Linear Algebra Project

M1 MIASHS

Theory & Python Implementation

Expected deliverable for the Python project: one Jupyter Notebook.

Learning Objectives

- Mobilize the core concepts : linear systems, vector spaces, kernel, image, rank, change of basis, diagonalization, orthogonal matrices, spectral theorem.
- Solve an *applied problem* by modeling with a matrix, diagonalizing, and interpreting the results (matrix behavior, asymptotics, stationary states).
- Implement in Python : produce functions for algebraic analysis and linear solving, and compare numerically with standard packages (e.g., `numpy`, `scipy`).

Indicative grading scheme.

Part I — Theory	**20 pts**
1. Kernel, image, rank, rank–nullity theorem	6 pts
2. Orthogonal matrices & spectral theorem	4 pts
3. Dynamical system, diagonalization, change of basis, inversion (Gauss)	10 pts
Part II — Python Implementation	**20 pts**
1. Gauss, inversion, linear solving	6 pts
2. Orthogonality, spectral (symmetric), `numpy` comparison	4 pts
3. Dynamical system study (simulation) & analysis	8 pts

Rules. The Python implementation deliverable is **individual or pairs**. Justify your answers by citing the course or appropriate academic sources.

1 Theory (20 pts)

A 2-hour written exam is scheduled for the theory part. You will analyze selected matrices taken from the Python assignment, which is distributed before the written exam. This will :

- assess your theoretical and academic command of linear algebra fundamentals ;
- connect the Python-based practice back to the underlying theory.

II) Python Implementation (20 pts)

The expected deliverable is (at least) a Notebook

`project_algebra_Name1Surname1_Name2Surname2.ipynb`

that runs without errors, showing your results, figures, and interpretations. You must comment the code (not necessarily every line, but each main block) to demonstrate that you understand what you are implementing.

EXERCISE 1 : Gaussian Elimination *from scratch* : solving and inversion

Objective. Implement the Gaussian elimination algorithm *with partial pivoting* in order to

(i) solve $Ax = b$;

(ii) invert a matrix using Gauss–Jordan elimination,

then **numerically validate** your results against `numpy`.

Required work. We will apply our functions to the following linear system :

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

Q1. Linear system solving *from scratch.* Implement a function

$$\texttt{gauss\_solve(A, b)} \rightarrow \hat{\mathbf{x}}$$

that :

- — checks the compatibility of dimensions ($A \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$) ;
- — performs direct elimination with *partial pivoting* (upper triangularization) ;
- — performs back-substitution to obtain $\hat{\mathbf{x}}$;
- — raises a clear exception if A is (numerically) singular (*i.e.*, a zero pivot occurs at some stage).

Expected outputs : solution $\hat{\mathbf{x}}$ and, for validation, the residual $\|A\hat{\mathbf{x}} - \mathbf{b}\|_2$.

Apply this function to $(\mathcal{S})$.

Q2. Matrix inversion via Gauss–Jordan *from scratch.* Implement a function

$$\texttt{inverse\_via\_gauss(A)} \rightarrow \widehat{A^{-1}}$$

that :

- — checks that A is square ;
- — constructs the augmented matrix $(A \,|\, I)$;
- — applies *Gauss–Jordan with partial pivoting* until reaching $(I \,|\, A^{-1})$;
- — raises an exception if A is (numerically) non-invertible.

Expected outputs : $\widehat{A^{-1}}$ and control norms $\|A\widehat{A^{-1}} - I\|_2$, $\|\widehat{A^{-1}}A - I\|_2$.

Apply this function to $(\mathcal{S})$.

(BONUS.) **Validation against `NumPy`.** Compare your results with the standard functions :

- Generate several test matrices (random $n = 3, 4, 8$, ill-conditioned matrices such as Hilbert, matrices with row permutations).
- For each test, compare :

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

- Report : relative errors $\frac{\|\hat{\mathbf{x}} - \mathbf{x}_{np}\|_2}{\|\mathbf{x}_{np}\|_2}$, $\frac{\|\widehat{A^{-1}} - A_{np}^{-1}\|_2}{\|A_{np}^{-1}\|_2}$, residuals $\|A\hat{\mathbf{x}} - \mathbf{b}\|_2$, and products $A\widehat{A^{-1}}$ / $\widehat{A^{-1}}A$ close to I.

Tips : fix a `seed`, display the metrics in a readable table, comment on the cases where tolerance is exceeded (conditioning).

Constraints.

- **Forbidden** for the main computation : `numpy.linalg.solve`, `numpy.linalg.inv`. They should only be used to *verify* your outputs.
- Handling singular or nearly singular matrices : raise a clear `ValueError` if the pivot is smaller than a fixed tolerance.
- **Docstrings in English or French** and concise comments explaining each step (pivot, row swaps, elimination, back substitution).
- Always make sure to **swap** b (or the right-hand block) when swapping rows in A.

Proposed function skeletons.

```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 ...
```

Guidelines and details for encoding the Gaussian pivot algorithm *step by step.*

A. Solving $x = \texttt{gauss\_solve}(A, b)$ (partial pivoting).

1. **Preparation.** Copy A in floating point ; force b into a column vector of shape $(n,)$.
2. **Elimination loop** for $k = 0, \dots, n-1$:
   (a) **Pivot choice.** Find the row index
   $$\text{pivot} \;=\; \arg \max_{i \in \{k,\dots,n-1\}} |A_{i,k}|.$$
   Check that $|A_{\text{pivot},k}| > \texttt{tol}$; otherwise raise a `ValueError` (singular or nearly singular matrix).
   (b) **Row swap.** If pivot $\neq k$, swap rows k and pivot in both A and b.
   (c) **Elimination below the pivot.** For each $i = k+1, \dots, n-1$:
   $$m \leftarrow \frac{A_{i,k}}{A_{k,k}}, \qquad A_{i,k:} \leftarrow A_{i,k:} - m\, A_{k,k:}, \qquad b_i \leftarrow b_i - m\, b_k.$$
   Remark. If at some step k the pivot is zero (*i.e.*, $A_{k,k} = 0$), then the matrix is not invertible.
3. **Back substitution.** Initialize $x \in \mathbb{R}^n$ as 0.
   $$\text{For } i = n-1, \dots, 0 : \quad s \leftarrow \sum_{j=i+1}^{n-1} A_{i,j}\, x_j, \quad \text{check } |A_{i,i}| > \texttt{tol}, \quad x_i \leftarrow \frac{b_i - s}{A_{i,i}}.$$
4. **Return.** Output x.

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

1. **Preparation.** Form the augmented matrix $(A \,|\, I_n)$ in floating point.
2. **Gauss–Jordan loop** for $k = 0, \dots, n-1$:
   (a) **Pivot choice.** pivot $= \arg\max_{i \ge k} |A_{i,k}|$. Test $|A_{\text{pivot},k}| > \texttt{tol}$, otherwise raise `ValueError`.
   (b) **Row swap.** If necessary, swap rows k and pivot *in the entire augmented matrix.*
   (c) **Pivot row normalization.** Divide the entire row k by $A_{k,k}$ to obtain a 1 on the pivot.
   (d) **Annihilate other rows.** For all $i \neq k$:
   $$\text{fact} \leftarrow A_{i,k}, \qquad \text{row } i \leftarrow \text{row } i - \text{fact} \times \text{row } k.$$
3. **Return.** When the left block is reduced to I_n, the right block is A^{-1} : extract and return it.

EXERCISE 2 : Spectral theorem, covariance matrix

Context. We model a **data analysis** problem : three sensors measure the same physical quantity (e.g., the temperature in a room). The *covariance matrix* S below describes the interdependencies between the measurements.

Reminder. The spectral theorem states that a **symmetric** real matrix $S \in \mathcal{S}_n(\mathbb{R})$ is necessarily diagonalizable. Moreover, a change-of-basis matrix one may use $Q = [\mathbf{q}_1 \mid \cdots \mid \mathbf{q}_n]$ can be orthogonal, i.e. :

$$
\begin{aligned}
& QQ^\top = Q^\top Q = I_d \\
\iff & Q^{-1} = Q^\top \\
\iff & \text{The column vectors of } Q \text{ are mutually orthogonal and have norm 1.} \\
\iff & \mathbf{q}_i \cdot \mathbf{q}_j = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}
\end{aligned}
$$

hence, denoting by D the diagonal matrix of eigenvalues :

$$S = QDQ^\top.$$

It is important to note that this has very good properties, both theoretically and computationally :

1. *On the one hand, it guarantees the existence of at least one basis in which S is diagonal ;*
2. *On the other hand, the change-of-basis matrix consists of orthonormal vectors. It is not necessary to compute a matrix inverse, because one can simply take the transpose.*

Objective. Numerically verify the spectral theorem for a covariance matrix S, build the decomposition $S = Q\,D\,Q^\top$ with Q orthogonal and $D = \operatorname{diag}(\lambda_1, \lambda_2, \lambda_3)$, then *interpret* the principal components by simulating data with covariance close to S.

Matrix under study.

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

Required work.

Q1. Checks and spectral decomposition (1 pt).

(a) Implement `is_symmetric(S,tol)` and verify $S^\top = S$.

(b) Compute an orthonormal basis of eigenvectors of S and form Q (columns = eigenvectors) and $D = \operatorname{diag}(\lambda_1, \lambda_2, \lambda_3)$ via `numpy.linalg.eigh`.

(c) Write Python code that verifies $Q^\top S Q = D$ and $QQ^\top = I_3$ (tolerance 10^{-10}).

Tips.

i. First verify that S is indeed *square*. Otherwise, return `False`.

ii. Loop over *only* indices $i < j$ (upper triangular part) to avoid duplicates *and justify in your write-up that this suffices because S is symmetric*.

iii. Compare `S[i,j]` and `S[j,i]` :

— exactly (`tol=0.0`) for integer/rational matrices ;

— with a tolerance (`tol>0`) for floats (round-off errors).

iv. If a single pair violates the condition, `return False` ; otherwise, `return True`.

Q2. Application to a dataset *(20 observations of 3 variables).* Copy the table below into your Notebook as an 20×3 `np.array` :

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

(i) **Formula to use.** If $X \in \mathbb{R}^{n\times d}$ (here $n = 20$, $d = 3$), the empirical mean is

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

the centered matrix is $\widetilde{X} = X - \mathbf{1}_n\widehat{\mu}^\top$, and the **empirical covariance** is

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

(ii) **Statement of question Q2.** Compute $\widehat{\mu}$ numerically, display $\widehat{\Sigma}$ *rounded to* 10^{-2} (e.g., via `np.round`), and comment in 3–4 lines on the possible closeness between $\widehat{\Sigma}$ and the matrix S in the prompt (finite-sample effect).

(BONUS.) **Estimating the parameters of a standard linear model.** Let a response variable Y be explained by $X = [X_1 \mid X_2 \mid X_3]$, with observed values

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

We consider, for each observation $i = 1, \ldots, 20$, the following multiple linear regression model :

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

where ε_i represents a small centered Gaussian noise. Write in Python a function `least_square_estimation` that computes, using a **matrix-based derivation** (not a black-box function), the ordinary least squares (OLS) estimator. In particular, your function must return

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

according to the formula seen in class which uses a data matrix augmented with a column of ones to represent the intercept β_0.

Remark. You will later be taught the conditions to verify and why, in this context, proposing a linear model makes sense together with an assessment of model quality.

Q3. Decorrelation and explained variances (1.5 pt).

(a) **Projection in the eigenbasis.** Using your matrix Q obtained in 1.(b) (spectral decomposition of S), first center the data X : $X_c = X - \mathbf{1}_n \widehat{\mu}^\top$. Then project X_c onto the eigenbasis :

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

Compute the *empirical variances* of the columns of Z using the formula :

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

and compare them to the diagonal entries of D. Display the required results via a `print`.

(b) **Conclusion in a few lines.** Explain why the columns of Q constitute *orthogonal directions of independent variations* (decorrelation via diagonalization of S), and interpret what "large" vs "small" eigenvalue can mean.

EXERCISE 3 : Dynamic model

Context. Let u_t, p_t, r_t be the **proportions** of inhabitants living respectively in **urban**, **suburban**, and **rural** areas at year t (assume $u_t + p_t + r_t = 1$). From one year to the next, the observed transitions are :

— from urban : 80% remain urban, 20% move to suburban, 0% move to rural ;

— from suburban : 10% move to urban, 70% remain suburban, 20% move to rural ;

— from rural : 0% move to urban, 20% move to suburban, 80% remain rural.

Letting A denote the transition matrix, the next-year state is given by

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

Objective. Revisit the urban/suburban/rural mobility setup. Build A in Python, check its stochastic properties, simulate the dynamics, attempt a diagonalization, invert the transition matrix via Gauss, and compare the powers A^n.

Required work.

Q1. Building A. From the transition percentages, code the matrix A and verify—via a Python method `is_stochastic`—that it is *column-stochastic* (or row-stochastic depending on your convention : be consistent with the statement). That is :

- all entries are nonnegative,
- the sum of the coefficients in each column (or row) equals 1.

Q2. Simulation. Write `simulate_markov(A, x0, T)` which returns the trajectory $(x_t)_{t=0,\dots,T}$. More precisely, produce an array (or any suitable format) of the trajectory

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

Q3. Test `simulate_markov` for at least two x_0 (including $(1,0,0)^\top$). *Warning! x_0 must have nonnegative components summing to 1.* Plot, for each initial vector, the evolution of the components (use `matplotlib.pyplot`). Comment on the plots :

- do you observe convergence to a stationary equilibrium ?
- if so, does this equilibrium depend on the initial vector x_0 ?

Q4. Diagonalization. The goal is to check whether the matrix A can be written as $A = PDP^{-1}$ where :

- D is a diagonal matrix containing the eigenvalues of A ;
- P is invertible if and only if its eigenvectors are linearly independent (i.e., $\text{rank}(P) = 3$).

Write a function `diag_attempt(A)` that :

(a) **uses a numerical package** (e.g., `numpy.linalg.eig` or `scipy.linalg.eig`) to *compute the pairs* (eigenvalues, associated eigenvectors) ;

(b) builds P (columns = eigenvectors) and D (diagonal = eigenvalues, in the same order as the columns of P). For P and D, construct these matrices from the eigenvectors and eigenvalues obtained via an appropriate package ;

(c) tests whether P is invertible (for example using `np.linalg.matrix_rank(P)` or `np.abs(np.linalg.det(P))` $>$ `tol`) ;

(d) numerically verifies that $A \approx P\,D\,P^{-1}$ *(tip : use `np.allclose`).*

Remark : handle the possible presence of complex eigenvalues/eigenvectors (depending on A). Compare your result with `numpy.linalg.eig`.

Q5. Powers & limit. Compute and compare A^n :

(i) obtained via `numpy.linalg.matrix_power` ;

(ii) obtained as the product PD^nP^{-1}, where P and D were obtained in the previous question.

How many iterations (*i.e.*, for which value of n) are needed so that the difference between these two computations is smaller than 10^{-6} ?

Next, numerically estimate

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

and comment on :

— the existence (or not) of a stationary state,

— the role of the eigenvalue $\lambda = 1$ in this convergence,

— the interpretation in terms of the proportions of inhabitants.