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	4 pts
2. Orthogonal matrices & spectral theorem	5 pts
3. Dynamical system, diagonalization, change of basis, inversion (Gauss)	11 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.

2 Python Implementation (20 pts)

The expected deliverable is (at least) one Jupyter Notebook `projet_algebre_LastNameFirstName.ipynb`, *runnable without errors*, presenting your results, any figures, and your comments.

EXERCISE 1 : Gaussian Elimination *from scratch* — Solving and Inversion

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

(i) solve $Ax = b$;

(ii) invert a matrix via Gauss–Jordan,

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

Tasks.

Q1. Linear solve *from scratch*. Implement a function

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

which :

— checks dimension compatibility ($A \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$) ;

— performs forward elimination with *partial pivoting* (upper triangularization) ;

— carries out *back substitution* to obtain $\hat{\mathbf{x}}$;

— raises a clear exception if A is (numerically) singular.

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

Q2. Gauss–Jordan inverse *from scratch*. Implement a function

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

which :

— checks that A is square ;

— builds the augmented matrix $(A \,|\, I)$;

— applies *Gauss–Jordan with partial pivoting* until obtaining $(I \,|\, A^{-1})$;

— raises an exception if A is (numerically) non-invertible.

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

(BONUS.) **Validation against `NumPy`.** Compare your results to standard routines :

— Generate several test matrices (random with $n = 3, 4, 8$; ill-conditioned such as Hilbert ; row-permuted matrices).

— 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 $\dfrac{\|\hat{\mathbf{x}} - \mathbf{x}_{\text{np}}\|_2}{\|\mathbf{x}_{\text{np}}\|_2}$, $\dfrac{\|\widehat{A^{-1}} - A_{\text{np}}^{-1}\|_2}{\|A_{\text{np}}^{-1}\|_2}$, the residual $\|A\hat{\mathbf{x}} - \mathbf{b}\|_2$, and the products $A\widehat{A^{-1}}$ / $\widehat{A^{-1}}A$ being close to I.

Tips : set a random `seed`, display metrics in a clear table, and comment on cases where the tolerance is exceeded (conditioning).

Constraints.

- **Forbidden** in the core algorithm : `numpy.linalg.solve`, `numpy.linalg.inv`. They may be used *only* to *verify* your outputs.
- Handle singular or near-singular matrices : raise a clear `ValueError` when the pivot falls below a fixed tolerance.
- **Docstrings in English or French** and concise comments explaining each step (pivot choice, row swaps, elimination, back substitution).

Function templates.

```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

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 ...
```

Writing guidelines — *step-by-step* algorithm

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

1. **Setup.** Copy A as floating-point ; coerce b to a column-shaped vector $(n,)$.
2. **Elimination loop** for $k = 0, \ldots, n-1$:
   (a) **Pivot choice.** Find the row index
   $$\text{pivot} \;=\; \arg\max_{i \in \{k,\ldots,n-1\}} |A_{i,k}|.$$
   Check that $|A_{\text{pivot},k}| > \texttt{tol}$; otherwise *raise* a `ValueError` (matrix singular or nearly singular).
   (b) **Row swap.** If pivot $\neq k$, swap rows k and pivot in A *and* in b.
   (c) **Eliminate below the pivot.** For each $i = k+1, \ldots, n-1$:
   $$m \leftarrow \frac{A_{i,k}}{A_{k,k}}, \qquad A_{i,k:} \leftarrow A_{i,k:} - m\, A_{k,k:}, \qquad b_i \leftarrow b_i - m\, b_k.$$
3. **Back substitution.** Initialize $x \in \mathbb{R}^n$ with zeros.
   $$\text{For } i = n-1, \ldots, 0: \quad s \leftarrow \sum_{j=i+1}^{n-1} A_{i,j}\, x_j, \quad \text{check } |A_{i,i}| > \texttt{tol}, \quad x_i \leftarrow \frac{b_i - s}{A_{i,i}}.$$
4. **Return.** Output x.

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

1. **Setup.** Form the augmented matrix $(A \,|\, I_n)$ in floating-point.
2. **Gauss–Jordan loop** for $k = 0, \ldots, n-1$:
   (a) **Pivot choice.** pivot $= \arg\max_{i \geq k} |A_{i,k}|$. Check $|A_{\text{pivot},k}| > \texttt{tol}$, else raise `ValueError`.
   (b) **Row swap.** If needed, swap rows k and pivot *across the full augmented matrix.*
   (c) **Normalize pivot row.** Divide the *entire* row k by $A_{k,k}$ so the pivot becomes 1.
   (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.** Once the left block is I_n, the right block is A^{-1} : extract and return the right block.

Remarks.

— **Partial pivoting** (max of $|A_{i,k}|$) improves numerical stability ; `tol` is used to *detect* tiny pivots.

— Always **swap** b (or the right block) when swapping rows of A.

— Document each step (docstrings in English) and **test** : compare against `numpy.linalg.solve` / `inv` (for validation only).

EXERCISE 2 : Application to a Covariance Matrix

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

Matrix under study.

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

Tasks.

Q1. Checks and spectral decomposition (1 pt).

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

(b) Compute an orthonormal eigenbasis of S and form Q (columns = eigenvectors) and $D = \mathrm{diag}(\lambda_1, \lambda_2, \lambda_3)$ 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 check that S is indeed *square*. Otherwise, return `False`.

ii. Iterate over indices $i < j$ only (upper triangular part) to avoid duplicates.

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

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

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

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

Q2. Application to a dataset. Copy the table below into your Notebook as an `np.array` of size 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) **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 **sample covariance** is

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

(ii) **To submit.** Compute $\widehat{\mu}$ numerically, display $\widehat{\Sigma}$ *rounded to* 10^{-2} (e.g., with `np.round`), and comment in 3–4 lines on the potential closeness between $\widehat{\Sigma}$ and the matrix S from the statement (finite-sample effect).

(BONUS.) Let Y be a response variable explained by $X = [X_1\ X_2\ X_3]$, whose observed values are given by

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

We consider, for each observation $i = 1, \ldots, 20$, the 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 denotes a small mean-zero Gaussian noise term.
Write in Python a function `least_square_estimation` that computes—by a **matrix-based derivation** (and not via a black-box routine)—the Ordinary Least Squares (OLS) estimator. In particular, your function must return the estimate

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

using the course formula that involves the data matrix augmented with a column of ones to account for the intercept β_0.

Q3. Decorrelation and explained variances (1.5 pt).

(a) **Projection onto the eigenbasis.** Using your matrix Q from 1.(b) (spectral decomposition of S), first center the data :

$$X_c \;=\; X - \mathbf{1}_n \widehat{\mu}^\top.$$

Then project onto the eigenbasis :

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

Compute the *sample variances* of the columns of Z (sample 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.

(b) **Conclusion (a few lines).** Explain why the columns of Q provide *orthogonal directions of independent variation* (decorrelation via diagonalization of S), and interpret "large" vs "small" eigenvalues as, respectively, *high* vs *low* explained variance for the corresponding component. Optionally, repeat the procedure with Q_{emp} obtained from the spectral decomposition of $\widehat{\Sigma}$ and comment on the numerical differences.

EXERCISE 3 : Dynamic Model

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

— from urban : 80% stay urban, 20% move to peri-urban, 0% move to rural ;
— from peri-urban : 10% move to urban, 70% stay peri-urban, 20% move to rural ;
— from rural : 0% move to urban, 20% move to peri-urban, 80% stay rural.

Letting A be the transition matrix, the state evolves as

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

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

Tasks.

Q1. Building A. Using the transition percentages (theoretical part), code the matrix A and verify—via a Python routine `is_stochastic`—that it is *column-stochastic* (or row-stochastic, depending on your convention—be consistent with the statement) : column (or row) sums equal to 1, and all entries ≥ 0.

Q2. Simulation. Write `simulate_markov(A, x0, T)` returning the trajectory $(x_t)_{t=0,\dots,T}$. Test two initial states x_0 (including $(1,0,0)^\top$), and plot the component evolution (by using, for instance, the `matplotlib.pyplot` package).

Q3. Diagonalization (if possible). Implement `diag_attempt(A)` that returns (P, D) if A is diagonalizable (check via rank(P)). Verify `np.allclose(A, P @ D @ np.linalg.inv(P))`. Compare with `numpy.linalg.eig`.

Q4. Powers & limit. Compare A^n computed by (i) `numpy.linalg.matrix_power` and (ii) PD^nP^{-1}. Numerically estimate $x_\infty = \lim x_t$ and comment (stationarity, the role of the eigenvalue $\lambda = 1$, and any other relevant observations).

Q5. Gauss-based inversion. Use the package of your choice to compute P^{-1} and check that the returned matrix coincides with the theoretically computed inverse.