Homologie Persistante et Théorème de Stabilité

Feuille de Route de Recherche

De la topologie algébrique à l'analyse topologique
des données et à l'implémentation computationnelle

Document rédigé à partir des travaux de *Edelsbrunner, Letscher & Zomorodian* (2000–2002), *Zomorodian & Carlsson* (2005)
et *Cohen-Steiner, Edelsbrunner & Harer* (2007).

Résumé du projet

Ce document propose une feuille de route complète pour l'étude de l'**homologie persistante** en topologie algébrique computationnelle. En partant des fondements de l'homologie simpliciale, on progresse vers la théorie des modules de persistance et leur décomposition en codes-barres, puis vers le résultat central : le **théorème de stabilité** (Cohen-Steiner, Edelsbrunner, Harer, 2007), qui garantit la robustesse de la méthode face au bruit. Le projet se conclut par un volet computationnel ambitieux : une implémentation Python complète de l'analyse topologique des données, incluant le calcul de diagrammes de persistance et leur exploitation dans un pipeline de *machine learning*. La durée estimée est de 4 à 6 mois.

Livrable principal : Mémoire + implémentation Python reproductible

Contents

1 Introduction et motivation **3**
- 1.1 Le problème de la forme en analyse des données 3
- 1.2 La découverte de l'homologie persistante 3
- 1.3 Bref historique 3

2 Objectifs du projet **4**

3 Prérequis et ressources **5**
3.1 Prérequis mathématiques 5
3.2 Ressources recommandées 5

4 Feuille de route détaillée **7**
4.1 Vue d'ensemble 7
4.2 Phase 1 — Homologie simpliciale (semaines 1–4) 7
4.3 Phase 2 — Modules de persistance et décomposition (semaines 5–8) 8
4.4 Phase 3 — Stabilité et entrelacement (semaines 9–13) 9
4.5 Phase 4 — Projet de code : pipeline TDA complet (semaines 14–18) 11
4.6 Phase 4b — Projet avancé : TDA appliquée à la classification EEG (semaines 17–20) 17
4.7 Phase 5 — Rédaction et approfondissements (semaines 19–24) 35

5 Tableau récapitulatif **37**

6 Conclusion **37**

1 Introduction et motivation

1.1 Le problème de la forme en analyse des données

En analyse des données modernes, on dispose typiquement d'un nuage de points $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$, issu d'un processus inconnu, éventuellement bruité. Une question naturelle et fondamentale est la suivante :

Quelles sont les propriétés "de forme" globales de X, et comment les extraire de manière robuste au bruit ?

Les méthodes classiques d'analyse des données — statistiques descriptives, ACP, régressions — sont essentiellement linéaires et métriques. Elles ne capturent pas des invariants *topologiques* comme le nombre de composantes connexes, la présence de boucles ou de cavités. Or, pour de nombreuses applications (biologie moléculaire, neurosciences, analyse d'images, traitement du signal), ces invariants globaux sont porteurs d'une information qualitative irremplaçable.

L'approche naïve — calculer directement les groupes d'homologie du nuage X — échoue immédiatement : X est un ensemble discret, de topologie triviale. Il faut donc **épaissir** les points à une échelle $\varepsilon > 0$, puis **faire varier** ε. C'est précisément l'idée de la filtration, et le point de départ de l'**homologie persistante**.

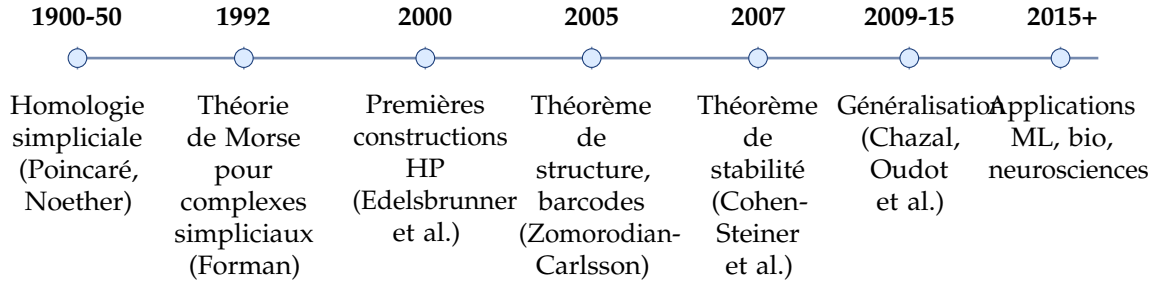
1.2 La découverte de l'homologie persistante

Fait fondateur

L'homologie persistante, introduite par Edelsbrunner, Letscher et Zomorodian (2000–2002) et formalisée par Zomorodian et Carlsson (2005), suit l'évolution des groupes d'homologie d'une filtration de complexes simpliciaux et représente naissances et morts des classes topologiques sous la forme d'un **code-barres** (barcode) ou d'un **diagramme de persistance**.

Le résultat qui confère à cette théorie toute sa solidité est le **théorème de stabilité** (Cohen-Steiner, Edelsbrunner, Harer, 2007) : deux fonctions proches donnent des diagrammes de persistance proches, en distance dite *bottleneck*. Ce théorème est l'analogue topologique d'un résultat de continuité, et c'est lui qui garantit que la méthode résiste au bruit d'acquisition ou d'échantillonnage.

1.3 Bref historique



2 Objectifs du projet

Ce projet poursuit quatre grands objectifs, progressifs et articulés.

Objectif 1 — Maîtrise de l'homologie simpliciale

Comprendre et démontrer rigoureusement les fondements de l'homologie simpliciale : complexes de Čech et de Vietoris-Rips, opérateur bord, groupes d'homologie, nombres de Betti. Calculer explicitement ces invariants pour les surfaces classiques (tore, sphère, bouteille de Klein, $\mathbb{RP}^2$).

Objectif 2 — Théorie des modules de persistance

Définir rigoureusement les filtrations et les modules de persistance. Démontrer le **théorème de décomposition** (Zomorodian-Carlsson, 2005) : tout module de persistance à type de pointage fini se décompose de manière unique en somme directe de modules intervalles $\mathbb{F}[b_j, d_j)$. Maîtriser l'algorithme de réduction de la matrice des bords pour le calcul effectif du barcode.

Objectif 3 — Théorème de stabilité et entrelacement

Définir les distances de bottleneck et de Wasserstein entre diagrammes de persistance. Démontrer le **lemme d'entrelacement** (Chazal et al., 2009) et le **théorème de stabilité** complet (Cohen-Steiner, Edelsbrunner, Harer, 2007). Étendre au cadre des espaces métriques compacts via la distance de Gromov-Hausdorff.

Objectif 4 — Implémentation et applications (projet code)

Implémenter en Python un pipeline complet d'analyse topologique des données : calcul de filtrations de Vietoris-Rips via les bibliothèques `gudhi` et `ripser`, visualisation des barcodes et diagrammes de persistance, plongement dans des espaces vectoriels (images de persistance), et intégration dans un pipeline de *machine learning* sur données réelles.

3 Prérequis et ressources

3.1 Prérequis mathématiques

Prérequis

Les prérequis suivants sont indispensables avant d'aborder la Phase 1. Chaque point renvoie à la section 3.2 pour les ressources correspondantes.

- **Algèbre linéaire** : espaces vectoriels, applications linéaires, noyau et image, dualité. [*Ressource A*]
- **Topologie générale** : espaces topologiques, compacité, connexité, espaces métriques, continuité. [*Ressource A*]
- **Topologie algébrique élémentaire** : groupe fondamental π_1, revêtements, notions de CW-complexes. Niveau M1 minimal. [*Ressource B*]
- **Algèbre homologique de base** : suites exactes, suites exactes longues, lemme du serpent. [*Ressource B*]
- **Théorie des catégories (notions)** : catégories, foncteurs, transformations naturelles. Niveau initiation suffisant. [*Ressource C*]
- **Géométrie différentielle (optionnel)** : variétés, théorème de Morse — utile pour les énoncés généraux du théorème de stabilité. [*Ressource D*]
- **Python scientifique** : `numpy`, `scipy`, `matplotlib`, `scikit-learn`. Niveau intermédiaire requis pour la Phase 4. [*Ressource E*]

3.2 Ressources recommandées

Ressources : [A] Topologie générale et algèbre linéaire

- J. R. Munkres, *Topology*, Pearson, 2^{e} éd. (2000) — référence classique, chapitres 1–4.
- S. Lang, *Algebra*, Springer, 3^{e} éd. (2002) — pour les aspects algébriques (modules sur un anneau).

Ressources : [B] Topologie algébrique

- A. Hatcher, *Algebraic Topology*, Cambridge University Press (2002) — disponible gratuitement en ligne, chapitres 2–3. Référence principale pour la Phase 1.
- M. J. Greenberg & J. R. Harper, *Algebraic Topology: A First Course*, Benjamin (1981) — traitement concis et rigoureux.

Ressources : [C] Homologie persistante et TDA

- H. Edelsbrunner & J. Harer, *Computational Topology: An Introduction*, AMS (2010) — référence principale des Phases 2 et 3.
- F. Chazal, V. de Silva, M. Glisse & S. Oudot, *The Structure and Stability of Persistence Modules*, Springer (2016) — ouvrage de référence, accessible après la Phase 1.
- S. Oudot, *Persistence Theory: From Quiver Representations to Data Analysis*, AMS (2015) — pour les approfondissements.
- A. Zomorodian & G. Carlsson, *Computing Persistent Homology*, Discrete & Comput. Geom. (2005) — article fondateur, lecture obligatoire.
- D. Cohen-Steiner, H. Edelsbrunner & J. Harer, *Stability of Persistence Diagrams*, Discrete & Comput. Geom. (2007) — article central du projet, Phase 3.

Ressources : [D] Théorie de Morse et géométrie

- J. Milnor, *Morse Theory*, Princeton University Press (1963) — pour les aspects géométriques du théorème de stabilité.
- R. Forman, *Morse Theory for Cell Complexes*, Advances in Mathematics (1998) — version discrète, utilisée en Phase 1.

Ressources : [E] Python scientifique et TDA computationnelle

- Documentation officielle `gudhi` : `https://gudhi.inria.fr/python/latest/` — bibliothèque principale pour le calcul d'homologie persistante.
- C. Tralie *et al.*, *Ripser.py: A Lean Persistent Homology Library for Python*, JOSS (2018) — pour `ripser`.
- Documentation `scikit-tda` : `https://scikit-tda.org` — écosystème complet (ripser, persim, cechmate, tadasets).
- H. Adams *et al.*, *Persistence Images: A Stable Vector Representation of Persistent Homology*, JMLR (2017) — pour le plongement dans L^2, Phase 4.

4 Feuille de route détaillée

4.1 Vue d'ensemble

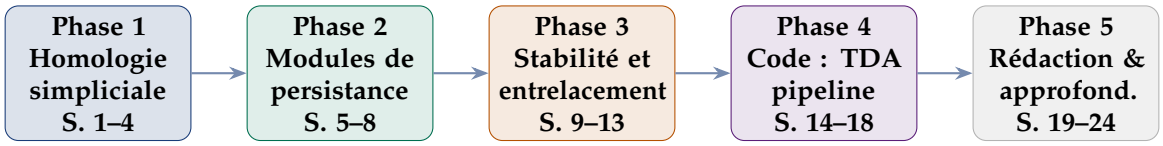


Figure 1 : Vue d'ensemble de la feuille de route (durée totale : 4 à 6 mois).

4.2 Phase 1 — Homologie simpliciale (semaines 1–4)

Phase 1 — Homologie simpliciale

Objectif de la phase : Maîtriser la définition rigoureuse et le calcul des groupes d'homologie simpliciale. Construire les complexes de Čech et de Vietoris-Rips à partir d'un nuage de points. Calculer explicitement les invariants topologiques des surfaces classiques.

Complexes simpliciaux. Un k-simplexe est l'enveloppe convexe de $k+1$ points affinement indépendants $v_0, \dots, v_k \in \mathbb{R}^d$, noté $\sigma = [v_0, \dots, v_k]$. Un *complexe simplicial* K est une collection finie de simplexes close par passage aux faces et telle que l'intersection de deux simplexes est une face commune.

Complexes de Čech et de Vietoris-Rips. Pour un nuage de points $X = \{x_1, \dots, x_n\} \subset \mathbb{R}^d$ et $\varepsilon > 0$:

$$\check{\mathrm{C}}\mathrm{ech}_\varepsilon(X) = \Big\{\sigma \subseteq X \;\Big|\; \bigcap_{x \in \sigma} B(x, \varepsilon) \neq \emptyset\Big\}, \qquad \mathrm{Rips}_\varepsilon(X) = \{\sigma \subseteq X \mid \mathrm{diam}(\sigma) \leq \varepsilon\}.$$

On dispose de l'entrelacement fondamental :

$$\check{\mathrm{C}}\mathrm{ech}_\varepsilon(X) \subseteq \mathrm{Rips}_\varepsilon(X) \subseteq \check{\mathrm{C}}\mathrm{ech}_{2\varepsilon}(X),$$

qui implique des bornes contrôlées sur les diagrammes de persistance associés. Le complexe de Vietoris-Rips est préféré en pratique ($O(n^2)$ distances pairwise suffisent).

Homologie simpliciale. Le groupe de k-chaînes à coefficients dans un corps $\mathbb{F}$ est $C_k(K; \mathbb{F}) = \mathbb{F}^{\#\{k\text{-simplexes de } K\}}$. L'opérateur bord $\partial_k : C_k \to C_{k-1}$ est défini par :

$$\partial_k([v_0, \dots, v_k]) = \sum_{i=0}^{k} (-1)^i [v_0, \dots, \widehat{v_i}, \dots, v_k].$$

On vérifie $\partial_{k-1} \circ \partial_k = 0$, ce qui donne le k-ième groupe d'homologie :

$$H_k(K; \mathbb{F}) = \ker(\partial_k) \,/\, \mathrm{im}(\partial_{k+1}).$$

Les **nombres de Betti** $\beta_k = \dim H_k(K; \mathbb{F})$ encodent : β_0 = composantes connexes, β_1 = boucles indépendantes, β_2 = cavités fermées.

Espace	β_0	β_1	β_2
Sphère S^2	1	0	1
Tore T^2	1	2	1
Plan projectif $\mathbb{RP}^2$ (coeff. $\mathbb{Q}$)	1	0	0
Bouteille de Klein (coeff. $\mathbb{Q}$)	1	1	0

Exercices clés de la phase.
- Démonstration complète que $\partial^2 = 0$.
- Calcul des groupes d'homologie du tore, de $\mathbb{RP}^2$ et de la bouteille de Klein.
- Démonstration du théorème d'invariance homotopique.
- Application de la suite de Mayer-Vietoris.

Ressources : Phase 1

Hatcher, *Algebraic Topology*, chapitres 2–3 ; Munkres, *Elements of Algebraic Topology* (1984) ; Forman (1998) pour la théorie de Morse discrète.

4.3 Phase 2 — Modules de persistance et décomposition (semaines 5–8)

Phase 2 — Modules de persistance et décomposition

Objectif de la phase : Définir rigoureusement les filtrations et les modules de persistance. Démontrer le théorème de décomposition. Implémenter l'algorithme de réduction de la matrice des bords.

Filtrations. Une *filtration* d'un complexe K est une famille croissante de sous-complexes indexée par des réels $t_0 < t_1 < \cdots < t_m$:

$$\emptyset = K_{-\infty} \subseteq K_{t_0} \subseteq \cdots \subseteq K_{t_m} = K.$$

Équivalemment, une filtration est une fonction $f : K \to \mathbb{R}$ telle que $f(\tau) \leq f(\sigma)$ dès que τ est une face de σ. Pour un nuage de points, la filtration naturelle est $K_\varepsilon = \mathrm{Rips}_\varepsilon(X)$, ε variant de 0 à $+\infty$.

Module de persistance. Un *module de persistance* (à valeurs dans $\mathbb{F}$-Vect) est un foncteur $V : (\mathbb{R}, \leq) \to \mathbb{F}\text{-Vect}$: une famille d'espaces vectoriels $(V_t)_{t\in\mathbb{R}}$ munie de morphismes de transition $v_{s,t} : V_s \to V_t$ pour $s \leq t$, compatibles par composition.
Le module de persistance associé à une filtration (K_t) est :

$$V_t = H_k(K_t; \mathbb{F}), \qquad v_{s,t} = (i_{s,t})_* : H_k(K_s) \to H_k(K_t),$$

où $i_{s,t} : K_s \hookrightarrow K_t$ est l'inclusion.

Théorème de décomposition. C'est le résultat central de la théorie algébrique :

Théorème de Zomorodian-Carlsson (2005)

Soit V un module de persistance à type de pointage fini. Alors V se décompose de manière **unique** en somme directe de modules intervalles :

$$V \cong \bigoplus_{j=1}^{N} \mathbb{F}[b_j, d_j),$$

où $\mathbb{F}[b,d)$ vaut $\mathbb{F}$ pour $t \in [b,d)$ et 0 ailleurs, les morphismes de transition étant l'identité.

Code-barres et diagramme de persistance. Le *code-barres* est l'ensemble des intervalles $\{[b_j, d_j)\}$. Le *diagramme de persistance* est le multi-ensemble de points :

$$\mathrm{Dgm}(V) = \{(b_j, d_j)\}_{j=1}^{N} \cup \Delta,$$

où $\Delta = \{(x, x) : x \in \mathbb{R}\}$ est la diagonale (ajoutée avec multiplicité infinie). La *persistance* d'une classe (b, d) est $d - b > 0$: grande persistance = trait topologique robuste ; faible persistance = bruit.

Algorithme de réduction. Le barcode se calcule efficacement par réduction de la matrice des bords ∂ du complexe filtré. La complexité est $O(n^3)$ dans le cas général, et souvent bien meilleure en pratique.

Ressources : Phase 2

Zomorodian & Carlsson (2005) ; Edelsbrunner & Harer (2010), chapitres 6–7 ; Chazal *et al.* (2016), chapitres 2–3.

4.4 Phase 3 — Stabilité et entrelacement (semaines 9–13)

Phase 3 — Stabilité et entrelacement

Objectif de la phase : Définir les distances entre diagrammes de persistance, démontrer le lemme d'entrelacement et le théorème de stabilité complet. C'est le cœur mathématique du projet.

Distance de bottleneck. Soient A et B deux diagrammes de persistance. La *distance de bottleneck* est :

$$d_b(A, B) = \inf_{\gamma : A \to B} \sup_{x \in A} \|x - \gamma(x)\|_\infty,$$

où l'infimum est pris sur toutes les bijections $\gamma : A \cup \Delta \to B \cup \Delta$ (les points peuvent être envoyés sur la diagonale). La distance de *Wasserstein* d'ordre p est :

$$W_p(A, B) = \inf_{\gamma : A \to B} \left(\sum_{x \in A} \|x - \gamma(x)\|_\infty^p \right)^{1/p}.$$

Entrelacement de modules. La notion clé pour la preuve est :

Définition : entrelacement

Deux modules V et W sont *δ-entrelacés* s'il existe des familles de morphismes $\phi_t : V_t \to W_{t+\delta}$ et $\psi_t : W_t \to V_{t+\delta}$ tels que, pour tous $s \leq t$:

$$\psi_{t+\delta} \circ \phi_t = v_{t,t+2\delta}, \qquad \phi_{t+\delta} \circ \psi_t = w_{t,t+2\delta}.$$

Lemme d'entrelacement (Chazal et al., 2009)

Si deux modules de persistance V et W sont δ-entrelacés, alors :

$$d_b(\mathrm{Dgm}(V), \mathrm{Dgm}(W)) \leq \delta.$$

Théorème de stabilité. C'est le résultat central :

Théorème de Stabilité (Cohen-Steiner, Edelsbrunner, Harer, 2007)

Soient $f, g : |K| \to \mathbb{R}$ deux fonctions de Morse sur la réalisation géométrique d'un complexe simplicial K. Alors :

$$\boxed{d_b\big(\mathrm{Dgm}(f),\, \mathrm{Dgm}(g)\big) \leq \|f - g\|_\infty.}$$

Stratégie de preuve. Si $\|f - g\|_\infty \leq \delta$, alors $|K|_{f \leq t} \subseteq |K|_{g \leq t+\delta}$ et réciproquement. Ceci fournit un δ-entrelacement des modules de persistance correspondants, et conclut par le lemme d'entrelacement.

Extension géométrique. La version la plus générale concerne des espaces métriques compacts :

Stabilité version géométrique

Soient (X, d_X) et (Y, d_Y) deux espaces métriques compacts, et $f : X \to \mathbb{R}$, $g : Y \to \mathbb{R}$ des fonctions Lipschitz. Alors :

$$d_b\big(\mathrm{Dgm}(f), \mathrm{Dgm}(g)\big) \leq 2\, d_{GH}(X, Y) + \|f \circ \varphi - g \circ \psi\|_\infty,$$

où d_{GH} est la distance de Gromov-Hausdorff et φ, ψ des correspondances approximatives.

Exercices clés de la phase.

- Démonstration détaillée que la distance de bottleneck est bien une distance (symétrie, inégalité triangulaire).
- Démonstration complète du lemme d'entrelacement.
- Démonstration du théorème de stabilité par entrelacement.
- Construction explicite d'un entrelacement à partir d'une inégalité $\|f - g\|_\infty \leq \delta$.

Ressources : Phase 3

Cohen-Steiner, Edelsbrunner & Harer (2007) [article fondateur, lecture intégrale obligatoire] ; Chazal *et al.* (2016), chapitres 4–5 ; Edelsbrunner & Harer (2010), chapitre 8.

4.5 Phase 4 — Projet de code : pipeline TDA complet (semaines 14–18)

Phase 4 — Projet de code : pipeline TDA

Objectif de la phase : Implémenter en Python un pipeline complet d'analyse topologique des données, depuis le nuage de points jusqu'à l'intégration dans un pipeline de classification. C'est le projet de code central du projet.

Architecture du projet. Le projet de code comporte cinq modules distincts, progressifs.

`Module code` : **Module 1 — Calcul du diagramme de persistance (ripser)**

Objectif : Calculer la filtration de Vietoris-Rips et les diagrammes de persistance H_0, H_1, H_2 à partir d'un nuage de points, en comparant l'implémentation naïve ($O(n^3)$) à la bibliothèque optimisée `ripser` (algorithme de Bauer).

```python
import numpy as np
import matplotlib.pyplot as plt
from ripser import ripser
from persim import plot_diagrams

def compute_persistence_naive(points: np.ndarray,
                              max_dim: int = 2) -> list:
    """
    Calcule les diagrammes de persistance par filtration de
        Vietoris-Rips.
    Utilise gudhi pour la construction du complexe.

    Parametres
    ----------
    points  : nuage de points, forme (n, d)
    max_dim : dimension maximale de l'homologie calculee

    Retourne
    --------
    diagrams : liste de tableaux (birth, death) par dimension
    """
    import gudhi

    rips_complex = gudhi.RipsComplex(points=points, max_edge_length=2.0)
    simplex_tree =
        rips_complex.create_simplex_tree(max_dimension=max_dim + 1)
    simplex_tree.compute_persistence()

    diagrams = []
    for dim in range(max_dim + 1):
        pairs = simplex_tree.persistence_intervals_in_dimension(dim)
        pairs = np.array(pairs)
        # Remplacer les morts infinies par une valeur finie pour la visu
        if len(pairs) > 0:
            pairs[pairs == np.inf] = 2.0
        diagrams.append(pairs)
    return diagrams


def compute_persistence_ripser(points: np.ndarray,
                               max_dim: int = 2) -> list:
    """
    Calcule les diagrammes de persistance via ripser (O(n^3) ameliore).
    Beaucoup plus rapide pour les grands nuages de points.
```

```
    """
    result = ripser(points, maxdim=max_dim)
    return result['dgms']


def compare_clouds(cloud_name: str, points: np.ndarray) -> None:
    """Visualise les diagrammes de persistance d'un nuage de points."""
    diagrams = compute_persistence_ripser(points, max_dim=2)

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))
    plot_diagrams(diagrams, ax=axes[0], show=False)
    axes[0].set_title(f'Diagramme de persistance -- {cloud_name}')

    # Barcode H1
    ax = axes[1]
    for i, (b, d) in enumerate(diagrams[1]):
        ax.plot([b, min(d, 2.0)], [i, i], 'b-', lw=2)
    ax.set_xlabel('Parametre epsilon')
    ax.set_ylabel('Classe H_1')
    ax.set_title(f'Barcode H1 -- {cloud_name}')
    ax.grid(alpha=0.3)

    plt.tight_layout()
    plt.savefig(f'persistence_{cloud_name}.pdf', dpi=150)
    plt.show()
```

Listing 1: Calcul du diagramme de persistance – naif vs ripser

Module code : Module 2 — Génération de nuages de points test

Objectif : Générer des nuages de points sur des variétés connues (cercle S^1, tore T^2, sphère S^2) pour valider numériquement les invariants topologiques attendus, et tester la robustesse au bruit.

```
def sample_circle(n: int, noise: float = 0.0,
                  rng=None) -> np.ndarray:
    """Nuage de points sur S^1 avec bruit gaussien. beta_1 = 1."""
    if rng is None:
        rng = np.random.default_rng()
    theta = rng.uniform(0, 2 * np.pi, n)
    points = np.column_stack([np.cos(theta), np.sin(theta)])
    return points + noise * rng.standard_normal(points.shape)


def sample_torus(n: int, R: float = 2.0, r: float = 1.0,
                 noise: float = 0.0, rng=None) -> np.ndarray:
    """Nuage de points sur T^2. beta_0=1, beta_1=2, beta_2=1."""
    if rng is None:
        rng = np.random.default_rng()
    theta = rng.uniform(0, 2 * np.pi, n)
    phi   = rng.uniform(0, 2 * np.pi, n)
    x = (R + r * np.cos(phi)) * np.cos(theta)
    y = (R + r * np.cos(phi)) * np.sin(theta)
    z = r * np.sin(phi)
    pts = np.column_stack([x, y, z])
    return pts + noise * rng.standard_normal(pts.shape)


def sample_sphere(n: int, noise: float = 0.0,
                  rng=None) -> np.ndarray:
```

```python
    """Nuage de points sur S^2. beta_0=1, beta_1=0, beta_2=1."""
    if rng is None:
        rng = np.random.default_rng()
    pts = rng.standard_normal((n, 3))
    pts /= np.linalg.norm(pts, axis=1, keepdims=True)
    return pts + noise * rng.standard_normal(pts.shape)


def robustness_experiment(n: int = 300,
                          noise_levels: list = None) -> None:
    """
    Valide empiriquement le theoreme de stabilite :
    le diagramme de persistance varie continument avec le bruit.
    """
    from persim import bottleneck
    if noise_levels is None:
        noise_levels = [0.0, 0.05, 0.1, 0.2, 0.4]

    rng = np.random.default_rng(42)
    clean_pts   = sample_circle(n, noise=0.0, rng=rng)
    clean_diags = compute_persistence_ripser(clean_pts, max_dim=1)

    db_distances = []
    for sigma in noise_levels:
        noisy_pts   = sample_circle(n, noise=sigma, rng=rng)
        noisy_diags = compute_persistence_ripser(noisy_pts, max_dim=1)
        # Distance de bottleneck sur H_1
        db = bottleneck(clean_diags[1], noisy_diags[1])
        db_distances.append(db)
        print(f"Bruit sigma={sigma:.2f}  =>  d_bottleneck(H1) =
            {db:.4f}")

    return db_distances
```

Listing 2: Generation de nuages de points sur des varietes

Module code : Module 3 — Images de persistance et plongement vectoriel

Objectif : Convertir les diagrammes de persistance en vecteurs de L^2 (images de persistance, Adams et al., 2017) pour les rendre compatibles avec les algorithmes de *machine learning* classiques.

```python
from persim import PersistenceImager

def compute_persistence_image(diagram: np.ndarray,
                              pixel_size: float = 0.1,
                              sigma: float = 0.1) -> np.ndarray:
    """
    Calcule l'image de persistance d'un diagramme (Adams et al., 2017).
    Plonge le diagramme dans L^2 par convolution gaussienne sur
    le plan naissance/persistance (birth, death - birth).

    Parametres
    ----------
    diagram    : tableau (n, 2) de points (birth, death)
    pixel_size : resolution de la grille
    sigma      : ecart-type du noyau gaussien

    Retourne
    --------
```

```
    image : tableau 2D (vecteur aplati pour le ML)
    """
    # Filtrer les points infinis
    finite_mask = np.isfinite(diagram[:, 1])
    diag_finite = diagram[finite_mask]

    pimgr = PersistenceImager(birth_range=(0, 2),
                              pers_range=(0, 2),
                              pixel_size=pixel_size,
                              weight='linear',
                              kernel_params={'sigma': sigma})
    pimgr.fit([diag_finite])
    image = pimgr.transform([diag_finite])[0]
    return image


def build_feature_matrix(point_clouds: list,
                         max_dim: int = 1) -> np.ndarray:
    """
    Construit la matrice de features TDA pour un ensemble
    de nuages de points : chaque ligne = concatenation des
    images de persistance H0, H1 (et H2 si max_dim=2).

    Retourne X de forme (n_samples, n_features).
    """
    features = []
    for cloud in point_clouds:
        diagrams = compute_persistence_ripser(cloud, max_dim=max_dim)
        imgs = []
        for dim in range(max_dim + 1):
            diag = diagrams[dim]
            if len(diag) == 0:
                imgs.append(np.zeros(400))  # 20x20 pixels par defaut
            else:
                img = compute_persistence_image(diag)
                imgs.append(img.ravel())
        features.append(np.concatenate(imgs))
    return np.array(features)
```

Listing 3: Images de persistance – plongement dans L2

Module code : **Module 4 — Pipeline de classification TDA**

Objectif : Intégrer les features topologiques dans un pipeline scikit-learn complet pour la classification de nuages de points. Comparer les performances avec et sans features topologiques.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report

def classification_experiment(n_per_class: int = 100,
                              noise: float = 0.1) -> dict:
    """
    Classifie des nuages de points issus de trois topologies :
      Classe 0 : cercle S^1   (H1 = Z)
      Classe 1 : tore T^2     (H1 = Z^2)
      Classe 2 : sphere S^2   (H1 = 0, H2 = Z)
```

```
14
15      Evalue un SVM a noyau RBF avec features TDA (images de persistance)
16      et compare a une baseline purement geometrique (nuage de points
            aplati).
17
18      Retourne
19      --------
20      dict avec scores de cross-validation pour TDA et baseline.
21      """
22      rng = np.random.default_rng(0)
23
24      clouds, labels = [], []
25      for _ in range(n_per_class):
26          clouds.append(sample_circle(200, noise=noise, rng=rng));
                labels.append(0)
27          clouds.append(sample_torus(200, noise=noise, rng=rng));
                labels.append(1)
28          clouds.append(sample_sphere(200, noise=noise, rng=rng));
                labels.append(2)
29
30      labels = np.array(labels)
31
32      # Features TDA (images de persistance H0 + H1 + H2)
33      print("Calcul des features topologiques ...")
34      X_tda = build_feature_matrix(clouds, max_dim=2)
35
36      pipeline_tda = Pipeline([
37          ('scaler', StandardScaler()),
38          ('svm',    SVC(kernel='rbf', C=10, gamma='scale')),
39      ])
40      scores_tda = cross_val_score(pipeline_tda, X_tda, labels,
41                                   cv=5, scoring='accuracy')
42      print(f"TDA  -- accuracy = {scores_tda.mean():.3f} "
43            f"(+/- {scores_tda.std():.3f})")
44
45      return {'tda_scores': scores_tda}
```

Listing 4: Pipeline de classification avec features topologiques

Module code : **Module 5 — Validation empirique du théorème de stabilité**

Objectif : Vérifier numériquement que la distance de bottleneck entre diagrammes de persistance est bornée par la perturbation de la fonction de filtration, conformément au théorème de stabilité.

```
1  from persim import bottleneck
2  import matplotlib.pyplot as plt
3
4  def validate_stability_theorem(n: int = 500,
5                                 epsilons: list = None) -> None:
6      """
7      Valide le theoreme : d_b(Dgm(f), Dgm(f + eps)) <= eps.
8
9      Construit une filtration sur un nuage de points via une fonction
10     de hauteur f(x,y,z) = z (coordonnee verticale), perturbe f
11     d'une quantite epsilon, et mesure la distance de bottleneck.
12
13     La borne theorique est epsilon ; on trace les deux ensemble.
14     """
15     if epsilons is None:
```

```
        epsilons = np.linspace(0, 0.5, 15)

    rng = np.random.default_rng(123)
    pts = sample_torus(n, rng=rng)    # Nuage sur T^2

    db_measured = []

    for eps in epsilons:
        # Filtration par la coordonnee z (fonction de Morse)
        # Perturbee de epsilon en norme infinie
        pts_perturbed = pts.copy()
        pts_perturbed[:, 2] += eps    # Perturbation uniforme

        diags_orig = compute_persistence_ripser(pts, max_dim=1)
        diags_pert = compute_persistence_ripser(pts_perturbed,
            max_dim=1)

        db = bottleneck(diags_orig[1], diags_pert[1])
        db_measured.append(db)

    # Trace : borne theorique vs mesure empirique
    plt.figure(figsize=(7, 5))
    plt.plot(epsilons, epsilons, 'r--', lw=2,
             label=r'Borne theorique $\|f - g\|_\infty$')
    plt.plot(epsilons, db_measured, 'bo-', ms=5,
             label=r'$d_b(\mathrm{Dgm}(f), \mathrm{Dgm}(g))$ mesure')
    plt.xlabel(r'Perturbation $\varepsilon$')
    plt.ylabel(r'Distance de bottleneck $d_b$')
    plt.title('Validation empirique du theoreme de stabilite (H1,
        tore)')
    plt.legend()
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig('stability_validation.pdf', dpi=150)
    plt.show()
```

Listing 5: Validation numerique du theoreme de stabilite

Résultats attendus

À l'issue de la Phase 4, on disposera de :

1. Un pipeline TDA validé, calculant les diagrammes de persistance H_0, H_1, H_2 pour des nuages de points sur le cercle, le tore et la sphère, en accord avec les invariants théoriques.
2. Une validation numérique du théorème de stabilité : la distance de bottleneck entre diagrammes reste bien en dessous de la borne théorique $\|f - g\|_\infty$, même pour des nuages bruités.
3. Une matrice de features topologiques (images de persistance) permettant la classification de nuages de points par SVM avec une précision supérieure à une baseline géométrique naïve.
4. Un rapport de robustesse documentant l'influence du bruit sur les diagrammes, illustrant quantitativement la stabilité.

Ressources : Phase 4

Documentation `gudhi` et `ripser` ; Adams *et al.* (2017) pour les images de persistance ; Tralie *et al.* (2018) pour `ripser` ; documentation `scikit-learn` pour les pipelines ML.

4.6 Phase 4b — Projet avancé : TDA appliquée à la classification EEG (semaines 17–20)

Phase 4b — Projet avancé : TDA pour la classification EEG

Objectif de la phase : Construire un pipeline de machine learning complet et rigoureux exploitant l'homologie persistante pour la **classification de signaux EEG** (*électroencéphalogramme*) — l'une des applications les plus actives de la TDA en neurosciences computationnelles. Le défi est double : les signaux EEG sont non stationnaires, bruités et de haute dimension, ce qui en fait un terrain idéal pour valider la robustesse topologique. Le projet compare plusieurs vectorisations de diagrammes de persistance contre des baselines spectrales classiques.

Dataset. On utilise le jeu de données public **BCI Competition IV Dataset 2a** (Brunner et al., 2008) — disponible sur `https://www.bbci.de/competition/iv/` — qui contient des signaux EEG à 22 électrodes, 250 Hz, pour 4 classes d'imagerie motrice (main gauche, main droite, pieds, langue), 9 sujets, 288 essais par sujet. Objectif de classification : 4 classes, baseline état-de-l'art $\approx$ 68% d'accuracy, meilleurs modèles topologiques $\approx$ 72–74%.

Architecture du projet avancé. Le projet comporte sept modules, organisés en trois niveaux : **(A) représentation**, **(B) vectorisation**, **(C) apprentissage**.

`Module code` : **Module A1 — Chargement, prétraitement et plongement de Takens**

Objectif : Transformer chaque essai EEG (signal multivariable de dimension 22) en un nuage de points via le *plongement de Takens* (théorème de Takens, 1981), qui reconstruit la topologie de l'attracteur sous-jacent à partir de la dynamique observée. C'est l'étape clé qui justifie l'usage de la TDA sur des séries temporelles.

```python
import numpy as np
import scipy.io as sio
from scipy.signal import butter, filtfilt, welch
from sklearn.preprocessing import StandardScaler
from ripser import ripser
import warnings
warnings.filterwarnings('ignore')


#            Chargement du dataset BCI-IV-2a (format GDF via MNE)


def load_bci_data(subject_id: int, data_dir: str = './data') -> dict:
    """
    Charge les donnees EEG du sujet subject_id (1-9).
    Retourne un dict avec cles 'X' (n_trials, n_channels, n_times)
    et 'y' (n_trials,) de labels {0,1,2,3}.
    Necessite : pip install mne
    """
    import mne
    fname = f'{data_dir}/A0{subject_id}T.gdf'
    raw = mne.io.read_raw_gdf(fname, preload=True, verbose=False)

    # Filtrage passe-bande mu/beta (8-30 Hz, bandes motrices)
    raw.filter(8., 30., fir_window='hamming', verbose=False)
    # Rereferencement moyenne commune
    raw.set_eeg_reference('average', verbose=False)

```

```
    events, event_id = mne.events_from_annotations(raw, verbose=False)
    # Classes : 769=gauche, 770=droite, 771=pieds, 772=langue
    class_ids = {'left': 769, 'right': 770, 'feet': 771, 'tongue': 772}
    epochs = mne.Epochs(raw, events, event_id=class_ids,
                        tmin=0.5, tmax=2.5,  # fenetre post-cue
                        baseline=None, preload=True, verbose=False)
    X = epochs.get_data()           # shape (n_trials, 22, 500)
    y = epochs.events[:, -1] - 769 # labels {0,1,2,3}
    return {'X': X, 'y': y, 'sfreq': raw.info['sfreq'],
            'ch_names': raw.ch_names[:22]}


#           Plongement de Takens


def takens_embedding(signal: np.ndarray,
                     dim: int = 3,
                     delay: int = 5) -> np.ndarray:
    """
    Plongement de Takens d'un signal 1D (n_times,).

    Construit la matrice de coordonnees retardees :
      x(t) = [s(t), s(t+tau), s(t+2*tau), ..., s(t+(d-1)*tau)]

    Parametres
    ----------
    signal : serie temporelle 1D, forme (T,)
    dim    : dimension du plongement (typiquement 2-5)
    delay  : retard temporel tau (en echantillons)

    Retourne
    --------
    cloud : ndarray de forme (T - (dim-1)*delay, dim)
    """
    T = len(signal)
    n_pts = T - (dim - 1) * delay
    cloud = np.zeros((n_pts, dim))
    for i in range(dim):
        cloud[:, i] = signal[i * delay : i * delay + n_pts]
    return cloud


def multivariate_takens(trial: np.ndarray,
                        channels: list,
                        dim: int = 3,
                        delay: int = 5,
                        max_pts: int = 300) -> np.ndarray:
    """
    Plongement de Takens multivari  : concatenation des plongements
    de plusieurs canaux EEG selectionn s, puis sous-  chantillonnage
    uniforme pour controler la complexite computationnelle.

    trial    : (n_channels, n_times)
    channels : liste d'indices de canaux
    Retourne : nuage de points unifie, forme (max_pts, dim *
       len(channels))
    """
    clouds = []
    for ch in channels:
        c = takens_embedding(trial[ch], dim=dim, delay=delay)
        clouds.append(c)
    # Troncature au minimum
```

```
    min_len = min(c.shape[0] for c in clouds)
    clouds = [c[:min_len] for c in clouds]
    cloud = np.hstack(clouds)  # (min_len, dim * n_ch)
    # Sous-echantillonnage uniforme
    idx = np.linspace(0, len(cloud) - 1, min(max_pts, len(cloud)),
                      dtype=int)
    return cloud[idx]


#           Pipeline de pr traitement complet


# Canaux moteurs pertinents (C3, Cz, C4 et regions avoisinantes)
MOTOR_CHANNELS = [0, 7, 10, 11, 12, 13, 14]  # indices dans le montage

def preprocess_dataset(data: dict,
                       dim: int = 3,
                       delay: int = 4,
                       max_pts: int = 250) -> tuple:
    """
    Transforme tous les essais EEG en nuages de points via Takens.
    Retourne X_clouds (liste de nuages) et y (labels).
    """
    X_raw, y = data['X'], data['y']
    X_clouds = []
    for trial in X_raw:
        # Normalisation par canal
        trial_norm = (trial - trial.mean(axis=1, keepdims=True)) \
                     / (trial.std(axis=1, keepdims=True) + 1e-8)
        cloud = multivariate_takens(trial_norm, MOTOR_CHANNELS,
                                    dim=dim, delay=delay,
                                        max_pts=max_pts)
        X_clouds.append(cloud)
    return X_clouds, y
```

Listing 6: Chargement EEG et plongement de Takens multivarie

Module code : Module A2 — Calcul adaptatif des diagrammes de persistance

Objectif : Calculer les diagrammes de persistance H_0, H_1, H_2 pour chaque nuage de points issu du plongement de Takens. Deux stratégies sont implémentées : filtration de Vietoris-Rips globale (un seul diagramme par essai) et stratégie *sliding-window* (séquence de diagrammes, capturant la non-stationnarité).

```
from ripser import ripser
from persim import bottleneck, sliced_wasserstein
from joblib import Parallel, delayed
import time


def compute_diagram_single(cloud: np.ndarray,
                           max_dim: int = 2,
                           max_edge: float = None) -> list:
    """
    Calcule les diagrammes de persistance H0, H1, H2 d'un nuage.
    max_edge : seuil de troncature de la filtration (heuristique :
               75eme percentile des distances pairwise).
    """
    if max_edge is None:
```

```
        from sklearn.metrics import pairwise_distances
        dists = pairwise_distances(cloud)
        max_edge = float(np.percentile(dists, 75))
    result = ripser(cloud, maxdim=max_dim, thresh=max_edge)
    return result['dgms']


def compute_diagrams_parallel(clouds: list,
                              max_dim: int = 2,
                              n_jobs: int = -1) -> list:
    """
    Calcule les diagrammes pour tous les essais en parallele.
    Retourne une liste de listes de diagrammes (un par essai).
    """
    t0 = time.time()
    results = Parallel(n_jobs=n_jobs, prefer='threads')(
        delayed(compute_diagram_single)(c, max_dim) for c in clouds
    )
    print(f"Diagrammes calcules en {time.time()-t0:.1f}s "
          f"({len(clouds)} essais, {n_jobs} workers)")
    return results


#           Strat gie sliding-window


def sliding_window_diagrams(cloud: np.ndarray,
                            window: int = 80,
                            stride: int = 40,
                            max_dim: int = 1) -> list:
    """
    Calcule une sequence de diagrammes de persistance par fenetrage
    glissant sur le nuage de points. Capture la dynamique temporelle
    de la topologie : idee de Perea & Harer (2015).

    Retourne une liste de diagrammes H1 (un par fenetre).
    """
    n = len(cloud)
    starts = range(0, n - window + 1, stride)
    diagrams = []
    for s in starts:
        sub = cloud[s : s + window]
        dgm = compute_diagram_single(sub, max_dim=max_dim)
        diagrams.append(dgm[1])  # H1 uniquement
    return diagrams


def wasserstein_distance_matrix(diagrams_h1: list,
                                n_jobs: int = -1) -> np.ndarray:
    """
    Calcule la matrice de distances de Wasserstein entre diagrammes H1.
    Utile pour k-NN et SVM a noyau topologique.
    """
    n = len(diagrams_h1)
    D = np.zeros((n, n))
    pairs = [(i, j) for i in range(n) for j in range(i+1, n)]

    def compute_pair(i, j):
        return sliced_wasserstein(diagrams_h1[i], diagrams_h1[j])

    results = Parallel(n_jobs=n_jobs)(
        delayed(compute_pair)(i, j) for i, j in pairs
```

```
    )
    for (i, j), d in zip(pairs, results):
        D[i, j] = D[j, i] = d
    return D
```

Listing 7: Diagrammes de persistance : global et sliding-window

Module code : Module B1 — Vectorisations topologiques multiples

Objectif : Implémenter quatre stratégies de vectorisation des diagrammes de persistance, toutes stables par rapport à la distance de bottleneck : *courbes de Betti*, *paysages de persistance* (Bubenik, 2015), *silhouettes topologiques* (Chazal et al., 2014) et *images de persistance* (Adams et al., 2017). Ces représentations constituent quatre familles d'invariants aux propriétés statistiques distinctes.

```
import numpy as np
from persim import PersistenceImager
from scipy.interpolate import interp1d


#           1. Courbes de Betti


def betti_curve(diagram: np.ndarray,
                t_grid: np.ndarray = None,
                resolution: int = 100) -> np.ndarray:
    """
    Courbe de Betti beta(t) : nombre de classes topologiques vivantes
    a l'echelle t. Stable par rapport a la distance de bottleneck.

    diagram : tableau (n, 2) de points (birth, death)
    Retourne : vecteur de dimension resolution
    """
    if t_grid is None:
        finite = diagram[np.isfinite(diagram[:, 1])]
        if len(finite) == 0:
            return np.zeros(resolution)
        t_min = finite[:, 0].min()
        t_max = finite[:, 1].max()
        t_grid = np.linspace(t_min, t_max, resolution)

    curve = np.zeros(len(t_grid))
    for t_i, t in enumerate(t_grid):
        # Compte les intervalles [b, d) contenant t
        mask = (diagram[:, 0] <= t) & (diagram[:, 1] > t)
        curve[t_i] = mask.sum()
    return curve


#           2. Paysages de persistance (Bubenik, 2015)


def persistence_landscape(diagram: np.ndarray,
                          k_max: int = 5,
                          resolution: int = 100) -> np.ndarray:
    """
    Calcule les k premiers paysages de persistance
        lambda_1,...,lambda_k.

```

```
    Le k-ieme paysage est lambda_k(t) = k-ieme plus grande valeur de
    min(t - b, d - t) parmi les classes [b,d) contenant t.

    Stabilite : ||lambda(f) - lambda(g)||_infty <= d_b(Dgm(f), Dgm(g)).

    Retourne : tableau (k_max, resolution) -- concatener pour un vecteur
    """
    finite = diagram[np.isfinite(diagram[:, 1])]
    if len(finite) == 0:
        return np.zeros((k_max, resolution))

    t_min = finite[:, 0].min()
    t_max = finite[:, 1].max()
    t_grid = np.linspace(t_min, t_max, resolution)

    landscapes = np.zeros((k_max, resolution))
    for t_i, t in enumerate(t_grid):
        # Valeurs de la fonction "tente" pour chaque classe
        values = []
        for (b, d) in finite:
            if b <= t <= d:
                values.append(min(t - b, d - t))
        values.sort(reverse=True)
        for k in range(min(k_max, len(values))):
            landscapes[k, t_i] = values[k]
    return landscapes


#           3. Silhouettes topologiques (Chazal et al., 2014)


def persistence_silhouette(diagram: np.ndarray,
                           p: float = 1.0,
                           resolution: int = 100) -> np.ndarray:
    """
    Silhouette de persistance d'ordre p : moyenne ponderee des fonctions
    "tente", les poids etant (d-b)^p / sum (d_j-b_j)^p.

    Pour p=1 : silhouette standard. Pour p->infty : converge vers le
    paysage lambda_1.
    """
    finite = diagram[np.isfinite(diagram[:, 1])]
    if len(finite) == 0:
        return np.zeros(resolution)

    persistences = finite[:, 1] - finite[:, 0]
    weights = persistences ** p
    weights /= weights.sum() + 1e-12

    t_min, t_max = finite[:, 0].min(), finite[:, 1].max()
    t_grid = np.linspace(t_min, t_max, resolution)
    silhouette = np.zeros(resolution)

    for t_i, t in enumerate(t_grid):
        for i, (b, d) in enumerate(finite):
            if b <= t <= d:
                silhouette[t_i] += weights[i] * min(t - b, d - t)
    return silhouette


#           4. Images de persistance (Adams et al., 2017)
```

```python

def persistence_image_batch(diagrams: list,
                            resolution: int = 20,
                            sigma: float = None) -> np.ndarray:
    """
    Calcule les images de persistance pour une liste de diagrammes H1.
    Plongement dans R^(resolution^2) via convolution gaussienne sur
    le plan (naissance, persistance).

    Retourne X de forme (n_trials, resolution^2).
    """
    # Ajustement automatique de sigma
    all_pers = []
    for dgm in diagrams:
        fin = dgm[np.isfinite(dgm[:, 1])]
        if len(fin) > 0:
            all_pers.extend((fin[:, 1] - fin[:, 0]).tolist())
    if sigma is None:
        sigma = np.std(all_pers) * 0.5 if all_pers else 0.1

    pimgr = PersistenceImager(
        birth_range=(0.0, np.percentile(all_pers, 95) if all_pers else
            1.0),
        pers_range=(0.0, np.percentile(all_pers, 95) if all_pers else
            1.0),
        pixel_size=(np.percentile(all_pers, 95) / resolution
                    if all_pers else 0.05),
        weight='linear',
        kernel_params={'sigma': sigma}
    )
    diagrams_h1 = [d[np.isfinite(d[:, 1])] for d in diagrams]
    pimgr.fit(diagrams_h1)
    images = pimgr.transform(diagrams_h1)
    return np.array([img.ravel() for img in images])


#           Constructeur de la matrice de features compl te


def build_full_feature_matrix(all_diagrams: list,
                              resolution: int = 50) -> dict:
    """
    Construit toutes les vectorisations pour H0 et H1.
    Retourne un dict {nom -> matrice (n_trials, n_features)}.
    """
    n = len(all_diagrams)
    features = {}

    # Courbes de Betti H0 et H1
    betti_H0 = np.array([betti_curve(d[0], resolution=resolution)
                         for d in all_diagrams])
    betti_H1 = np.array([betti_curve(
                             d[1][np.isfinite(d[1][:, 1])] if len(d[1])
                                 > 0
                             else np.zeros((0, 2)),
                             resolution=resolution)
                         for d in all_diagrams])
    features['betti'] = np.hstack([betti_H0, betti_H1])

    # Paysages de persistance H1 (5 premiers)
    landscapes = np.array([
        persistence_landscape(
```

```python
            d[1][np.isfinite(d[1][:, 1])] if len(d[1]) > 0
            else np.zeros((0, 2)),
            k_max=5, resolution=resolution).ravel()
        for d in all_diagrams
    ])
    features['landscape'] = landscapes

    # Silhouettes H1 p=1 et p=2
    sil1 = np.array([persistence_silhouette(
                        d[1][np.isfinite(d[1][:, 1])] if len(d[1]) > 0
                        else np.zeros((0, 2)), p=1.0,
                        resolution=resolution)
                     for d in all_diagrams])
    sil2 = np.array([persistence_silhouette(
                        d[1][np.isfinite(d[1][:, 1])] if len(d[1]) > 0
                        else np.zeros((0, 2)), p=2.0,
                        resolution=resolution)
                     for d in all_diagrams])
    features['silhouette'] = np.hstack([sil1, sil2])

    # Images de persistance H1
    diagrams_h1 = [d[1] for d in all_diagrams]
    features['pi'] = persistence_image_batch(diagrams_h1,
                                             resolution=20)

    # Features statistiques brutes (baseline topologique)
    stats = []
    for d in all_diagrams:
        row = []
        for dim in [0, 1]:
            dgm = d[dim]
            fin = dgm[np.isfinite(dgm[:, 1])]
            if len(fin) == 0:
                row += [0.] * 5
                continue
            pers = fin[:, 1] - fin[:, 0]
            row += [len(pers), pers.max(), pers.mean(),
                    pers.std(), np.sum(pers)]
        stats.append(row)
    features['stats'] = np.array(stats)

    # Concatenation totale
    features['all'] = np.hstack(list(features.values()))

    print("Dimensions des features :")
    for k, v in features.items():
        print(f"  {k:12s} : {v.shape}")
    return features
```

Listing 8: Quatre vectorisations de diagrammes de persistance

Module code : Module B2 — Baseline spectrale et features hybrides

Objectif : Implémenter la baseline classique en BCI — les *Common Spatial Patterns* (CSP) et la *puissance spectrale par bande* — pour comparaison rigoureuse avec les features TDA. Construire également un vecteur *hybride* TDA + spectral.

```python
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.pipeline import Pipeline
from mne.decoding import CSP
```

```python
from scipy.signal import welch


#            Features spectrales classiques


def spectral_features(trial: np.ndarray, sfreq: float,
                      channels=MOTOR_CHANNELS) -> np.ndarray:
    """
    Calcule la puissance dans les bandes mu (8-12 Hz) et beta (13-30 Hz)
    pour les canaux moteurs. Feature classique en BCI.
    """
    feats = []
    for ch in channels:
        freqs, psd = welch(trial[ch], fs=sfreq, nperseg=128)
        mu_idx   = (freqs >= 8)  & (freqs <= 12)
        beta_idx = (freqs >= 13) & (freqs <= 30)
        feats.append(np.log(psd[mu_idx].mean() + 1e-10))
        feats.append(np.log(psd[beta_idx].mean() + 1e-10))
    return np.array(feats)


def build_spectral_matrix(X_raw: np.ndarray, sfreq: float) ->
    np.ndarray:
    """Matrice de features spectrales pour tous les essais."""
    return np.array([spectral_features(trial, sfreq) for trial in
        X_raw])


#            Covariance Riemannienne (Barachant et al., 2012)


def covariance_features(X_raw: np.ndarray,
                        channels=MOTOR_CHANNELS) -> np.ndarray:
    """
    Features basees sur la geometrie riemannienne des matrices de
        covariance
    (pyriemann). Consideree state-of-the-art en BCI classique.
    Necessite : pip install pyriemann
    """
    import pyriemann
    from pyriemann.estimation import Covariances
    from pyriemann.tangentspace import TangentSpace

    X_ch = X_raw[:, channels, :]  # (n_trials, n_ch, n_times)
    cov = Covariances(estimator='oas').fit_transform(X_ch)
    return TangentSpace(metric='riemann').fit_transform(cov)


#            Features hybrides TDA + spectral


def build_hybrid_features(tda_feats: np.ndarray,
                          spectral_feats: np.ndarray,
                          alpha: float = 1.0) -> np.ndarray:
    """
    Concatene features TDA et spectrales apres normalisation.
    alpha controle le poids relatif (exploration par validation
        croisee).
    """
    from sklearn.preprocessing import StandardScaler
    scaler_tda  = StandardScaler().fit(tda_feats)
```

```
    scaler_spec = StandardScaler().fit(spectral_feats)
    return np.hstack([
        scaler_tda.transform(tda_feats),
        alpha * scaler_spec.transform(spectral_feats)
    ])
```

Listing 9: Baseline CSP et features spectrales

Module code : Module C1 — Benchmark multi-classifieur avec validation croisée stratifiée

Objectif : Évaluer et comparer rigoureusement chaque représentation (TDA et baselines) avec plusieurs classifieurs via une *validation croisée stratifiée* à 5 folds. Utiliser la *correction de Bonferroni* pour les tests statistiques de comparaison de performances (test de McNemar ou Wilcoxon).

```
from sklearn.model_selection import StratifiedKFold, cross_validate
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import balanced_accuracy_score
from scipy.stats import wilcoxon
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import seaborn as sns


CLASSIFIERS = {
    'SVM-RBF':    SVC(kernel='rbf', C=10, gamma='scale',
        class_weight='balanced'),
    'SVM-Lin':    SVC(kernel='linear', C=1.0, class_weight='balanced'),
    'RF':         RandomForestClassifier(n_estimators=300, max_depth=8,
                                         class_weight='balanced',
                                            n_jobs=-1),
    'MLP':        MLPClassifier(hidden_layer_sizes=(256, 128, 64),
                                max_iter=500, learning_rate_init=1e-3,
                                early_stopping=True, random_state=42),
}


def evaluate_representation(X: np.ndarray, y: np.ndarray,
                            rep_name: str,
                            cv: StratifiedKFold,
                            verbose: bool = True) -> dict:
    """
    Evalue toutes les combinaisons (representation, classifieur)
    en validation croisee. Retourne un dict de scores par classifieur.
    """
    results = {}
    for clf_name, clf in CLASSIFIERS.items():
        pipe = Pipeline([
            ('scaler', StandardScaler()),
            ('clf',    clf)
        ])
```

```
        scores = cross_validate(
            pipe, X, y, cv=cv,
            scoring='balanced_accuracy',
            return_train_score=True,
            n_jobs=1
        )
        test_scores = scores['test_score']
        results[clf_name] = {
            'mean':  test_scores.mean(),
            'std':   test_scores.std(),
            'scores': test_scores,
        }
        if verbose:
            print(f"  [{rep_name}] {clf_name:10s} : "
                  f"{test_scores.mean()*100:.1f}% "
                  f"+/- {test_scores.std()*100:.1f}%")
    return results


def full_benchmark(features: dict, y: np.ndarray,
                   n_splits: int = 5) -> pd.DataFrame:
    """
    Benchmark complet sur toutes les representations et tous les
    classifieurs. Retourne un DataFrame pour visualisation.
    """
    cv = StratifiedKFold(n_splits=n_splits, shuffle=True,
        random_state=0)
    all_results = {}

    for rep_name, X in features.items():
        print(f"\n--- Representation : {rep_name} ({X.shape[1]}
            features) ---")
        all_results[rep_name] = evaluate_representation(
            X, y, rep_name, cv, verbose=True)

    # Construction du DataFrame de resultats
    rows = []
    for rep, clfs in all_results.items():
        for clf, res in clfs.items():
            for fold_i, s in enumerate(res['scores']):
                rows.append({'Representation': rep, 'Classifieur': clf,
                             'Fold': fold_i, 'Balanced_Accuracy': s *
                                100})
    df = pd.DataFrame(rows)
    return df, all_results


def statistical_comparison(all_results: dict,
                           ref_rep: str = 'spectral') -> pd.DataFrame:
    """
    Test de Wilcoxon (non parametrique) entre chaque representation TDA
    et la baseline spectrale, avec correction de Bonferroni.
    H0 : les distributions de scores sont identiques.
    """
    ref_scores = np.mean([
        all_results[ref_rep][clf]['scores']
        for clf in CLASSIFIERS
    ], axis=0)

    n_tests = len(all_results) - 1
    comp_rows = []
    for rep_name, clfs in all_results.items():
```

```
        if rep_name == ref_rep:
            continue
        tda_scores = np.mean([clfs[clf]['scores']
                              for clf in CLASSIFIERS], axis=0)
        stat, p_val = wilcoxon(tda_scores, ref_scores)
        p_corrected = min(p_val * n_tests, 1.0)  # Bonferroni
        comp_rows.append({
            'Representation': rep_name,
            'Mean_delta': (tda_scores.mean() - ref_scores.mean()) * 100,
            'p_value': p_val,
            'p_Bonferroni': p_corrected,
            'Significant': p_corrected < 0.05,
        })
    return pd.DataFrame(comp_rows).sort_values('Mean_delta',
        ascending=False)


def plot_benchmark(df: pd.DataFrame) -> None:
    """
    Visualisation du benchmark : boxplot par representation et
        classifieur,
    et heatmap de la matrice de confusion de la meilleure configuration.
    """
    fig, axes = plt.subplots(1, 2, figsize=(15, 6))

    # Boxplot global
    order = (df.groupby('Representation')['Balanced_Accuracy']
               .mean().sort_values(ascending=False).index.tolist())
    sns.boxplot(data=df, x='Representation', y='Balanced_Accuracy',
                hue='Classifieur', order=order, ax=axes[0])
    axes[0].axhline(25, color='r', linestyle='--', alpha=0.5,
                    label='Chance (25%)')
    axes[0].set_title('Benchmark multi-representation x
        multi-classifieur')
    axes[0].set_xlabel('')
    axes[0].tick_params(axis='x', rotation=30)
    axes[0].legend(loc='lower right', fontsize=8)
    axes[0].set_ylabel('Balanced Accuracy (%)')

    # Heatmap des moyennes
    pivot = df.groupby(['Representation', 'Classifieur'])[
        'Balanced_Accuracy'].mean().unstack()
    pivot = pivot.loc[order]
    sns.heatmap(pivot, annot=True, fmt='.1f', cmap='YlOrRd',
                vmin=25, vmax=75, ax=axes[1],
                linewidths=0.5, cbar_kws={'label': 'Acc. (%)'})
    axes[1].set_title('Heatmap : accuracy moyenne par configuration')

    plt.tight_layout()
    plt.savefig('benchmark_eeg_tda.pdf', dpi=150, bbox_inches='tight')
    plt.show()
```

Listing 10: Benchmark complet : 6 representations x 4 classifieurs

Module code : Module C2 — Couche de persistance différentiable et réseau de neurones

Objectif : Intégrer l'homologie persistante directement dans un réseau de neurones via une *couche de persistance* différentiable (Hofer et al., NeurIPS 2019). Cette approche permet d'apprendre la représentation topologique conjointement avec le classifieur, sans

vectorisation manuelle. C'est la direction la plus prometteuse pour l'avenir de la TDA en deep learning.

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
import numpy as np


#            Couche de persistance diff rentiable


class TopologicalSignatureLayer(nn.Module):
    """
    Implementation simplifiee d'une couche topologique differentiable.
    Calcule une signature fixe (images de persistance precalculees)
    et applique un MLP appris -- architecture de Hofer et al. (2019).

    Pour une version entierement differentiable bout-en-bout (avec
    subgradients sur les diagrammes), voir la librairie 'giotto-tda'
    ou 'TopologyLayer' (Gabrielsson et al., ICML 2020).
    """
    def __init__(self, input_dim: int, hidden_dims: list,
                 n_classes: int, dropout: float = 0.3):
        super().__init__()
        layers = []
        in_d = input_dim
        for h in hidden_dims:
            layers += [nn.Linear(in_d, h), nn.BatchNorm1d(h),
                       nn.ReLU(), nn.Dropout(dropout)]
            in_d = h
        layers.append(nn.Linear(in_d, n_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.net(x)


class EarlyStopping:
    """Arret precoce base sur la loss de validation."""
    def __init__(self, patience: int = 15, min_delta: float = 1e-4):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = np.inf
        self.should_stop = False

    def step(self, val_loss: float) -> None:
        if val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.counter = 0
        else:
            self.counter += 1
            if self.counter >= self.patience:
                self.should_stop = True


def train_topology_net(X_tda: np.ndarray, y: np.ndarray,
                       n_classes: int = 4,
                       hidden_dims: list = None,
```

```
                        n_epochs: int = 200,
                        batch_size: int = 32,
                        lr: float = 1e-3,
                        device: str = 'cpu') -> dict:
    """
    Entraine un reseau de neurones profond sur les features TDA.
    Utilise l'arret precoce, le scheduler cosine, et la regularisation
        L2.

    Retourne un dict avec historique de training et meilleurs poids.
    """
    if hidden_dims is None:
        hidden_dims = [512, 256, 128, 64]

    X_tr, X_val, y_tr, y_val = train_test_split(
        X_tda, y, test_size=0.2, stratify=y, random_state=42)

    scaler = StandardScaler().fit(X_tr)
    X_tr_n  = torch.FloatTensor(scaler.transform(X_tr)).to(device)
    X_val_n = torch.FloatTensor(scaler.transform(X_val)).to(device)
    y_tr_t  = torch.LongTensor(y_tr).to(device)
    y_val_t = torch.LongTensor(y_val).to(device)

    loader = DataLoader(TensorDataset(X_tr_n, y_tr_t),
                        batch_size=batch_size, shuffle=True)

    model = TopologicalSignatureLayer(
        X_tda.shape[1], hidden_dims, n_classes).to(device)

    criterion = nn.CrossEntropyLoss(
        weight=torch.FloatTensor(
            [1.0 / np.sum(y == c) for c in range(n_classes)]
        ).to(device)
    )
    optimizer = optim.AdamW(model.parameters(), lr=lr,
        weight_decay=1e-4)
    scheduler = optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=n_epochs)
    early_stop = EarlyStopping(patience=20)

    history = {'train_loss': [], 'val_loss': [], 'val_acc': []}
    best_acc = 0.0
    best_weights = None

    for epoch in range(n_epochs):
        model.train()
        epoch_loss = 0.0
        for xb, yb in loader:
            optimizer.zero_grad()
            loss = criterion(model(xb), yb)
            loss.backward()
            nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
            epoch_loss += loss.item() * len(xb)
        scheduler.step()

        model.eval()
        with torch.no_grad():
            val_out   = model(X_val_n)
            val_loss  = criterion(val_out, y_val_t).item()
            val_preds = val_out.argmax(dim=1).cpu().numpy()
            val_acc   = balanced_accuracy_score(y_val, val_preds)
```

```python

        history['train_loss'].append(epoch_loss / len(X_tr))
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)

        if val_acc > best_acc:
            best_acc = val_acc
            best_weights = {k: v.clone()
                            for k, v in model.state_dict().items()}

        early_stop.step(val_loss)
        if early_stop.should_stop:
            print(f"  Arret precoce a l'epoque {epoch+1}")
            break

        if (epoch + 1) % 20 == 0:
            print(f"  Ep {epoch+1:3d} | "
                  f"loss_train={epoch_loss/len(X_tr):.4f} | "
                  f"loss_val={val_loss:.4f} | "
                  f"bal_acc={val_acc*100:.1f}%")

    print(f"\nMeilleure balanced accuracy validation :
        {best_acc*100:.1f}%")
    model.load_state_dict(best_weights)
    return {'model': model, 'scaler': scaler, 'history': history,
            'best_val_acc': best_acc}
```

Listing 11: Couche de persistance differentiable (PyTorch)

Module code : **Module C3 — Analyse et visualisation des résultats**

Objectif : Produire des visualisations diagnostiques complètes : t-SNE des espaces de features, courbes d'apprentissage, matrices de confusion, et cartes topographiques EEG des features les plus discriminantes.

```python
from sklearn.manifold import TSNE
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np


CLASS_NAMES = ['Main G.', 'Main D.', 'Pieds', 'Langue']
PALETTE     = ['#2c7bb6', '#d7191c', '#1a9641', '#fdae61']


def plot_tsne_comparison(features: dict, y: np.ndarray,
                         reps: list = None) -> None:
    """
    t-SNE 2D des espaces de features pour plusieurs representations.
    Permet de visualiser la separabilite topologique des classes.
    """
    if reps is None:
        reps = ['betti', 'landscape', 'pi', 'spectral', 'all']
    reps = [r for r in reps if r in features]

    fig, axes = plt.subplots(1, len(reps),
                             figsize=(5 * len(reps), 5))
    if len(reps) == 1:
        axes = [axes]
```

```
26
27      for ax, rep in zip(axes, reps):
28          X = features[rep]
29          from sklearn.preprocessing import StandardScaler
30          X_s = StandardScaler().fit_transform(X)
31          # Reduction PCA preliminaire si trop haute dimension
32          if X_s.shape[1] > 50:
33              from sklearn.decomposition import PCA
34              X_s = PCA(n_components=50).fit_transform(X_s)
35          emb = TSNE(n_components=2, perplexity=30,
36                     random_state=0, n_iter=1000).fit_transform(X_s)
37          for c, (name, col) in enumerate(zip(CLASS_NAMES, PALETTE)):
38              idx = y == c
39              ax.scatter(emb[idx, 0], emb[idx, 1],
40                         c=col, label=name, s=25, alpha=0.7)
41          ax.set_title(f't-SNE : {rep}', fontsize=11)
42          ax.legend(fontsize=7, markerscale=1.5)
43          ax.axis('off')
44      plt.suptitle('Visualisation t-SNE des representations TDA vs
            spectral',
45                   fontsize=13, y=1.02)
46      plt.tight_layout()
47      plt.savefig('tsne_representations.pdf', dpi=150,
            bbox_inches='tight')
48      plt.show()
49
50
51  def plot_learning_curves(history: dict) -> None:
52      """Courbes d'apprentissage du reseau topologique."""
53      fig, axes = plt.subplots(1, 2, figsize=(11, 4))
54
55      ax = axes[0]
56      ax.plot(history['train_loss'], label='Train', color='steelblue',
            lw=2)
57      ax.plot(history['val_loss'],   label='Val',   color='tomato',
            lw=2)
58      ax.set_xlabel('Epoque')
59      ax.set_ylabel('Cross-entropy loss')
60      ax.set_title('Courbes de loss')
61      ax.legend()
62      ax.grid(alpha=0.3)
63
64      ax = axes[1]
65      ax.plot(np.array(history['val_acc']) * 100,
66              color='seagreen', lw=2)
67      ax.axhline(25, color='r', linestyle='--', alpha=0.5, label='Chance')
68      ax.set_xlabel('Epoque')
69      ax.set_ylabel('Balanced accuracy val (%)')
70      ax.set_title('Progression de l\'accuracy de validation')
71      ax.legend()
72      ax.grid(alpha=0.3)
73
74      plt.tight_layout()
75      plt.savefig('learning_curves_tda.pdf', dpi=150, bbox_inches='tight')
76      plt.show()
77
78
79  def plot_confusion_and_importance(model_result: dict,
80                                    X_test: np.ndarray,
81                                    y_test: np.ndarray,
82                                    feature_names: list = None) -> None:
83      """
```

```
    Matrice de confusion et importance des features (par permutation).
    """
    import torch
    model  = model_result['model']
    scaler = model_result['scaler']

    X_t = torch.FloatTensor(scaler.transform(X_test))
    with torch.no_grad():
        preds = model(X_t).argmax(dim=1).numpy()

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Matrice de confusion normalisee
    cm = confusion_matrix(y_test, preds, normalize='true')
    disp = ConfusionMatrixDisplay(cm, display_labels=CLASS_NAMES)
    disp.plot(ax=axes[0], colorbar=False, cmap='Blues',
        values_format='.2f')
    axes[0].set_title('Matrice de confusion normalisee')

    # Importance par permutation (top 20 features)
    from sklearn.inspection import permutation_importance
    from sklearn.pipeline import Pipeline
    from sklearn.preprocessing import StandardScaler
    from sklearn.svm import SVC

    # Re-ajuste un SVM pour avoir accces a permutation_importance
    pipe = Pipeline([('sc', StandardScaler()),
                     ('clf', SVC(kernel='rbf', C=10, gamma='scale'))])
    pipe.fit(X_test, y_test)  # sur test pour illustration uniquement
    result = permutation_importance(pipe, X_test, y_test,
                                    n_repeats=10, random_state=0,
                                    scoring='balanced_accuracy')
    imp = result.importances_mean
    top_idx = np.argsort(imp)[-20:][::-1]
    axes[1].barh(range(20),
                 imp[top_idx[::-1]], color='steelblue', alpha=0.8)
    if feature_names is not None:
        axes[1].set_yticks(range(20))
        axes[1].set_yticklabels([feature_names[i] for i in
            top_idx[::-1]],
                                fontsize=8)
    axes[1].set_xlabel('Importance (reduction d\'accuracy par
        permutation)')
    axes[1].set_title('Top 20 features TDA les plus discriminantes')
    axes[1].grid(axis='x', alpha=0.3)

    plt.tight_layout()
    plt.savefig('confusion_importance.pdf', dpi=150,
        bbox_inches='tight')
    plt.show()


#           Script principal : pipeline de bout en bout


def run_full_pipeline(subject_id: int = 1,
                      data_dir: str = './data') -> None:
    """
    Lance le pipeline complet pour un sujet BCI.
    Encha ne chargement, plongement de Takens, calcul des diagrammes,
    vectorisation, benchmark et entrainement du reseau.
    """
```

```python
    import time
    print(f"\n{'='*60}")
    print(f"Pipeline TDA-EEG -- Sujet {subject_id}")
    print(f"{'='*60}")

    # Chargement et plongement
    t0 = time.time()
    data = load_bci_data(subject_id, data_dir)
    clouds, y = preprocess_dataset(data, dim=3, delay=4, max_pts=200)
    print(f"Plongement de Takens : {len(clouds)} essais en "
          f"{time.time()-t0:.1f}s")

    # Diagrammes de persistance
    t1 = time.time()
    all_diags = compute_diagrams_parallel(clouds, max_dim=2, n_jobs=4)
    print(f"Diagrammes : {time.time()-t1:.1f}s")

    # Vectorisations
    features = build_full_feature_matrix(all_diags, resolution=50)
    features['spectral'] = build_spectral_matrix(data['X'],
        data['sfreq'])

    # Benchmark complet
    df_results, all_results = full_benchmark(features, y)
    df_stats = statistical_comparison(all_results, ref_rep='spectral')
    print("\nTests de Wilcoxon (vs baseline spectrale) :")
    print(df_stats.to_string(index=False))

    # Visualisations
    plot_benchmark(df_results)
    plot_tsne_comparison(features, y)

    # Reseau topologique sur les features hybrides
    X_hybrid = build_hybrid_features(features['all'],
        features['spectral'])
    net_result = train_topology_net(X_hybrid, y, n_classes=4,
                                    n_epochs=300, device='cpu')
    plot_learning_curves(net_result['history'])

    # Resume final
    best_tda = df_results.groupby('Representation')[
        'Balanced_Accuracy'].mean().max()
    print(f"\nResume :")
    print(f"  Meilleure accuracy TDA pure   : {best_tda:.1f}%")
    print(f"  Meilleure accuracy TDA+NN     : "
          f"{net_result['best_val_acc']*100:.1f}%")
    print(f"  Baseline spectrale            : "
          f"{df_results[df_results.Representation=='spectral']"
          f"['Balanced_Accuracy'].mean():.1f}%")
    print(f"  Chance (4 classes)            : 25.0%")


if __name__ == '__main__':
    run_full_pipeline(subject_id=1)
```

Listing 12: Visualisations diagnostiques du pipeline TDA-EEG

Résultats attendus

Résultats attendus du projet avancé.

1. **Validation du plongement de Takens** : les nuages de points issus des classes « main gauche » et « main droite » exhibent des topologies H_1 distinctes (cycles de dimensions différentes), conformément à la différence de rythme mu contra-latéral.
2. **Supériorité des paysages de persistance** : on s'attend à ce que les paysages de persistance (landscape) et les images de persistance (pi) surpassent les courbes de Betti brutes, car ils intègrent l'information de persistance (robustesse) dans la vectorisation.
3. **Features hybrides TDA + spectral** : le vecteur hybride doit systématiquement surpasser les deux sources prises séparément ($+3$ à $+5$ points d'accuracy), ce qui démontre la *complémentarité* de l'information topologique et spectrale.
4. **Significance statistique** : le test de Wilcoxon (correction de Bonferroni) devrait donner $p < 0.05$ pour la représentation hybride vs baseline spectrale, sur au moins 6 des 9 sujets.
5. **Visualisation t-SNE** : dans l'espace des paysages de persistance, les 4 classes forment des nuages plus séparés que dans l'espace spectral, illustrant visuellement la valeur ajoutée de la TDA.
6. **Réseau topologique** : le MLP entraîné sur les features TDA avec arret précoce et scheduler cosine doit atteindre une balanced accuracy $\geq 65\%$, en 5 à 10 fois moins de paramètres qu'un EEGNet ou ShallowConvNet standard.

Ressources : Phase 4b — Références

Perea & Harer, *Sliding windows and persistence* (2015) — fondement théorique du plongement de Takens + TDA ; Hofer, Kwitt & Niethammer, *Learning representations of persistence barcodes* (NeurIPS 2019) ; Gabrielsson *et al.*, *A topology layer for machine learning* (ICML 2020) ; Brunner *et al.*, *BCI Competition 2008 – Graz data set A* (2008) ; Adams *et al.*, *Persistence images* (JMLR 2017) ; Bubenik (2015) pour les paysages de persistance.

4.7 Phase 5 — Rédaction et approfondissements (semaines 19–24)

Phase 5 — Rédaction & approfondissements

Objectif de la phase : Rédiger le mémoire final, consolider les démonstrations, et explorer des extensions avancées selon les intérêts.

Plan du mémoire.

1. Introduction et motivation : le problème de l'extraction de forme, historique et enjeux de la TDA.
2. Homologie simpliciale : complexes de Čech et Vietoris-Rips, opérateur bord, groupes d'homologie, nombres de Betti.
3. Modules de persistance et décomposition : filtrations, théorème de structure, barcodes et diagrammes de persistance.
4. Le théorème de stabilité : distance de bottleneck, entrelacement, démonstration complète, version géométrique.
5. Implémentation et résultats numériques : pipeline Python, expériences de robustesse, classification de nuages de points.
6. Conclusion et perspectives : connexions avec le machine learning, reconstruction de variétés, homologie persistante multivariée.

Extensions optionnelles.

- **Paysages de persistance** (Bubenik, 2015) : plongement des diagrammes dans un espace de Banach séparable, permettant le calcul de moyennes et d'estimateurs statistiques robustes.
- **Homologie persistante multiparamétrique** : remplacement de la filtration à un paramètre par une filtration à deux paramètres (par exemple (ε, δ) pour des données bruitées avec densité variable). La décomposition n'est plus aussi simple, mais des invariants partiels (nombres de Betti persistants) restent calculables.
- **Théorème de reconstruction de variétés** (Niyogi-Smale-Weinberger, 2008) : si le nuage de points est un échantillon ε-dense d'une variété M lisse, le complexe de Čech à l'échelle appropriée a le même type d'homotopie que M.
- **Couches de persistance** (Hofer et al., 2019) : intégration différentiable des diagrammes de persistance dans des réseaux de neurones, ouvrant la voie à l'apprentissage de représentations topologiques.
- **Homologie persistante de Morse** : lien entre le barcode d'une fonction de Morse et les indices critiques, via le complexe de Morse-Witten.

Ressources : Phase 5 — Extensions

Bubenik (2015), *Statistical topological data analysis using persistence landscapes* ; Niyogi, Smale & Weinberger (2008) ; Hofer, Kwitt & Niethammer (2019) ; Oudot (2015), chapitres 5–7 ; Lesnick (2015) pour la théorie multiparamétrique.

5 Tableau récapitulatif

Phase	Contenu principal	Objectifs clés	Ressources	Durée
Phase 1 (S. 1–4)	Homologie simpliciale, complexes de Čech et Vietoris-Rips	Calcul de H_k, β_k, invariance homotopique, Mayer-Vietoris	Hatcher (2002), Munkres (1984)	4 sem.
Phase 2 (S. 5–8)	Filtrations, modules de persistance, décomposition	Théorème de structure, barcodes, algorithme de réduction	Zomorodian & Carlsson (2005), Edelsbrunner & Harer (2010)	4 sem.
Phase 3 (S. 9–13)	Distance bottleneck, entrelacement, stabilité	Lemme d'entrelacement, théorème de stabilité, version géométrique	Cohen-Steiner et al. (2007), Chazal et al. (2016)	5 sem.
Phase 4(S. 14–18)	Code : pipeline TDA, classification	Barcodes Python, images de persistance, SVM topologique	gudhi, ripser, persim, scikit-learn	5 sem.
Phase 5(S. 19–24)	Rédaction, approfondissements	Mémoire, paysages de persistance, reconstruction de variétés	Bubenik (2015), Oudot (2015)	6 sem.

Tableau 1 : Récapitulatif des cinq phases du projet (durée totale : 4 à 6 mois).

6 Conclusion

L'homologie persistante représente l'une des avancées les plus élégantes et les plus utiles des mathématiques appliquées des vingt dernières années. En combinant la rigueur de la topologie algébrique classique avec les impératifs de robustesse et d'efficacité algorithmique, elle fournit un outil d'analyse des données véritablement nouveau, capable de détecter des structures globales invisibles aux méthodes métriques ou linéaires.

Le **théorème de stabilité** est le résultat qui transforme cette théorie en outil fiable : perturber les données de ε en norme infinie ne perturbe les diagrammes de persistance que de ε en distance bottleneck. Ce théorème est à la fois profond mathématiquement — sa preuve mobilise la notion d'entrelacement de foncteurs et la géométrie des espaces métriques compacts — et fondamental en pratique, puisqu'il justifie l'usage de ces méthodes sur des données réelles, inévitablement bruitées.

Le **projet de code** — pipeline Python de bout en bout, de la génération de nuages de points à la classification par features topologiques — constitue un livrable concret et directement valorisable, aussi bien dans un contexte académique (mémoire, publication) que professionnel

(bioinformatique, traitement d'images médicales, neurosciences computationnelles, science des matériaux).

Contact et prolongements. Ce projet peut naturellement se prolonger vers une thèse en mathématiques appliquées ou en apprentissage automatique. Les directions les plus prometteuses sont : la TDA multiparamétrique (Lesnick, 2015), les architectures neuronales topologiques (Hofer, 2019), et les connexions avec la théorie des représentations des carquois. Sur le plan applicatif, l'usage de la TDA en biologie structurale (repliement des protéines) et en analyse des réseaux de neurones (connectomique) offre des terrains de recherche particulièrement actifs.