**LISTE DES MODULES À IMPLÉMENTER POUR LE PROJET FPS**

**Note méthodologique (Fidélité FPS) :**

Toutes les implémentations proposées dans ce dépôt (pour les termes Sᵢ(t), Eₙ(t), Oₙ(t), les formes dynamiques, les critères de raffinement, etc.) sont des hypothèses initiales : elles servent uniquement à rendre possible la falsification expérimentale dès la phase 1.

Aucune de ces définitions n’est considérée comme définitive ou universelle.

• Elles doivent être modifiées, raffinées ou remplacées si l’expérience, l’observation des runs ou le développement théorique l’exigent.

• Les modes statique/dynamique sont tous activables via la config, et peuvent être étendus ou ajustés à tout moment.

• La plasticité (la possibilité de revisiter n’importe quel module ou critère, même tardivement) est une part intégrante de la méthode FPS.

**Toute contribution ou usage futur du code doit préserver cette ouverture :**

si une hypothèse d’implémentation ne correspond plus à la dynamique réellement observée, elle doit être falsifiée et adaptée, conformément à la feuille de route FPS.

**Module 1: init.py - Initialisation**

**À implémenter dans ce module :**

* Fonction init\_strates() qui initialise toutes les strates avec :
  + Paramètres A₀, f₀ₙ, φₙ, αₙ, βₙ, k, x₀ depuis config.json
  + Matrice de poids w\_{ni} pour connexions inter-strates
  + Gestion de la seed aléatoire (random.seed, np.random.seed)
  + Initialisation des états En(t=0) = A₀ et On(t=0) = 0
* Fonction load\_config() pour charger config.json
* Fonction setup\_logging() pour préparer les fichiers de log
* Écriture automatique dans seeds.txt de chaque seed utilisée
* Création structure de données pour stocker l'état du système

**Module 2: dynamics.py - Calculs des termes FPS**

NOTE FPS – Plasticité méthodologique :

La définition actuelle de [Sᵢ(t)]/[Eₙ(t)]/[Oₙ(t)] (ainsi que de φₙ(t), θ(t), η(t), μₙ(t) et les latences) est une hypothèse de phase 1, appelée à être falsifiée/raffinée selon la feuille de route FPS. Toute version ultérieure doit préserver la plasticité du module.

**À implémenter dans ce module :**

* compute\_In(t, mode="constant", params={}) - Input contextuel avec modes :
  + Mode "constant" : valeur fixe
  + Mode "choc" : impulsion à t0
  + Mode "rampe" : augmentation linéaire
  + Mode "sinus" : oscillation périodique
  + Mode "uniform" : U[0,1]
* compute\_sigma(x, k, x0) - Fonction sigmoïde 1/(1+e^(-k(x-x0)))
* compute\_An(t, A0, In\_t, k, x0) - Amplitude adaptative A₀·σ(Iₙ(t))
* compute\_S\_i(t, n, history) - Signal des autres strates :
  + if t == 0: return 0else: return history[t-1]['S'] - history[t-1]['O'][n]
* compute\_delta\_fn(t, alpha\_n, w\_ni, S\_i) - Modulation αₙ·Σᵢw\_{ni}·Sᵢ(t)
* compute\_fn(t, f0n, delta\_fn) - Fréquence f₀ₙ + Δfₙ(t)
* compute\_phi\_n(t, phi\_n\_prev, mode="static") - Phase avec modes :
  + Mode "static" : φₙ constant
  + Mode "dynamic" : évolution à définir après phase 1
* compute\_gamma(t, mode="static", T=None) - Latence expressive :
  + Mode "static" : retourne 1.0
  + Mode "dynamic" : 1/(1+e^{-2(t-T/2)})
* compute\_gamma\_n(t, n, mode="static", T=None) - Latence par strate :
  + Mode "static" : retourne 1.0
  + Mode "dynamic" : 1/(1+e^{-kₙ(t-t₀ₙ)}) avec kₙ=2.0, t₀ₙ=T/2
* compute\_On(t, n, An, fn, phi\_n, gamma\_n) - Sortie observée :
  + # Contribution de la strate n au signal global. return An \* np.sin(2\*np.pi\*fn\*t + phi\_n) \* gamma\_n
* compute\_En(t, n, history, A0) - Sortie attendue (harmonique cible) :
  + # Attracteur basé sur le nombre d’or. phi = 1.618if len(history) > 0: return phi \* history[-1]['O'][n]else: return A0 # Valeur initiale = amplitude de base
* compute\_r(t, phi, epsilon, omega, theta) - Ratio spiralé φ+ε·sin(2π·ω·t+θ)
* compute\_C(t, phi\_n\_array) - Coefficient (1/N)·Σcos(φₙ₊₁-φₙ)
* compute\_A(t, delta\_fn\_array) - Modulation moyenne (1/N)·ΣΔfₙ(t)
* compute\_A\_spiral(t, C\_t, A\_t) - Amplitude harmonisée C(t)·A(t)
* compute\_Fn(t, beta\_n, On\_t, En\_t, gamma\_t) - Feedback βₙ·(Oₙ(t)-Eₙ(t))·γ(t)
* compute\_S(t, An\_array, fn\_array, phi\_n\_array, mode="simple") :
  + Mode "simple" : Σₙ Aₙ(t)·sin(2π·fₙ(t)·t+φₙ(t))
  + Mode "extended" : avec γₙ(t) et G(Eₙ(t)-Oₙ(t))
* compute\_E(t, signal\_array) - Amplitude max : maxₙ|S(t)|
* compute\_L(t, signal\_array) - Latence : argmaxₙ|S(t)|

**Module 3: regulation.py - Fonctions de régulation**

NOTE FPS – Plasticité méthodologique :

Les formes de G(x), envₙ(x,t) sont à adapter selon l’expérience. Ne jamais considérer la version présente comme définitive.

**À implémenter dans ce module :**

* compute\_G(x, archetype="tanh", params={}) avec 4 archétypes :
  + "tanh" : tanh(λx) avec λ dans params
  + "sinc" : sin(x)/x
  + "resonance" : sin(βx)·exp(-αx²) avec α,β dans params
  + "adaptive" : forme à définir selon contexte
* compute\_Gn(x, t, An\_t, fn\_t, mu\_n\_t, env\_n) - Réponse locale :
  + Aₙ(t)·sinc[fₙ(t)·(x-μₙ(t))]·envₙ(x,t)
* compute\_G\_temporal(x, t, eta\_t, theta\_t) - Version temporelle :
  + η(t)·sin(θ(t)·x)
* compute\_env\_n(x, t, mode="static", sigma\_n=0.1, mu\_n=0.0, T=None) :
  + Mode "static" : gaussienne avec σₙ=0.1 constant
  + Mode "dynamic" : σₙ(t)=0.1+0.05·sin(2π·t/T)
* compute\_mu\_n(t, mode="static") - Centre enveloppe :
  + Mode "static" : 0.0
  + Mode "dynamic" : à définir phase 2
* compute\_sigma\_n(t, mode="static", T=None) - Écart-type :
  + Mode "static" : 0.1
  + Mode "dynamic" : 0.1+0.05·sin(2π·t/T)

**Module 4: metrics.py - Calcul des métriques**

NOTE FPS – Plasticité méthodologique :

Les métriques (effort, entropy, etc.) sont ajustables : toute modification ou alternative testée doit être documentée et laissée ouverte dans la config.

**À implémenter dans ce module :**

* compute\_cpu\_step(start\_time, end\_time, N) - Temps CPU : (end-start)/N
* compute\_effort(delta\_An\_array, delta\_fn\_array, delta\_gamma\_n\_array, An\_max, fn\_max, gamma\_max) :
  + # Version normalisée pour cohérence dimensionnelle. effort = Σₙ [|ΔAₙ|/An\_max + |Δfₙ|/fn\_max + |Δγₙ|/gamma\_max]
* compute\_effort\_status(effort\_t, effort\_history, threshold) :
  + Retourne "stable", "transitoire" ou "chronique"
* compute\_variance\_d2S(S\_history, dt) - Variance d²S/dt² (fluidité)
* compute\_entropy\_S(S\_t, sampling\_rate) - Entropie spectrale (innovation) :
  + # Entropie de Shannon du spectre de puissance normalisé. freqs, psd = signal.periodogram(S\_t, sampling\_rate)psd\_norm = psd / np.sum(psd)return -np.sum(psd\_norm \* np.log(psd\_norm + 1e-15))
* compute\_mean\_abs\_error(En\_array, On\_array) - mean(|Eₙ(t)-Oₙ(t)|)
* compute\_mean\_high\_effort(effort\_history, percentile=80) - Moyenne haute
* compute\_d\_effort\_dt(effort\_history, dt) - Dérivée temporelle
* compute\_t\_retour(S\_history, t\_choc, dt, threshold=0.95) :
  + # Temps pour revenir à 95% de l'état pré-choc # État pré-choc = moyenne de |S(t)| sur fenêtre [t\_choc-10\*dt, t\_choc]. pre\_shock\_window = S\_history[max(0, t\_choc-10):t\_choc] etat\_pre\_choc = np.mean(np.abs(pre\_shock\_window))# Chercher quand |mean(|S(t)|) - etat\_pre\_choc| < 0.05 \* etat\_pre\_choc
* compute\_max\_median\_ratio(S\_history) - max(S(t))/médiane(S(t))
* check\_thresholds(metrics\_dict, thresholds\_dict) - Vérification seuils :
  + # Note: Seuils initiaux théoriques, à ajuster après 5 runs de calibration
* log\_metrics(t, metrics\_dict, csv\_writer, hdf5\_file=None) - Export

**Module 5: simulate.py - Boucle principale**

**À implémenter dans ce module :**

* run\_simulation(config, mode="FPS") avec modes :
  + "FPS" : simulation complète
  + "Kuramoto" : oscillateurs contrôle
  + "neutral" : sans feedback ni spiral
* Boucle temporelle principale qui pour chaque pas t :
  + Calcule tous les Iₙ(t) selon stratégie de perturbation
  + Calcule S\_i(t) en utilisant l'historique (résout circularité)
  + Met à jour tous les termes (An, fn, φn, etc.) via dynamics.py
  + Calcule En(t) et On(t) avec nos définitions claires
  + Applique la régulation G(x) via regulation.py
  + Calcule S(t) global
  + Mesure cpu\_step avec time.perf\_counter()
  + Calcule toutes les métriques via metrics.py
  + Log dans CSV (et HDF5 si N>10)
* Gestion des états précédents pour calcul des deltas
* Support des différents modes statique/dynamique selon config
* Détection "mode alerte" si écart >3σ sur critères non-déclencheurs
* Écriture fichiers séparés A\_n\_{id}.csv, f\_n\_{id}.csv si N>10

**Module 6: validate\_config.py - Validation configuration**

**À implémenter dans ce module :**

* validate\_config(config\_path) qui vérifie :
  + Présence des sections : system, strates, spiral, to\_calibrate
  + N>0, T>0, seed définie
  + len(strates) == N
  + Chaque strate a : A0, f0, alpha, beta, k, x0, w
  + Paramètres spiral : phi, epsilon, omega, theta
  + Seuils dans to\_calibrate : variance\_d2S, entropy\_S, gamma\_n, env\_n, sigma\_n
  + Note: "Seuils théoriques initiaux, à modifier après runs de calibration"
* generate\_default\_config(N, T) - Génère config par défaut
* update\_config\_threshold(config, criterion, new\_value, reason) :
  + Met à jour un seuil et log la modification
* Validation des modes de perturbation (type, t0, amplitude)
* Vérification cohérence des poids w\_{ni}
* Si exploration.detect\_fractal\_patterns == True, vérifier :
  + exploration.metrics est une liste non vide
  + exploration.window\_sizes est une liste non vide
  + 0 < exploration.fractal\_threshold < 1
  + Message clair dans le log si config fractale valide (“Fractal motif detection config: OK”)

**Module 7: analyze.py - Analyse et raffinement**

NOTE FPS – Plasticité méthodologique :

Le processus de raffinements/ajustements de seuils, critères, et dynamiques doit être itératif et réversible.

**À implémenter dans ce module :**

* analyze\_criteria\_and\_refine(logs\_batch, config) qui :
  + Analyse 5 runs d'un batch
  + Vérifie franchissement des seuils pour chaque critère
  + Déclenche raffinements si seuil franchi sur >50% des runs
* Actions de raffinement par critère :
  + refine\_fluidity(config) : ajuste γₙ(t), envₙ(x,t)
  + refine\_stability(config) : ajuste σ(x), αₙ
  + refine\_resilience(config) : ajuste αₙ, βₙ
  + refine\_innovation(config) : ajuste θ(t), η(t), μₙ(t)
  + refine\_regulation(config) : ajuste βₙ, G(x)
  + refine\_cpu(config) : optimise complexité
  + refine\_chronic\_effort(config) : ajuste αₙ, μₙ(t), σₙ(t)
  + refine\_transient\_effort(config) : ajuste w\_{ni}
* log\_refinement(changelog\_path, date, run\_id, criterion, old\_value, new\_value, reason)
* compute\_correlation\_effort\_cpu(effort\_history, cpu\_history)
* export\_threshold\_journal(threshold\_history, output\_path)

**Module 8: perturbations.py - Gestion des perturbations**

**À implémenter dans ce module :**

* generate\_perturbation(t, config) avec types :
  + "choc" : impulsion à t0 avec amplitude
  + "rampe" : croissance linéaire de 0 à amplitude
  + "sinus" : amplitude·sin(2π·freq·t)
  + "bruit" : random.uniform(-amplitude, amplitude)
  + "none" : pas de perturbation
* apply\_perturbation\_to\_In(In\_array, perturbation\_value)
* generate\_perturbation\_sequence(T, dt, perturbation\_configs) :
  + Pour combiner plusieurs perturbations
* Paramètres de perturbation : type, t0, amplitude, durée, fréquence

**Module 9: visualize.py - Visualisation**

**À implémenter dans ce module :**

* plot\_signal\_evolution(t\_array, S\_array, title="") - Évolution S(t)
* plot\_strata\_comparison(t\_array, An\_arrays, fn\_arrays) - Par strate
* plot\_phase\_diagram(phi\_n\_arrays) - Diagramme de phase
* plot\_metrics\_dashboard(metrics\_history) - Tableau de bord complet
* create\_empirical\_grid(scores\_dict) avec icônes/couleurs :
  + Note 1 : ✖ rouge - Rupture/Chaotique
  + Note 2 : ▲ orange - Instable
  + Note 3 : ● jaune - Fonctionnel
  + Note 4 : ✔ vert - Harmonieux
  + Note 5 : ∞ bleu doré - FPS-idéal
* animate\_spiral\_evolution(data, output\_path) - Animation temps réel
* plot\_fps\_vs\_kuramoto(fps\_data, kuramoto\_data) - Comparaison
* generate\_correlation\_matrix(criteria\_terms\_mapping) - Matrice critère↔termes
* export\_html\_report(all\_data, output\_path) - Rapport HTML complet

**Module 10: test\_fps.py - Tests unitaires**

**À implémenter dans ce module :**

* Tests pour chaque fonction compute avec modes statique/dynamique :
  + test\_compute\_gamma\_static() : assert == 1.0
  + test\_compute\_gamma\_dynamic() : assert valeur sigmoïde correcte
  + test\_compute\_gamma\_n\_static() et test\_compute\_gamma\_n\_dynamic()
  + test\_compute\_env\_n\_static() et test\_compute\_env\_n\_dynamic()
  + test\_compute\_sigma() : vérifier formule sigmoïde
  + test\_compute\_An(), test\_compute\_fn(), etc.
  + test\_compute\_S\_i() : vérifier résolution circularité
  + test\_compute\_En\_On() : vérifier cohérence
* Tests de validation :
  + test\_validate\_config\_valid() et test\_validate\_config\_invalid()
* Tests de perturbations :
  + test\_perturbation\_choc(), test\_perturbation\_rampe(), etc.
* Tests d'intégration :
  + test\_full\_simulation\_fps(), test\_full\_simulation\_kuramoto()
* Tests de performance :
  + test\_cpu\_step\_bounds() : vérifier temps < seuil
* Tests de régulation G(x) pour chaque archétype

**Module 11: utils.py - Utilitaires**

**À implémenter dans ce module :**

* merge\_logs(log\_files, output\_path) - Fusion CSV multiples
* load\_simulation\_state(checkpoint\_path) - Charger état sauvegardé
* save\_simulation\_state(state, checkpoint\_path) - Sauvegarder état
* replay\_from\_logs(csv\_path) - Rejouer simulation depuis logs
* compare\_runs(run1\_path, run2\_path, metrics) - Comparateur
* batch\_runner(configs\_list, parallel=True) - Runs parallèles
* export\_to\_hdf5(data\_dict, hdf5\_path) - Export gros volumes
* generate\_run\_id() - ID unique pour chaque run
* setup\_directories() - Créer structure dossiers logs/results

**Module 12: kuramoto.py - Oscillateurs de Kuramoto (contrôle)**

**À implémenter dans ce module :**

* kuramoto\_step(phases, frequencies, K, N, dt) :
  + dφᵢ/dt = ωᵢ + (K/N)·Σⱼsin(φⱼ-φᵢ)
* compute\_kuramoto\_order(phases) : (1/N)·Σₙexp(iφₙ)
* run\_kuramoto\_simulation(config) - Version complète Kuramoto
* Paramètres : K=0.5, N=20, ωᵢ~U[0,1]
* Mêmes métriques loguées que FPS pour comparaison

**Module 13: explore.py – Exploration, détection d’émergence et d’anomalie FPS**

NOTE FPS – Plasticité méthodologique :

Ce module capture les phénomènes non anticipés ou émergents.

Il doit rester ouvert, extensible, et permettre à chaque contributeur d’ajouter ses propres détecteurs ou analyses.

Toute émergence ou anomalie détectée est loguée, traçable (avec seed/config), et sujette à reproductibilité.

Les seuils, méthodes et types de détection sont adaptables après les runs initiaux et l’analyse des premières émergences.

**À implémenter dans ce module :**

* detect\_anomalies(data, metrics, threshold=3.0, min\_duration=3)

Détecte les séquences persistantes de valeurs hors-norme sur chaque métrique (au-delà de threshold \* σ, durée minimale min\_duration).

Retourne une list[dict] avec format structuré (event\_type, t\_start, t\_end, metric, value, severity).

* detect\_spiral\_bifurcations(data, phase\_metric='C(t)', threshold=np.pi)

Analyse les changements de phase/bifurcations dans la métrique d’accord spiralé.

* detect\_harmonic\_emergence(data, signal\_metric='S(t)', n\_harmonics=5, window=100, step=10)

Utilise une FFT glissante pour détecter l’apparition de nouvelles harmoniques dans le signal global.

* explore\_phase\_space(data, metric='S(t)', window=50, min\_diagonal\_length=5)

Recurrence plot : cherche les motifs récurrents/cycles attracteurs dans l’espace de phase. Retourne des événements structurés.

* detect\_fractal\_patterns(data, metrics=['S(t)', 'C(t)', 'effort(t)'], window\_sizes=[1, 10, 100], threshold=0.8) Analyse multi-échelles (recurrence plot, corrélation glissante, fractal dimension, etc.) sur les signaux S(t), C(t), effort(t), etc.
  + Détecte et logue automatiquement :

• périodes de motifs auto-similaires (pattern récurrent)

• score de similarité fractale (> threshold du config)

• échelle (fenêtre d’observation), t\_start, t\_end

• Exporte dans fractal\_events\_<run\_id>.csv

• Section “Motifs fractals détectés” générée dans exploration\_report\_<run\_id>.md

* + Paramètres contrôlés par config.json :

• window\_sizes (échelles testées)

• fractal\_threshold

• metrics à surveiller

* format\_value\_for\_csv(value)

Assure l’export correct de valeurs complexes dans les logs (array/list → JSON, float/int → str…).

* Fonction log\_fractal\_events(events, csv\_path)

Mise à jour automatique du rapport d’exploration (section “Motifs fractals détectés”)

* log\_events(events, csv\_path)

Écrit le log CSV des émergences, colonnes : event\_type, t\_start, t\_end, metric, value, severity.

* run\_exploration(run\_data\_path, output\_dir, config=None)

Orchestration complète :

* + Charge les logs du run (CSV/HDF5)
  + Lance tous les détecteurs sur les métriques définies
  + Agrège les événements et les exporte dans emergence\_events\_<run\_id>.csv
  + Génère un rapport Markdown détaillé
* generate\_report(events, report\_path)

Génère un rapport lisible :

* Titre, date, total d’événements
* Regroupement par type et sévérité
* Détails sur les cycles/bifurcations majeurs (top 5)
* Pour chaque événement : t\_start-t\_end, metric, valeur(s), sévérité

**Inputs :**

* Fichiers CSV/HDF5 standards FPS (1 ligne par pas de temps, colonnes : t, S(t), C(t), A\_spiral(t), effort(t), etc.)
* Configurations des seuils, métriques à surveiller

**Outputs :**

* emergence\_events\_<run\_id>.csv : toutes les émergences et anomalies loguées, prêtes pour post-analyse ou visualisation
* exploration\_report\_<run\_id>.md : rapport synthétique et lisible
* (optionnel) figures de recurrence plot, FFT, phase, pour vérification visuelle

**Intégration/Utilisation :**

* Appelé automatiquement après chaque run ou batch de simulation, ou en exploration “offline” de logs existants.
* Compatible avec la gestion des seeds/config pour assurer la reproductibilité.
* Peut tourner en tâche indépendante ou en pipeline parallèle à l’analyse stricte.

**ÉLÉMENTS RESTANTS SANS MODULE SPÉCIFIQUE**

**Fichiers de configuration à créer :**

* config.json - Structure complète avec sections system/strates/spiral*/*exploration/to\_calibrate
  + Inclure note "Seuils initiaux théoriques, à ajuster après calibration" :
    - NOTE FPS – Plasticité méthodologique : Les paramètres et formules listés sont issus de la phase 1 et sont destinés à évoluer. Rien n’est figé. Voir la feuille de route FPS.
* seeds.txt - Log automatique des graines (géré par init.py mais fichier séparé)
* changelog.txt - Format : "[Date] | RunID | Métrique=valeur | Action | seed="

**Notebooks à créer :**

* fps\_main.ipynb - Notebook d'exploration interactive

**Documentation à maintenir :**

* Matrice de correspondance critère ↔ termes FPS (dans docs/)
* Journal des seuils avec historique (généré par analyze.py)
* README avec exemples d'utilisation et clarifications :
  + État pré-choc = moyenne |S(t)| sur 10\*dt avant perturbation
  + Seuils initiaux basés sur théorie, à ajuster empiriquement

**Paramètres à définir après phase 1 :**

* Forme dynamique de φₙ(t) si nécessaire
* Formes de θ(t) et η(t) pour G(x,t) adaptatif
* Dynamique de μₙ(t) pour phase 2

**Processus à documenter :**

* Protocole des 5 runs initiaux de calibration
* Workflow d'analyse batch et raffinement
* Procédure de modification des seuils

Cette organisation en modules couvre l'intégralité des éléments mentionnés dans la feuille de route FPS, avec une structure claire et modulaire facilitant l'implémentation et les tests.

Les implémentations proposées pour Sᵢ(t), Eₙ(t), Oₙ(t), etc., sont des choix de phase 1, à falsifier et raffiner si besoin.

Les modes statique/dynamique sont tous testables et activables via la config, et rien n’interdit d’en ajouter ou de modifier une formule si la théorie ou l’expérience le justifie.

La plasticité et la réouverture de chaque module sont une partie de la méthode FPS : tout module peut être revisité si l’observation ou le test l’exige.