

Projet d'Algèbre Linéaire

M1 MIASHS

Théorie & Implémentation Python

Livrables attendus pour le projet Python: un Notebook Jupyter.

Objectifs pédagogiques

- Mobiliser les notions fondamentales : systèmes linéaires, espaces vectoriels, noyau, image, rang, changements de bases, diagonalisation, matrices orthogonales, théorème spectral.
- Résoudre un *problème appliqué* en modélisant par une matrice, en diagonalisation et en interprétant les résultats (comportement des matrices, asymptotique, états stationnaires).
- Implémenter en Python : produire des fonctions d'analyse algébrique, résolution et comparaison numérique aux résultats renvoyés par les packages usuels (e.g., `numpy`, `scipy`).

Barème indicatif.

Partie I — Théorie	20 pts
1. Noyau, image, rang, théorème du rang	6 pts
2. Matrices orthogonales & théorème spectral	4 pts
3. Système dynamique, diagonalisation, passage, inversion (Gauss)	10 pts
Partie II — Implémentation Python	20 pts
1 Gauss, inversion, résolution linéaire	6 pts
2 Orthogonalité, spectral (symétrique), comparaison <code>numpy</code>	4 pts
3 Étude du système dynamique (simulation) & analyse	8 pts

Règles. Le livrable de l'implémentation Python est un travail **individuel ou en binôme**. Justifier vos réponses en citant le cours ou toute source académique.

1 Théorie (20 pts)

Un examen sur papier d'une durée de 2h est prévu pour la partie théorique. Il s'agira d'étudier certaines matrices présentées dans le sujet Python, distribué en amont de l'examen écrit. Cela permettra ainsi :

- de vous évaluer sur l'approche théorique et académique des bases de l'algèbre linéaire ;
- de mettre en "théorie" la pratique encodé en Python.

2 Implémentation Python (20 pts)

Le livrable attendu est (au moins) un Notebook `projet_algebre_NomPrenom.ipynb` *exécutable sans erreur*, présentant vos résultats, figures éventuelles et commentaires.

EXERCICE 1 : Pivot de Gauss *from scratch* : résolution et inversion

Objectif. Implémenter l'algorithme d'élimination de Gauss *avec pivot partiel* pour

- (i) résoudre $Ax = b$;
 - (ii) inverser une matrice par Gauss–Jordan,
- puis **valider** numériquement vos résultats face à `numpy`.

Travail demandé.

Q1. Résolution linéaire *from scratch*. Implémentez une fonction

$$\text{gauss_solve}(A, b) \rightarrow \hat{x}$$

qui :

- vérifie la compatibilité des dimensions ($A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$);
- effectue l'élimination directe avec *pivot partiel* (triangulation supérieure);
- réalise la *remontée* (back-substitution) pour obtenir \hat{x} ;
- lève une exception claire si A est (numériquement) singulière.

Sorties attendues : solution \hat{x} et, pour validation, le résidu $\|A\hat{x} - b\|_2$.

Q2. Inverse par Gauss–Jordan *from scratch*. Implémentez une fonction

$$\text{inverse_via_gauss}(A) \rightarrow \widehat{A^{-1}}$$

qui :

- vérifie que A est carrée;
- construit la matrice augmentée $(A | I)$;
- applique *Gauss–Jordan avec pivot partiel* jusqu'à obtenir $(I | A^{-1})$;
- lève une exception si A est (numériquement) non inversible.

Sorties attendues : $\widehat{A^{-1}}$ et les normes de contrôle $\|A\widehat{A^{-1}} - I\|_2$, $\|\widehat{A^{-1}}A - I\|_2$.

(BONUS.) Validation contre NumPy. Comparez vos résultats aux fonctions usuelles :

- Générez plusieurs matrices tests (aléatoires $n = 3, 4, 8$, matrices mal conditionnées type Hilbert, matrices avec permutations de lignes).
- Pour chaque test, comparez :

$$\hat{x} \text{ vs } \text{numpy.linalg.solve}(A, b), \quad \widehat{A^{-1}} \text{ vs } \text{numpy.linalg.inv}(A).$$

- Rapportez : erreurs relatives $\frac{\|\hat{x} - x_{np}\|_2}{\|x_{np}\|_2}$, $\frac{\|\widehat{A^{-1}} - A_{np}^{-1}\|_2}{\|A_{np}^{-1}\|_2}$, résidus $\|A\hat{x} - b\|_2$, et produits $\widehat{A}A^{-1} / \widehat{A^{-1}}A$ proches de I .

Conseils : fixez une `seed`, affichez les métriques dans un tableau lisible, commentez les cas où la tolérance est dépassée (conditionnement).

Contraintes.

- **Interdit** pour le calcul principal : `numpy.linalg.solve`, `numpy.linalg.inv`. Elles ne servent que pour *vérifier* vos sorties.
- Gestion des matrices singulières ou quasi singulières : lever une `ValueError` claire si le pivot inférieure à une certaine tolérance préfixée..
- **Docstrings en anglais ou en français** et commentaires concis expliquant chaque étape (pivot, échanges de lignes, élimination, remontée).

Squelettes de fonctions.

```
1 def gauss_solve(A: np.ndarray, b: np.ndarray, tol: float = 1e-12) -> np.ndarray:
2     """
3     Parameters
4     -----
5     Solve  $Ax=b$  via Gaussian elimination (partial pivoting).
6     A: np.ndarray Square matrix (n, n), real-valued.
7     b: np.ndarray Right-hand side (n,) or (n,1).
8     tol: float pivot threshold to detect (near-)singularity.
9
10    Returns
11    -----
12    x: np.ndarray Solution vector of shape (n,); raises ValueError if A
13        is (near-)singular.
14    """
15    A = np.array(A, dtype=float, copy=True)
16    b = np.array(b, dtype=float, copy=True).reshape(-1)
17    n = A.shape[0]
18    # Forward elimination with partial pivoting
19    for k in range(n):
20        # Choose pivot row if needed
21        ...
22        # Swap rows
23        if pivot != k:
24            ...
25        # Eliminate below
26        for i in range(k + 1, n):
27            ...
28    # Back substitution
29    x = np.zeros(n)
30    for i in range(n - 1, -1, -1):
31        ...
32    return x
33
34 def inverse_via_gauss(A: np.ndarray, tol: float = 1e-12) -> np.ndarray:
35     """
36     Compute  $A^{-1}$  via Gauss-Jordan on  $(A|I)$ .
37     A: (n,n) real; tol: pivot threshold.
38     Returns  $A^{-1}$  (n,n); raises ValueError if (near-)singular.
39     """
40    A = np.array(A, dtype=float, copy=True)
41    n = A.shape[0]
42    M = np.hstack([A, np.eye(n)]) # (A | I)
43    # Gauss-Jordan
44    for k in range(n):
45        ...
46    return ...
```

Consigne de rédaction — algorithme *pas à pas*

A. Résolution $x = \text{gauss_solve}(A, b)$ (pivot partiel).

1. **Préparation.** Copier A en flottants ; forcer b en vecteur colonne $(n,)$.
2. **Boucle d'élimination** pour $k = 0, \dots, n - 1$:
 - (a) **Choix du pivot.** Trouver l'indice de ligne

$$\text{pivot} = \arg \max_{i \in \{k, \dots, n-1\}} |A_{i,k}|.$$

Vérifier que $|A_{\text{pivot},k}| > \text{tol}$; sinon lever une `ValueError` (matrice singulière ou quasi singulière).

- (b) **Échange de lignes.** Si $\text{pivot} \neq k$, échanger les lignes k et pivot dans A et dans b .
- (c) **Élimination sous le pivot.** Pour chaque $i = k + 1, \dots, n - 1$:

$$m \leftarrow \frac{A_{i,k}}{A_{k,k}}, \quad A_{i,k:} \leftarrow A_{i,k:} - m A_{k,k:}, \quad b_i \leftarrow b_i - m b_k.$$

3. **Remontée (substitution arrière).** Initialiser $x \in \mathbb{R}^n$ à 0.

$$\text{Pour } i = n - 1, \dots, 0 : \quad s \leftarrow \sum_{j=i+1}^{n-1} A_{i,j} x_j, \quad \text{vérifier } |A_{i,i}| > \text{tol}, \quad x_i \leftarrow \frac{b_i - s}{A_{i,i}}.$$

4. **Retour.** Renvoyer x .

B. Inversion $A^{-1} = \text{inverse_via_gauss}(A)$ (Gauss–Jordan).

1. **Préparation.** Former la matrice augmentée $(A | I_n)$ en flottants.
2. **Boucle Gauss–Jordan** pour $k = 0, \dots, n - 1$:
 - (a) **Choix du pivot.** $\text{pivot} = \arg \max_{i \geq k} |A_{i,k}|$. Tester $|A_{\text{pivot},k}| > \text{tol}$, sinon `ValueError`.
 - (b) **Échange de lignes.** Si besoin, échanger les lignes k et pivot dans toute l'augmentée.
 - (c) **Normalisation de la ligne pivot.** Diviser toute la ligne k par $A_{k,k}$ afin d'avoir un 1 sur le pivot.
 - (d) **Annihilation des autres lignes.** Pour tout $i \neq k$:

$$\text{fact} \leftarrow A_{i,k}, \quad \text{ligne } i \leftarrow \text{ligne } i - \text{fact} \times \text{ligne } k.$$

3. **Retour.** Quand la partie gauche est réduite à I_n , la partie droite est A^{-1} : extraire et renvoyer le bloc droit.

Remarques.

- Le **pivot partiel** (max de $|A_{i,k}|$) améliore la stabilité numérique ; `tol` sert à détecter les pivots trop petits.
- Toujours **échanger** b (ou le bloc droit) quand on échange des lignes de A .
- Documenter chaque étape (docstrings en anglais) et **tester** : comparer avec `numpy.linalg.solve` / `inv` (uniquement pour valider).

EXERCICE 2 : Application pour matrice de covariance

Objectif. Vérifier numériquement le théorème spectral pour une matrice de covariance S , construire la décomposition $S = Q D Q^\top$ avec Q orthogonale et $D = \text{diag}(\lambda_1, \lambda_2, \lambda_3)$, puis *interpréter* les composantes principales en simulant des données de covariance proche de S .

Matrice étudiée.

$$S = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}.$$

Travail demandé.

Q1. Détections et décomposition spectrale (1 pt).

- (a) Implémentez `is_symmetric(S, tol)` et vérifiez $S^\top = S$.
- (b) Calculez une base orthonormée de vecteurs propres de S et formez Q (colonnes = vecteurs propres) et $D = \text{diag}(\lambda_1, \lambda_2, \lambda_3)$ via `numpy.linalg.eigh`.
- (c) Écrivez un code Python qui vérifie que $Q^\top S Q = D$ et $Q Q^\top = I_3$ (tolérance 10^{-10}).

Conseils.

- i. Vérifiez d'abord que S est bien *carrée*. Sinon, retournez **False**.
- ii. Parcourez *seulement* les indices $i < j$ (triangulaire supérieure) pour éviter les doublons.
- iii. Comparez `S[i, j]` et `S[j, i]` :
 - en exact (`tol=0.0`) pour des matrices entières/rationnelles ;
 - avec une tolérance (`tol>0`) pour des flottants (erreurs d'arrondi).
- iv. Si une seule paire viole la condition, **return False** ; sinon, **return True**.

Q2. Application à un jeu de données.

Recopiez le tableau ci-dessous dans votre Notebook comme un `np.array` de taille 20×3 :

$$X = [X_1 \ X_2 \ X_3] = \begin{pmatrix} 3.79 & 1.68 & 2.82 \\ 2.69 & 0.19 & 8.03 \\ 4.09 & 2.82 & 5.74 \\ 5.78 & 4.79 & 4.42 \\ 4.20 & 5.07 & 5.74 \\ 5.79 & 4.00 & 8.98 \\ 2.80 & 3.15 & 6.63 \\ 5.48 & 5.31 & 7.51 \\ 5.05 & 2.83 & 4.00 \\ 3.68 & 3.93 & 5.20 \\ 0.63 & 2.44 & 7.47 \\ 5.71 & 4.38 & 3.14 \\ 4.51 & 2.04 & 7.54 \\ 3.19 & 4.40 & 5.24 \\ 5.45 & 3.77 & 5.75 \\ 3.93 & 4.65 & 6.22 \\ 3.14 & 3.35 & 5.33 \\ 3.23 & 0.99 & 5.59 \\ 5.12 & 2.39 & 6.07 \\ 1.74 & 1.81 & 2.58 \end{pmatrix}.$$

(i) **Formule à utiliser.** Si $X \in \mathbb{R}^{n \times d}$ (ici $n = 20$, $d = 3$), la moyenne empirique est

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^n X_{i.} \in \mathbb{R}^d,$$

la matrice centrée $\tilde{X} = X - \mathbf{1}_n \hat{\mu}^\top$, et la **covariance empirique** est

$$\hat{\Sigma} = \frac{1}{n-1} \tilde{X}^\top \tilde{X} \in \mathbb{R}^{3 \times 3}.$$

(ii) **À rendre.** Calculez numériquement $\hat{\mu}$, affichez $\hat{\Sigma}$ *arrondie* à 10^{-2} (par ex. via `np.round`), et commentez en 3–4 lignes la proximité éventuelle entre $\hat{\Sigma}$ et la matrice S de l'énoncé (effet de l'échantillon fini).

(BONUS.) Soit une variable expliquée Y par $X = [X_1 \ X_2 \ X_3]$, dont les valeurs observées sont

$$Y = \begin{pmatrix} 2.674 \\ 4.610 \\ 4.536 \\ 4.808 \\ 5.230 \\ 6.847 \\ 4.825 \\ 6.446 \\ 3.858 \\ 4.517 \\ 4.592 \\ 4.025 \\ 5.285 \\ 4.572 \\ 5.091 \\ 5.289 \\ 4.295 \\ 3.739 \\ 4.773 \\ 2.177 \end{pmatrix}$$

On considère, pour chaque observation $i = 1, \dots, 20$, le modèle de régression linéaire multiple suivant :

$$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \beta_3 X_{i3} + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2),$$

où ε_i représente un petit bruit gaussien centré. Écrire en Python une fonction `least_square_estimation` qui calcule, par un **raisonnement matriciel** (et non avec une fonction toute faite), l'estimateur des moindres carrés ordinaires (MCO, ou *Ordinary Least Squares*). En particulier, votre fonction doit retourner l'estimation

$$(\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2, \hat{\beta}_3),$$

selon la formule vue en cours qui fait intervenir une matrice des données complétée par une colonne de 1 pour représenter l'ordonnée à l'origine β_0 .

Q3. Décorrélation et variances expliquées (1.5 pt).

- (a) **Projection dans la base propre.** À partir de votre matrice Q obtenue en 1.(b) (décomposition spectrale de S), centrez d'abord les données $X : X_c = X - \mathbf{1}_n \hat{\mu}^\top$. Projetez ensuite X_c dans la base propre :

$$Z = X_c Q.$$

Calculez les *variances empiriques* des colonnes de Z (formule d'échantillon : $\text{Var}(Z_{.j}) = \frac{1}{n-1} \sum_{i=1}^n (Z_{ij} - \bar{Z}_{.j})^2$) et comparez-les aux éléments de la diagonale de D .

- (b) **Conclusion en quelques lignes.** Expliquez pourquoi les colonnes de Q constituent des *directions orthogonales de variations indépendantes* (décorrélation par diagonalisation de S), et interprétez "grande" vs "petite" valeur propre comme *variance expliquée* élevée/faible par la composante correspondante. Vous pouvez, en complément, répéter la procédure avec Q_{emp} issu de la décomposition spectrale de $\hat{\Sigma}$ et commenter l'écart numérique.

EXERCICE 3 : Modèle dynamique

Contexte. Soient u_t, p_t, r_t les **proportions** d'habitants résidant respectivement en zone **urbaine**, **périurbaine** et **rurale** à l'année t (on suppose $u_t + p_t + r_t = 1$). D'une année à l'autre, les transitions observées sont les suivantes :

- depuis l'urbain : 80% restent en urbain, 20% passent en périurbain, 0% vont en rural ;
- depuis le périurbain : 10% vont en urbain, 70% restent en périurbain, 20% vont en rural ;
- depuis le rural : 0% vont en urbain, 20% vont en périurbain, 80% restent en rural.

En notant A la matrice de passage, on obtient une transition à l'année suivante via

$$x_{t+1} = A x_t.$$

Objectif. Reprendre le contexte de mobilité urbaine/périurbaine/rurale. Construire A en Python, vérifier ses propriétés stochastiques, simuler la dynamique, tenter une diagonalisation, inverser la matrice de passage par Gauss et comparer les puissances A^n .

Travail demandé.

- Q1. Construction de A .** À partir des pourcentages de transition (théorie), coder la matrice A et vérifier - via une méthode Python `is_stochastic` - qu'elle est *stochastique par colonnes* (ou par lignes selon votre convention — soyez cohérent avec l'énoncé) : sommes des colonnes (ou lignes) égale à 1, et toute entrée ≥ 0 .
- Q2. Simulation.** Écrire `simulate_markov(A, x0, T)` qui renvoie la trajectoire $(x_t)_{t=0, \dots, T}$. Tester deux x_0 (dont $(1, 0, 0)^\top$), et tracer l'évolution des composantes (employer, par exemple, `matplotlib.pyplot`).
- Q3. Diagonalisation (si possible).** Tenter `diag_attempt(A)` qui renvoie (P, D) si A est diagonalisable (test via `rg(P)`). Vérifier `np.allclose(A, P @ D @ np.linalg.inv(P))`. Comparer avec `numpy.linalg.eig`.
- Q4. Puissances & limite.** Comparer A^n calculé par (i) `numpy.linalg.matrix_power` et (ii) $P D^n P^{-1}$. Estimer numériquement $x_\infty = \lim x_t$ et commenter (stationnarité, rôle de la valeur propre $\lambda = 1$, toute autre remarque...).
- Q5. Inversion par Gauss.** Employer le package de votre choix pour calculer P^{-1} et vérifier si le résultat retourné coïncide bien avec la matrice calculée en théorie.