2f'%(np.arccos(np.dot(u.T,v)/(norm(u)*norm(v)))*180/np.pi))
13 print('u.v^T=', np.dot(u,v.T))
```

```
Liste: norm(u)=3.74 norm(v)=13.93
u.v=50.00 u^T.v=50.00 u.v^T=50.00 theta=16.38
Vecteurs colonnes: norm(u)=3.74 norm(v)=13.93
u^T.v=50.00 theta=16.38
u.v^T= [[ 7  8  9]
 [14 16 18]
 [21 24 27]]
```

Le script 2 calcul la norme et le produit scalaire des vecteurs $\mathbf{u}$ et $\mathbf{v}$ de $\mathbb{R}^3$ définis comme des listes simples aux lignes 3-4, puis comme vecteurs colonnes aux lignes 8-9. La fonction `norm()` permet de calculer la norme L_2 d'un vecteur. Puisque la transposée n'affecte pas les listes simples, le calcul du produit scalaire (ligne 6) avec les fonctions `np.dot(u,v)`, `np.dot(u.T,v)` et `np.dot(u,v.T)` donnent toujours le même résultat. Cependant, lorsque des vecteurs lignes ou colonnes sont utilisés, il est nécessaire d'utiliser un produit scalaire compatible entre les vecteurs. Si $\mathbf{u}$ et $\mathbf{v}$ sont des vecteurs colonnes, `np.dot(u.T,v)` donne le produit scalaire (ligne 11), alors que `np.dot(u,v.T)` donne une matrice (ligne 13). Finalement, l'angle θ entre $\mathbf{u}$ et $\mathbf{v}$ est calculé avec l'éq(1.4).

Le produit vectoriel de $\mathbf{u}$ par $\mathbf{v}$ est défini dans $\mathbb{R}^3$ comme

$$\mathbf{u} \times \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \sin\theta \ \mathbf{n}, \tag{1.5}$$

où le vecteur unitaire $\mathbf{n}$ est perpendiculaire à $\mathbf{u}$ et $\mathbf{v}$ (le sens positif selon la règle de la main droite) . Il peut facilement être calculé avec la matrice produit vectoriel $\mathbf{U}_\times$, construite à partir ses éléments de $\mathbf{u}$, c'est-à-dire

$$\mathbf{u} \times \mathbf{v} = \mathbf{U}_\times \mathbf{v} = \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix} \tag{1.6}$$

SCRIPT 3
Calcul du produit vectoriel de $\mathbf{u} \times \mathbf{v}$ et $\mathbf{a} \times \mathbf{b}$

```
1 import numpy as np
2 u, v = np.array([1,2,3]), np.array([4,5,6])   #Listes simples
3 a, b = u.reshape(1,3), v.reshape(1,3)         #Vecteurs lignes
4 print('u x v =',np.cross(u,v), '; a x b =',np.cross(a,b) #Calcul u x v et a x b
```

```
u x v = [-3 6 -3] ; a x b = [-3 6 -3]
```

Le script 3 calcule $\mathbf{u} \times \mathbf{v}$ avec des listes simples, puis $\mathbf{a} \times \mathbf{b}$ avec des vecteurs lignes. En Python, la fonction `cross()` permet de calculer le produit vectoriel de listes simples ou vecteurs lignes de $\mathbb{R}^3$. Cette fonction n'accepte pas les vecteurs colonnes.

1.2 Matrices

Une matrice est une table de nombres constituée de m lignes et de n colonnes. L'élément de $\mathbf{A}$ situé à la ligne i et colonne j est dénoté a_{ij}. Il est possible de combiner des matrices *élément-par-élément*, si elles sont de mêmes dimensions, en plus d'une multiplication par un scalaire. Soit les matrices $\mathbf{A}$ et $\mathbf{B}$ de dimension 2×3

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 1 & 2 \\ 4 & 3 & 1 \end{bmatrix}, \tag{1.7}$$

alors $(2\mathbf{A} + 3\mathbf{B}) * \mathbf{A}$ se calcul *élément-par-élément* de la façon suivante :

$$(\begin{bmatrix} 2 & 4 & 6 \\ 4 & -2 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 3 & 6 \\ 12 & 9 & 3 \end{bmatrix}) * \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 14 & 36 \\ 32 & -7 & 0 \end{bmatrix} \tag{1.8}$$

Le produit matriciel *ligne-par-colonne* peut être effectué si les dimensions des matrices sont compatibles. Par exemple, le produit d'une matrice 2×3 et d'une matrice 3×2 donne une matrice 2×2, le nombre de colonnes de la première matrice (3 colonnes) doit être identique au nombre de lignes de la seconde matrice (3 lignes). Par exemple, le produit de matrice $\mathbf{C} = \mathbf{A}\mathbf{B}^T$ se calcule

$$\mathbf{C} = \mathbf{A}\mathbf{B}^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 4 \\ 1 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 13 \\ -1 & 5 \end{bmatrix} \tag{1.9}$$

Le script 4 calcule la combinaison des matrices $\mathbf{A}$ et $\mathbf{B}$ *élément-par-élément* selon les eqs. (1.7), puis le produit matriciel *ligne-par-colonne* $\mathbf{C} = \mathbf{A}\mathbf{B}^T$ selon l'eq.(1.9). En Python, la fonction `np.dot()` de `numpy` permet de calculer le produit *ligne-par-colonne* de vecteurs et matrices compatibles.

SCRIPT 4
Combinaison de matrices *élément-par-élément* et *ligne-par-colonne*

```
import numpy as np
A = np.array([[1, 2, 3],[2, -1, 0]])
B = np.array([[0, 1, 2],[4, 3, 1]])
print('Combinaison (2A + 3B) * A\n',(2*A+3*B)*A)      #Produit element-par-element
print('Produit matriciel C = A B^T\n',np.dot(A,B.T)) #Produit ligne-par-colonne
```

```
Combinaison (2A + 3B) * A
 [[ 2 14 36]
 [32 -7  0]]
Produit matriciel C = A B^T
 [[ 8 13]
 [-1  5]]
```

Une matrice carrée $\mathbf{A}$ est une matrice ayant le même nombre de lignes que de colonnes. Le déterminant est une propriété importante des matrices carrées qui se calcule directement avec les éléments de la matrice. Dans le cas d'une matrice 2×2, nous avons

$$\det(\mathbf{A}) = \det(\begin{bmatrix} a & b \\ c & d \end{bmatrix}) = ad - bc \tag{1.10}$$

En Python, la fonction `det()` de la bibliothèque `linalg` de `numpy` permet de calculer le déterminant d'une matrice carrée. Une matrice identité $\mathbf{1}$ est une matrice carrée ayant des 1 sur sa diagonale et des zéros ailleurs. La matrice identité est analogue au nombre 1, c'est-à-dire que la multiplication d'une matrice par celle-ci produit la matrice initiale. En Python, la fonction `np.eye(n)` permet de produire la matrice identité $n \times n$. L'inverse d'une matrice carrée $\mathbf{A}$, dénoté $\mathbf{A}^{-1}$, est une matrice carrée de même dimension que $\mathbf{A}$, telle que

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{1} \tag{1.11}$$

Dans le cas de la matrice 2×2 de l'éq.(1.10), nous avons

$$\mathbf{A}^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{\det(\mathbf{A})} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \tag{1.12}$$

On vérifie que $\mathbf{A}\mathbf{A}^{-1} = \mathbf{1}$

$$\mathbf{A}\mathbf{A}^{-1} = \frac{1}{ad-bc} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{ad-bc} \begin{bmatrix} ad-bc & -ab+ab \\ cd-cd & ad-bc \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{1} \tag{1.13}$$

Selon l'éq.(1.12), il n'est pas possible de calculer l'inverse d'une matrice carrée lorsque son déterminant est égal à zéro. En Python, la fonction `inv()` de la bibliothèque `linalg` de `numpy` permet de calculer l'inverse de $\mathbf{A}$.

1.3 Systèmes linéaires

Une équation linéaire est de la forme

$$\sum_{i=1}^{n} a_i x_i = b \tag{1.14}$$

où les a_i et b sont des paramètres scalaires connus, alors que les x_i sont des variables scalaires inconnues. Un système d'équations linéaires est un ensemble de m équations linéaires qui partage le même ensemble de n variables inconnues $\mathbf{x} = [x_1, x_2, ..., x_n]^T$. avec m paramètres connus $\mathbf{b} = [b_1, b_2, ..., b_m]^T$ et $m \times n$ paramètres connus a_{ij}, c'est-à-dire

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + ... + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + ... + a_{2n}x_n &= b_2 \\ ... \quad ... \quad ... \quad ... \quad ... &= ... \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 + ... + a_{mn}x_n &= b_m \end{aligned}$$

On peut réécrire ce système d'équations linéaires sous forme matriciel

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1.15}$$

avec

$$\mathbf{A} \equiv \begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & ... & a_{mn} \end{bmatrix}, \quad \mathbf{x} \equiv \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{et} \quad \mathbf{b} \equiv \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}. \tag{1.16}$$

Il existe 3 solutions au système (1.15). En Python, la fonction `solve(A,b)` de la bibliothèque `linalg` de `numpy` permet de résoudre (1.15) de façon précise et numériquement stable lorsque $\mathbf{A}$ est carré et de plein rang.

1.3.1 Solution unique (m = n)

Il existe une *solution unique* lorsque les m équations de (1.15) sont linéairement indépendantes, c'est-à-dire que le rang($\mathbf{A}$) = m. La solution $\mathbf{x}$ se calcule avec l'inverse de $\mathbf{A}$ par

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{1.17}$$

En Python, la fonction `inv()` de la bibliothèque `linalg` de `numpy` permet de calculer l'inverse de $\mathbf{A}$.

1.3.2 Solution des moindres carrées (m > n)

Il n'existe aucune solution permettant de satisfaire toutes les m équations avec les n variables inconnues. Il faut plutôt calculer la solution qui satisfait au mieux toutes les équations avec une erreur $\mathbf{e}$ minimale, soit la *solution des moindres carrées*. Ainsi

$$z = \frac{1}{2}\|\mathbf{e}\|^2 = \frac{1}{2}\mathbf{e}^T\mathbf{e} \rightarrow \min_{\mathbf{x}} \tag{1.18}$$

où l'erreur $\mathbf{e}$ sur les m équations est définie par

$$\mathbf{e} \equiv \mathbf{b} - \mathbf{A}\mathbf{x} \tag{1.19}$$

La condition de normalité du problème (1.18) est obtenue en égalisant à zéro le gradient de z relativement à $\mathbf{x}$, puis en utilisant la règle de dérivation en chaîne, qui donne

$$\mathbf{0} = \frac{dz}{d\mathbf{x}} = \left(\frac{d\mathbf{e}}{d\mathbf{x}}\right)^T \frac{dz}{d\mathbf{e}} = -\mathbf{A}^T\mathbf{e} \tag{1.20}$$

En substituant (1.19) dans (1.20), nous obtenons

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b} \tag{1.21}$$

En vertu de l'hypothèse de $\text{rang}(\mathbf{A}) = m$, le produit $\mathbf{A}^T\mathbf{A}$ est carrée et inversible, et donc, la solution $\mathbf{x}$ qui minimise la norme Euclidienne de l'erreur d'approximation $\mathbf{e}$ est

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \tag{1.22}$$

Il est habituel de réécrire ce résultat en définissant le pseudoinverse gauche de $\mathbf{A}$, dénoté $\mathbf{A}^p$, par

$$\mathbf{x} = \mathbf{A}^p\mathbf{b}, \quad \mathbf{A}^p \equiv (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T \tag{1.23}$$

En Python, la fonction `pinv()` de la bibliothèque `linalg` de `numpy` permet de calculer le pseudoinverse de $\mathbf{A}$.

1.3.3 Solution de norme minimale (m < n)

Il existe une infinité de solutions permettant de satisfaire les m équations avec les n variables inconnues. Il faut alors choisir une solution parmi l'ensemble des solutions admissibles. Un choix simple est de calculer la solution, dont la norme des inconnus est la plus petite, soit la *solution de norme minimale*.

$$z(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|^2 \rightarrow \min_{\mathbf{x}} \quad \text{sujet à} \quad \mathbf{A}\mathbf{x} = \mathbf{b} \tag{1.24}$$

Le problème d'optimisation (1.24) est sous contrainte. Nous utilisons les multiplicateurs de Lagrange $\boldsymbol{\lambda}$ pour introduire une nouvelle fonction d'optimisation $\zeta(\mathbf{x})$ sans contrainte

$$\zeta(\mathbf{x}) = z(\mathbf{x}) + \boldsymbol{\lambda}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) \rightarrow \min_{\mathbf{x}}, \tag{1.25}$$

dont $\boldsymbol{\lambda}$ est encore à déterminer. En suivant une procédure similaire au cas précédent, la solution qui minimise la norme Euclidienne des inconnus $\mathbf{x}$ est

$$\mathbf{x} = \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1}\mathbf{b} \tag{1.26}$$

Il est habituel de réécrire ce résultat en définissant le pseudoinverse droit de $\mathbf{A}$, aussi dénoté $\mathbf{A}^p$, par

$$\mathbf{x} = \mathbf{A}^p\mathbf{b}, \quad \mathbf{A}^p \equiv \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1} \tag{1.27}$$

De façon surprenante, la solution de norme minimale est très similaire à la solution des moindres carrées. Plusieurs auteurs utilisent le pseudoinverse gauche et le pseudoinverse droit pour désigner les deux solutions. En Python, la fonction `pinv()` de la bibliothèque `linalg` de `numpy` permet de détecter automatiquement la bonne version du pseudoinverse à calculer.

Exemple 1 : Résoudre le système de 3 équations à 3 inconnus.

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 3 \\ x_1 + 2x_2 + x_3 &= 1 \\ 2x_1 + x_2 + x_3 &= 0 \end{aligned}$$

On peut réécrire le système d'équations sous forme matriciel, tel que $\mathbf{Ax} = \mathbf{b}$

$$\begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 0 \end{bmatrix},$$

où la solution $\mathbf{x}$ est obtenue par $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Puisque le $\det(\mathbf{A}) = -4 \neq 0$, nous avons

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \frac{1}{-4} \begin{bmatrix} 1 & 1 & -3 \\ 1 & -3 & 1 \\ -3 & 1 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 2 \end{bmatrix} \quad \Rightarrow \quad \begin{cases} x_1 = -1 \\ x_2 = 0 \\ x_3 = 2 \end{cases}$$

Le script 5 calcule la solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, puis résout directement le système $\mathbf{Ay} = \mathbf{b}$. Les erreurs d'arrondies sont moindres pour `y`, la valeur de la seconde variable est un peu plus près de zéro.

SCRIPT 5
Exemple : Résolution d'un système de 3 équations à 3 inconnus

```
import numpy as np
from numpy.linalg import det, inv, solve
A=np.array([[1, 1, 2],[1, 2, 1],[2, 1, 1]]) #Matrice 3x3
b=np.array([3, 1, 0]).reshape(3,1)          #Vecteur colonne
print('A=\n',A)
print('b^T=',b.T)
print('det(A)=%.1f'%det(A))
Ai=inv(A)
print('inv(A)=\n',Ai)
x=np.dot(Ai,b)    #Calcul x = inv(A) * b
print('x^T=',x.T) #Un peu moin precis
y=solve(A,b)      #Resout le systeme Ay=b
print('y^T=',y.T) #Un peu plus precis
```

```
A=
 [[1 1 2]
 [1 2 1]
 [2 1 1]]
b^T= [[3 1 0]]
det(A)=-4.0
inv(A)=
 [[-0.25 -0.25 0.75]
 [-0.25 0.75 -0.25]
 [ 0.75 -0.25 -0.25]]
x^T= [[-1.00000000e+00 2.22044605e-16 2.00000000e+00]]
y^T= [[-1.00000000e+00 7.40148683e-17 2.00000000e+00]]
```

Chapitre 2

Approximation Polynomiale

Dans ce chapitre, nous établissons les bases de l'approximation polynomiale. La méthode de *Vandermonde* est utilisée pour calculer les coefficients du polynôme d'approximation de points (x_i, y_i) sans contrainte ou avec contrainte de pente t_j à l'abcisse x_j. En général, un polynôme $p(x) = y$ de degré n nécessite $n+1$ coefficient a_i, tel que

$$p_n(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n = y \tag{2.1}$$

Il peut être écrit sous la forme du produit scalaire de deux vecteurs de dimension $n+1$

$$p_n(x) = \begin{bmatrix} x^n & x^{n-1} & \cdots & x & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} = y \tag{2.2}$$

En Python, la fonction `np.polyval(a,x)` de `numpy` permet d'évaluer le polynôme de l'éq.(2.2).

2.1 Polynômes de degré 1 (linéaire)

Soit $S = \{(x_i, y_i)\}_1^m$, l'ensemble de m points de $\mathbb{R}^2$. On désire calculer les coefficients du polynôme de degré $n = 1$ (droite) qui approxime au mieux S. Nous avons un système de m polynômes

$$p_1(x_i) = a_0 x_i + a_1 = \begin{bmatrix} x_i & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = y_i, \quad i = 1, \ldots, m \tag{2.3}$$

qui peut être écrit sous forme matricielle

$$\mathbf{Ma} = \mathbf{y} \tag{2.4}$$

avec

$$\mathbf{M} \equiv \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix}, \quad \mathbf{a} \equiv \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad \text{et} \quad \mathbf{y} \equiv \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{2.5}$$

Si $m > 2$, la solution des moindres carrées se calcule avec le pseudoinverse

$$\mathbf{a} = \mathbf{M}^p \mathbf{y}, \quad \mathbf{M}^p \equiv (\mathbf{M}^T \mathbf{M})^{-1} \mathbf{M}^T \tag{2.6}$$

Si $m = 2$, $\mathbf{M}$ est carrée, la droite passe par les 2 points et la solution se calcul avec `solve(M,y)`.

2.2 Polynômes de degré 2 (quadratique)

Cette fois, on désire calculer les coefficients du polynôme de degré $n = 2$ (parabole) qui approxime au mieux le même ensemble S. Nous avons encore un système de m polynômes

$$p_2(x_i) = a_0 x_i^2 + a_1 x_i + a_2 = \begin{bmatrix} x_i^2 & x_i & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = y_i, \quad i = 1, ..., m \tag{2.7}$$

qui peut aussi être écrit sous forme matricielle

$$\mathbf{Ma} = \mathbf{y} \tag{2.8}$$

avec

$$\mathbf{M} \equiv \begin{bmatrix} x_1^2 & x_1 & 1 \\ \vdots & \vdots & \vdots \\ x_m^2 & x_m & 1 \end{bmatrix}, \quad \mathbf{a} \equiv \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \quad \text{et} \quad \mathbf{y} \equiv \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{2.9}$$

Si $m > 3$, la solution des moindres carrées se calcule avec `pinv(M).dot(y)`. Si $m = 3$, $\mathbf{M}$ est carrée, le polynôme passe par les 3 points et la solution se calcule avec `solve(M,y)`.

SCRIPT 6
Exemple : Vandermonde degre 1 et 2

```
import numpy as np
from numpy.linalg import pinv
xp = np.array([-1, 0,  1, 2 ])
yp = np.array([ 1, 0.5, 1.5, 2.5])
x  = np.linspace(-1,2,20)
M= np.vstack((xp**1,xp**0)).T    #Empile 2 lignes de 4 valeurs, puis transpose
a = pinv(M).dot(yp)              #Calcul a = M^p * yp
y1 = np.polyval(a,x)             #Calcul y1 = a_0 x + a_1
print('Degre 1: a.T=',a.T)
print('M=',M)                    #M est 4 x 2
M= np.vstack((xp**2, xp**1,xp**0)).T #Empile 3 lignes de 4 valeurs, puis transpose
a = pinv(M).dot(yp)
y2 = np.polyval(a,x)             #Calcul y2 = a_0 x^2 + a_1 x + a_3
print('Degre 2: a.T=',a.T)
print('M=',M)                    #M est 4 x 3
```

```
Degre 1: a.T= [0.55 1.1 ]
M= [[-1 1]
 [ 0  1]
 [ 1  1]
 [ 2  1]]
Degre 2: a.T= [0.375 0.175 0.725]
M= [[ 1 -1 1]
 [ 0  0  1]
 [ 1  1  1]
 [ 4  2  1]]
```

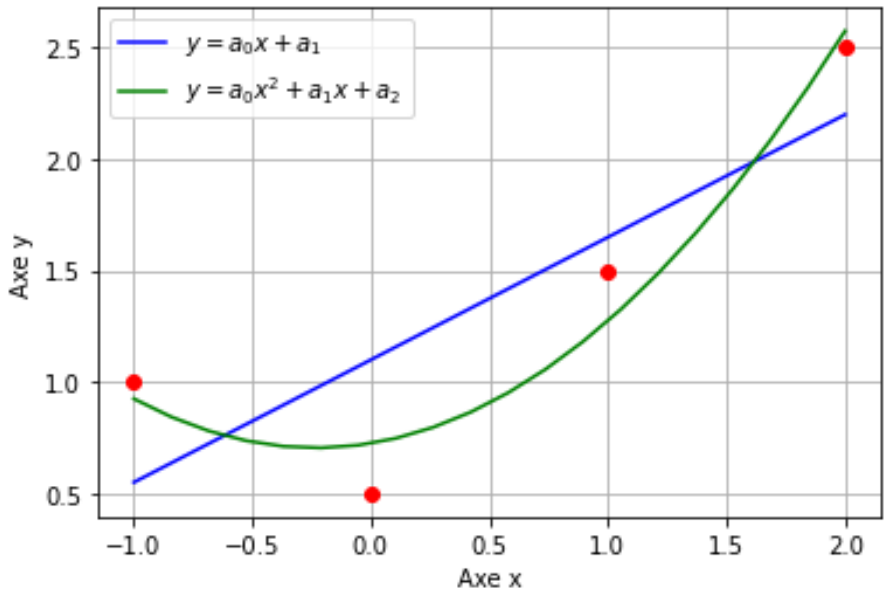

FIGURE 2.1 – Approximation linéaire et quadratique avec 4 points

2.3 Polynômes de degré n

D'une façon générale, on désire calculer les coefficients du polynôme de degré n qui approxime au mieux l'ensemble S. Nous avons un système de m polynômes

$$p_n(x_i) = a_0x_i^n + a_1x_i^{n-1} + \cdots + a_{n-1}x_i + a_n = y_i, \quad i = 1, ..., m \tag{2.10}$$

Il peut être écrit sous la forme du produit scalaire de deux vecteurs de dimension $n+1$

$$p_n(x_i) = \begin{bmatrix} x_i^n & x_i^{n-1} & \cdots & x_i & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} = y_i, \quad i = 1, ..., m \tag{2.11}$$

qui peut être réécrit sous forme matricielle

$$\mathbf{Ma} = \mathbf{y} \tag{2.12}$$

avec

$$\mathbf{M} \equiv \begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m & 1 \end{bmatrix}, \quad \mathbf{a} \equiv \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} \quad \text{et} \quad \mathbf{y} \equiv \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{2.13}$$

Si $\mathbf{M}$ est de plein rang avec $m > n$, la solution des moindres carrées se calcule par le pseudoinverse ou la résolution de (2.12) avec `solve(M,y)`. Si $n+1 = m$, alors $\mathbf{M}$ est carrée et le polynôme passe exactement par les m points. On dit alors qu'il s'agit d'un *polynôme d'interpolation* plutôt que d'un *polynôme d'approximation*. En Python, la fonction `np.polyfit(x,y,n)` permet de calculer les coefficients du polynôme de degré n avec les m points (x_i, y_i).

2.4 Approximation avec pente prescrite

En plus de S, on désire prescrire o pentes t_j aux valeurs d'abscisses x_j, soit l'ensemble $T = \{(x_j, t_j)\}_1^o$. Soit un polynôme de degré n

$$p_n(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n = y \tag{2.14}$$

La contrainte de pente t est obtenue en dérivant (2.14)

$$p'_n(x) = n a_0 x^{n-1} + (n-1) a_1 x^{n-2} + \cdots + a_{n-1} + 0 = t \tag{2.15}$$

avec

$$p'(x) = \frac{d}{dx} p(x) \quad \text{et} \quad t = \frac{dy}{dx} \tag{2.16}$$

Les o contraints de pente t_j de $T = \{(x_j, t_j)\}_1^o$ peuvent être écrites sous la forme d'un système de o produit scalaire de deux vecteurs de dimension $n+1$

$$p'_n(x_j) = \begin{bmatrix} n x_j^{n-1} & (n-1) x_j^{n-2} & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} = t_j, \quad j = 1, ..., o \tag{2.17}$$

qui peut être réécrit sous forme matricielle

$$\mathbf{N}\mathbf{a} = \mathbf{t} \tag{2.18}$$

avec

$$\mathbf{N} \equiv \begin{bmatrix} n x_1^{n-1} & (n-1) x_1^{n-2} & \cdots & 1 & 0 \\ n x_2^{n-1} & (n-1) x_2^{n-2} & \cdots & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ n x_o^{n-1} & (n-1) x_o^{n-2} & \cdots & 1 & 0 \end{bmatrix}, \quad \mathbf{a} \equiv \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} \quad \text{et} \quad \mathbf{t} \equiv \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_o \end{bmatrix} \tag{2.19}$$

Même si $o > n+1$, la matrice $\mathbf{N}$ est toujours singulière, la dernière colonne de zéros empêche toutes solutions pour a_n. En conséquence, le système de pentes prescrites $\mathbf{N}\mathbf{a} = \mathbf{t}$ doit toujours être utilisé avec un système de points prescrits $\mathbf{M}\mathbf{a} = \mathbf{y}$.

2.5 Approximation de points et pentes

En plus de calculer les coefficients du polynôme de degré n qui approxime au mieux l'ensemble des m points de $S = \{(x_i, y_i)\}_1^m$, on désire ajouter o pentes t_j prescrites aux valeurs d'abscisse x_j, soit l'ensemble $T = \{(x_j, t_j)\}_1^o$. Supposons un polynôme de degré $n = 3$.
Nous avons pour $S = \{(x_i, y_i)\}_1^m$

$$\mathbf{M}\mathbf{a} = \mathbf{y} \tag{2.20}$$

avec

$$\mathbf{M} \equiv \begin{bmatrix} x_1^3 & x_1^2 & x_1 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_m^3 & x_m^2 & x_m & 1 \end{bmatrix}, \quad \mathbf{a} \equiv \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \text{et} \quad \mathbf{y} \equiv \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{2.21}$$

Nous avons pour $T = \{(x_j, t_j)\}_1^o$

$$\mathbf{Na} = \mathbf{t} \tag{2.22}$$

avec

$$\mathbf{N} \equiv \begin{bmatrix} 3x_1^2 & 2x_1 & 1 & 0 \\ 3x_2^2 & 2x_2 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 3x_o^2 & x_o & 1 & 0 \end{bmatrix}, \quad \mathbf{a} \equiv \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \text{et} \quad \mathbf{t} \equiv \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_o \end{bmatrix} \tag{2.23}$$

En combinant les systèmes (2.20) et (2.22), nous obtenons

$$\mathbf{Ua} = \mathbf{v} \tag{2.24}$$

avec

$$\mathbf{U} \equiv \begin{bmatrix} \mathbf{M} \\ \mathbf{N} \end{bmatrix} \quad \text{et} \quad \mathbf{v} \equiv \begin{bmatrix} \mathbf{y} \\ \mathbf{t} \end{bmatrix} \tag{2.25}$$

pour lequel la solution des moindres carrées pour $\mathbf{a}$ se calcule avec le pseudoinverse ou par résolution directe de $\mathbf{Ua} = \mathbf{v}$. La fonction Python `polyfit()` ne permet pas de résoudre un problème mixte de points et pentes prescrites. Dans ce cas, il faut utiliser la fonction générale d'approximation de courbe `optimize.curve_fit(func,xdata=x,ydata=y)` de la bibliothèque `scipy`.

Exemple 2 : Calculer le polynôme d'interpolation de degré 4 qui passe par 4 points et 1 pentes.

$$\begin{aligned} S &= \{(-1,1),(0,0.5),(1,1.5),(2,2.5)\} \\ T &= \{(1,0)\} \end{aligned}$$

Puisqu'un polynôme de degré 4 a 5 coefficients, il est possible de satisfaire entièrement les 5 conditions, soit 4 points et 1 pente. La matrice $\mathbf{U}$ est une 5×5 et peut être résolue avec la fonction `solve(U,v)`.

Pour S, nous avons

$$\begin{aligned} \mathbf{M} &= \begin{bmatrix} x_1^4 & x_1^3 & x_1^2 & x_1^1 & x_1^0 \\ x_2^4 & x_2^3 & x_2^2 & x_2^1 & x_2^0 \\ x_3^4 & x_3^3 & x_3^2 & x_3^1 & x_3^0 \\ x_4^4 & x_4^3 & x_4^2 & x_4^1 & x_4^0 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 16 & 8 & 4 & 2 & 1 \end{bmatrix} \\ \mathbf{y} &= \begin{bmatrix} y_1 & y_2 & y_3 & y_4 \end{bmatrix}^T = \begin{bmatrix} 1 & 0.5 & 1.5 & 2.5 \end{bmatrix}^T \end{aligned}$$

sachant que $x_i^1 = x_i$ et $x_i^0 = 1$.

Pour T, nous avons

$$\begin{aligned} \mathbf{N} &= \begin{bmatrix} 4x_1^3 & 3x_1^2 & 2x_1^1 & x_1^0 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 3 & 2 & 1 & 0 \end{bmatrix} \\ \mathbf{t} &= \begin{bmatrix} t_1 \end{bmatrix} = \begin{bmatrix} 0 \end{bmatrix} \end{aligned} \tag{2.26}$$

sachant que $x_1 = 1$ et $t_1 = 0$.

Le script 7 affecte les coordonnées des points aux variables `xp` et `yp`, ainsi que l'abscisse `xt=1` et la pente `tt=0`. La matrice $\mathbf{M}$ est construite par empilements de 5 lignes `xp` de puissance 4 à la puissance 0, puis transposée, afin d'obtenir une matrice 4×5. La matrice $\mathbf{N}$ est construite empilement des 5 puissances de la valeur de $x_1 = 1$, puis transposée, afin d'obtenir une matrice 1×5 selon (2.26).

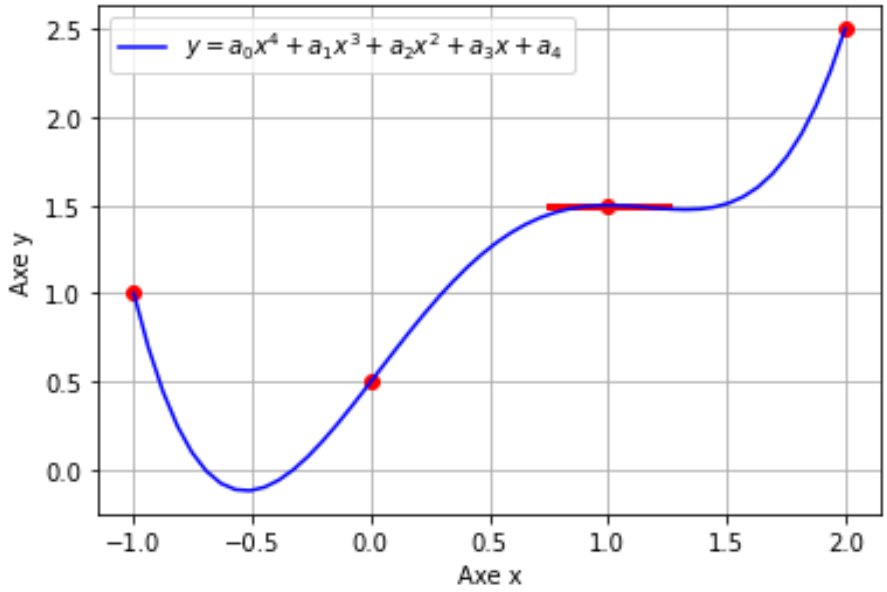

FIGURE 2.2 – Polynôme d'interpolation de degré 4 passant par 4 points et 1 pente

SCRIPT 7
Exemple : Polynôme d'interpolation de degré 4 passant par 4 points et 1 pente

```
import numpy as np
from numpy.linalg import pinv, solve
import matplotlib.pyplot as plt
xp = np.array([-1, 0,  1, 2 ])  #Points prescrits
yp = np.array([ 1, 0.5, 1.5, 2.5])
xt, tt = 1, 0                   #Pente prescrites x=1 avec t=0
x  = np.linspace(-1,2,50)
M = np.vstack((xp**4,xp**3,xp**2,xp**1,xp**0)).T   #Empile 5 lignes, puis transpose
N = np.vstack((4*xt**3,3*xt**2,2*xt**1,xt**0,0)).T #Empile 5 lignes, puis transpose
U = np.vstack((M,N))                               #Empile M sur N
v = np.append(yp,tt)                               #Ajoute tt a la fin de yp
a = solve(U,v)        #U est carre 5x5
y = np.polyval(a,x)   #Calcul y = a_0 x^4 + a_1 x^3 + a_3 x^2 + a_4 x + a_5
print('Degre 4: a.T=',a.T)
print('U=',U)
```

```
Degre 4: a.T= [ 0.625 -1.5 0.125  1.75   0.5  ]
U= [[ 1 -1  1 -1  1]
 [ 0  0  0  0  1]
 [ 1  1  1  1  1]
 [16  8  4  2  1]
 [ 4  3  2  1  0]]
```