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 `numpy`	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 autre source académique.

I) 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.

II) Implémentation Python (20 pts)

Le livrable attendu est (au moins) un Notebook

`projet_algebre_Nom1Prenom1_Nom2Prenom2.ipynb`

exécutable sans erreur, présentant vos résultats, figures et interprétations. Il faudra commenter le code (non pas forcément toutes les lignes mais bien tous les blocs principaux) pour me montrer que vous comprenez ce que vous implémentez.

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. On appliquera nos méthodes au système suivant :

$$(\mathcal{S}) : \begin{cases} x_1 + x_2 + x_3 + x_4 = 1 \\ 2x_1 + 5x_2 + x_3 + x_4 = 2 \\ x_1 + x_2 + 4x_3 + 2x_4 = 3 \\ 3x_1 + x_2 + x_3 + 6x_4 = 4 \end{cases}$$

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

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

qui :

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

Sorties attendues : solution $\hat{\mathbf{x}}$ et, pour validation, le résidu $\|A\hat{\mathbf{x}} - \mathbf{b}\|_2$.
Appliquer cette méthode à $(\mathcal{S})$.

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

`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 $\|A\widehat{A^{-1}} - I\|_2$, $\|\widehat{A^{-1}}A - I\|_2$.
Appliquer cette méthode à $(\mathcal{S})$.

***(BONUS.)* Validation contre `NumPy`.** Comparer vos résultats aux fonctions usuelles :

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

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

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

Conseils : fixer une `seed`, afficher les métriques dans un tableau lisible, commenter 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).
- Veiller à toujours **échanger** b (ou le bloc droit) quand on échange des lignes de A.

Squelettes de fonctions proposés.

```python
def gauss_solve(A: np.ndarray, b: np.ndarray, tol: float = 1e-12) -> np.ndarray:
    """
    Parameters:
    - Solve Ax=b via Gaussian elimination (partial pivoting).
    - A: np.ndarray Square matrix (n, n), real-valued.
    - b: np.ndarray Right-hand side (n,) or (n,1).
    - tol: float pivot threshold to detect (near-)singularity.

    Returns:
    - x: np.ndarray Solution vector of shape (n,); raises ValueError if
       A is (near-)singular.
    """

    A = np.array(A, dtype=float, copy=True)
    b = np.array(b, dtype=float, copy=True).reshape(-1)
    n = A.shape[0]
    # Forward elimination with partial pivoting
    for k in range(n):
        # Choose pivot row if needed
        ...
        # Swap rows
        if pivot != k:
            ...
        # Eliminate below
        for i in range(k + 1, n):
            ...
    # Back substitution
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        ...
    return x
```

```python
def inverse_via_gauss(A: np.ndarray, tol: float = 1e-12) -> np.ndarray:
    """
    Compute A^{-1} via Gauss-Jordan on (A|I).
    A: (n,n) real; tol: pivot threshold.
    Returns Ainv (n,n); raises ValueError if (near-)singular.
    """
    A = np.array(A, dtype=float, copy=True)
    n = A.shape[0]
    M = np.hstack([A, np.eye(n)])  # (A | I)
    # Gauss-Jordan
    for k in range(n):
        ...
    return ...
```

Conseils et détails pour encoder l'algorithme du pivot de Gauss *pas à pas*.

A. Résolution $x = \texttt{gauss\_solve}(A, b)$ (pivot partiel).

1. **Préparation.** Copier A en flottants ; forcer b en vecteur colonne $(n,)$.
2. **Boucle d'élimination** pour $k = 0, \ldots, n-1$:
   (a) **Choix du pivot.** Trouver l'indice de ligne
   $$\text{pivot} \;=\; \arg\max_{i \in \{k,\ldots,n-1\}} |A_{i,k}|.$$
   Vérifier que $|A_{\text{pivot},k}| > \texttt{tol}$; sinon *lever une* `ValueError` (matrice singulière ou quasi singulière).
   (b) **Échange de lignes.** Si pivot $\neq k$, échanger les lignes k et pivot dans A et dans b.
   (c) **Élimination sous le pivot.** Pour chaque $i = k+1, \ldots, n-1$:
   $$m \leftarrow \frac{A_{i,k}}{A_{k,k}}, \qquad A_{i,k:} \leftarrow A_{i,k:} - m\, A_{k,k:}, \qquad b_i \leftarrow b_i - m\, b_k.$$
   Remarque. Si l'on obtient à une étape k un pivot nul (*i.e.*, $A_{k,k} = 0$), alors c'est que la matrice n'est pas inversible.
3. **Remontée (substitution arrière).** Initialiser $x \in \mathbb{R}^n$ à 0.
   $$\text{Pour } i = n-1, \ldots, 0 : \quad s \leftarrow \sum_{j=i+1}^{n-1} A_{i,j}\, x_j, \quad \text{vérifier } |A_{i,i}| > \texttt{tol}, \quad x_i \leftarrow \frac{b_i - s}{A_{i,i}}.$$
4. **Retour.** Renvoyer x.

B. Inversion $A^{-1} = \texttt{inverse\_via\_gauss}(A)$ (Gauss–Jordan).

1. **Préparation.** Former la matrice augmentée $(A \,|\, I_n)$ en flottants.
2. **Boucle Gauss–Jordan** pour $k = 0, \ldots, n-1$:
   (a) **Choix du pivot.** pivot $= \arg\max_{i \geq k} |A_{i,k}|$. Tester $|A_{\text{pivot},k}| > \texttt{tol}$, 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 $i \neq k$:
   $$\text{fact} \leftarrow A_{i,k}, \qquad \text{ligne } i \leftarrow \text{ligne } i - \text{fact} \times \text{ligne } k.$$
3. **Retour.** Quand la partie gauche est réduite à I_n, la partie droite est A^{-1} : extraire et renvoyer le bloc droit.

EXERCICE 2 : Théorème spectrale, matrice de covariance

Contexte. On modélise un problème d'**analyse de données** : trois capteurs mesurent une même grandeur physique (par exemple la température dans une pièce). La *matrice de covariance* S ci-dessous décrit les interdépendances entre les mesures.

Rappel. Le théorème spectrale établit qu'une matrice **symétrique** et à coefficients **réels** $S \in \mathscr{S}_n(\mathbb{R})$ est nécessairement diagonalisable. De surcroît, une matrice de passage que l'on peut proposer $Q = [\mathbf{q}_1 \mid \cdots \mid \mathbf{q}_n]$ peut dans ce cas être orthogonale, c'est-à-dire :

$$\begin{aligned} & QQ^\top = Q^\top Q = I_d \\ \iff & Q^{-1} = Q^\top \\ \iff & \text{Les vecteurs colonnes de } Q \text{ sont orthogonaux entre eux et de normes 1.} \\ \iff & \mathbf{q}_i \cdot \mathbf{q}_j = \begin{cases} 1, & \text{si } i = j \\ 0, & \text{si } i \neq j \end{cases} \end{aligned}$$

donnant ainsi, en notant D la matrice diagonale constituée des valeurs propres :

$$S = QDQ^\top.$$

Il est important de relever que cela a de très bonnes propriétés, tant sur le plan théorique que computationnel :

1. *D'une part, cela assure l'existence obligatoire d'au moins une base dans laquelle S est diagonale ;*
2. *D'autre part, la matrice de passage est constituée de vecteurs orthonormés entre-eux. Il n'est pas nécessairement de calculer l'inverse d'une matrice, car il suffit simplement de prendre sa transposée.*

Objectif. Vérifier numériquement le théorème spectral pour une matrice de covariance S, construire la décomposition $S = Q\,D\,Q^\top$ avec Q orthogonale et $D = \mathrm{diag}(\lambda_1, \lambda_2, \lambda_3)$, 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) Implémenter `is_symmetric(S,tol)` et vérifier $S^\top = S$.

(b) Calculer une base orthonormée de vecteurs propres de S et former Q (colonnes = vecteurs propres) et $D = \mathrm{diag}(\lambda_1, \lambda_2, \lambda_3)$ via `numpy.linalg.eigh`.

(c) Écriver un code Python qui vérifie que $Q^\top S Q = D$ et $QQ^\top = I_3$ (tolérance 10^{-10}).

Conseils.

i. Vérifier d'abord que S est bien *carrée*. Sinon, retourner `False`.

ii. Parcourer *seulement* les indices $i < j$ (triangulaire supérieure) pour éviter les doublons *en justifiant dans votre rédaction que cela suffit car S est symétrique*.

iii. Comparer `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 *(20 observations de 3 variables).* Recopier le tableau ci-dessous dans votre Notebook comme un `np.array` de taille 20×3 :

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

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

$$\widehat{\mu} \;=\; \frac{1}{n} \sum_{i=1}^{n} X_{i\cdot} \in \mathbb{R}^d,$$

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

$$\widehat{\Sigma} \;=\; \frac{1}{n-1} \widetilde{X}^\top \widetilde{X} \;\in \mathbb{R}^{3 \times 3}.$$

(ii) **Énoncé de la question Q2.** Calculer numériquement $\widehat{\mu}$, afficher $\widehat{\Sigma}$ *arrondie à* 10^{-2} (par ex. via `np.round`), et commenter en 3–4 lignes la proximité éventuelle entre $\widehat{\Sigma}$ et la matrice S de l'énoncé (effet de l'échantillon fini).

(BONUS.) **Estimation des paramètres d'un modèle linéaire usuel.** Soit une variable expliquée Y par $X = [X_1 \mid X_2 \mid X_3]$, 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 $i = 1, \ldots, 20$, le modèle de régression linéaire multiple suivant :

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

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

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

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.

Remarque. Il vous sera enseignés plus tard dans votre cursus les conditions à vérifier et pourquoi dans ce contexte proposer un modèle linéaire fait sens avec étude de la qualité du modèle.

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), centrer d'abord les données X : $X_c = X - \mathbf{1}_n \widehat{\mu}^\top$. Projeter ensuite X_c dans la base propre :

$$Z \;=\; X_c\, Q.$$

Calculer les *variances empiriques* des colonnes de Z via la formule :

$$\mathrm{Var}(Z_{\cdot j}) = \frac{1}{n-1} \sum_{i=1}^{n} (Z_{ij} - \overline{Z}_{\cdot j})^2$$

et comparer-les aux éléments de la diagonale de D. Afficher les résultats demandés via un `print`.

(b) **Conclusion en quelques lignes.** Expliquer pourquoi les colonnes de Q constituent des *directions orthogonales de variations indépendantes* (décorrélation par diagonalisation de S), et interpréter ce que peuvent signifier "grande" vs "petite" valeur propre.

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, 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). C'est-à-dire :

- toutes les entrées sont positives ou nulles,
- la somme des coefficients de chaque colonne (ou ligne) vaut 1.

Q2. Simulation. Écrire `simulate_markov(A, x0, T)` qui renvoie la trajectoire $(x_t)_{t=0,\dots,T}$. Plus précisément, produire un `array` ou tout autre format de la trajectoire

$$(x_0,\ x_1 = Ax_0,\ x_2 = Ax_1,\ \dots,\ x_T = Ax_{T-1}).$$

Q3. Tester `simulate_markov` pour au moins deux x_0 (dont $(1,0,0)^\top$). *Attention ! x_0 est à composantes positives ou nulles, de somme 1.* Tracer pour chaque vecteur initial l'évolution des composantes (utiliser `matplotlib.pyplot`). Commenter les graphiques obtenus :

- observe-t-on une convergence vers un équilibre stationnaire ?
- si oui, cet équilibre dépend-il du vecteur initial x_0 ?

Q4. Diagonalisation. L'objectif est de vérifier si la matrice A peut être mise sous la forme $A = PDP^{-1}$ où :

- D est une matrice diagonale contenant les valeurs propres de A ;
- P est inversible si et seulement si les vecteurs propres qui la constituent sont linéairement indépendants (i.e. $\mathrm{rg}(P) = 3$).

Écrire une fonction `diag_attempt(A)` qui :

(a) **utilise un package numérique** (par ex. `numpy.linalg.eig` ou `scipy.linalg.eig`) pour *calculer les couples* (valeurs propres, vecteurs propres associés) ;

(b) construit P (colonnes = vecteurs propres) et D (diagonale = valeurs propres, dans le même ordre que les colonnes de P). Pour P et D, construire ses matrices à partir des vecteurs propres et valeurs propres obtenus via un package adapté ;

(c) teste si P est inversible (par exemple via l'emploi `np.linalg.matrix_rank(P)` ou `np.abs(np.linalg.det(P)) > tol`) ;

(d) vérifie numériquement que $A \approx P\,D\,P^{-1}$ *(conseil : employer `np.allclose`).*

Remarque : gérer l'éventuelle présence de valeurs/vecteurs propres complexes (selon A). Comparer votre résultat avec `numpy.linalg.eig`.

Q5. Puissances & limite. Calculer puis comparer A^n :

(i) obtenue par `numpy.linalg.matrix_power` ;

(ii) obtenue comme produit PD^nP^{-1}, où P et D ont été obtenus à la question précédente.

Combien faut-il d'itération (*i.e.*, pour quelle valeur de n) pour que l'écart entre ces deux calculs soit inférieur à 10^{-6} ?

Estimer ensuite numériquement

$$x_\infty = \lim_{t\to\infty} x_t$$

et commenter :

— l'existence ou non d'un état stationnaire,

— le rôle de la valeur propre $\lambda = 1$ dans cette convergence,

— l'interprétation en termes de proportion des habitants.