**Dictionnaire mathématique de la FPS – via Chapitre 4**

La poésie inspire la métrique, la structure garantit la critique et la réinvention, la spirale relie les deux

|  |  |  |
| --- | --- | --- |
| **Terme** |  | **Définition** |
| S(t) |  | Signal global du système, résultant de la superposition des strates. |
| Aₙ(t) |  | Amplitude contextuelle d’une strate, régulée par une fonction sigmoïde selon une pression contextuelle. |
| Iₙ(t) |  | Input contextuel : densité d’information, stress, attention, etc. |
| σ(x) |  | Fonction d’adaptation douce (sigmoïde/tanh) qui régule l’impact du contexte. |
| A₀ |  | Amplitude de base propre à la strate. |
| k |  | Sensibilité de la strate. |
| x₀ |  | Seuil de basculement. |
| fₙ(t) |  | Fréquence propre modulée de la strate. |
| f₀ₙ |  | Fréquence de base de la strate. |
| Δfₙ(t) |  | Modulation adaptative de la fréquence. |
| αₙ |  | Souplesse d’adaptation d’une strate. |
| w\_{ni} |  | Poids de connexion vers une autre strate. |
| S\_i(t) |  | Signal d’autres strates (exploratoire : par convention, S\_i(0) = 0, et pour t > 0 : S\_i(t) = Σ\_{j≠i} O\_j(t-1)) |
| A(t) |  | Modulation moyenne du système, somme des Δfₙ(t). |
| C(t) |  | Coefficient d’accord spiralé basé sur la phase. |
| A\_spiral(t) |  | Amplitude harmonisée globale : A(t) × C(t). |
| Fₙ(t) |  | Feedback vivant d’une strate. |
| βₙ |  | Plasticité d’une strate. |
| Oₙ(t) |  | Sortie actuelle (exploratoire = An(t) \* sin(2πfn(t)t + φn(t)) \* γn(t)) |
| Eₙ(t) |  | Sortie attendue / harmonique (exploratoire = φ \* On(t-dt) où φ est le nombre d’or) |
| γ(t) |  | Latence expressive. |
| fₙ₊₁(t) |  | Fréquence de la strate suivante dans la spirale. |
| r(t) |  | Ratio spiralé non constant. |
| φ |  | Nombre d’or. |
| ε |  | Amplitude de variation harmonique. |
| ω |  | Fréquence lente de modulation spiralée. |
| θ |  | Phase initiale du spiral. |
| E(t) |  | Amplitude maximale globale. |
| L(t) |  | Latence maximale de variation d’une strate. |
| φₙ |  | Phase de la strate n. |
| γₙ(t) |  | Latence expressive d’une strate spécifique. |
| G(x) |  | Fonction spiralée de régulation adaptative. |
| G(x, t) |  | Version temporelle de G. |
| Gₙ(x, t) |  | Réponse harmonique locale d’une strate. |
| η(t) |  | Amplitude contextuelle dans G(x, t). |
| θ(t) |  | Fréquence adaptative dans G(x, t). |
| μₙ(t) |  | Décalage / recentrage adaptatif. |
| σₙ(t) |  | Écart-type de l’enveloppe. |
| σ(x) |  | Désigne une sigmoïde ou une tanh selon le mode. |
| envₙ(x,t) |  | Enveloppe adaptative (gaussienne ou sigmoïde). |

**Tableau structuré des paramètres et métriques (v1.1) – FPS**

Pour chaque paramètre mathématique ci-dessus, voir le tableau structuré ci-après pour toutes les valeurs, dépendances, colonnes de log et hypothèses à valider.

Ce tableau fait foi pour toutes les itérations, simulations et ajustements du projet FPS.

| **Paramètre** | **Valeur init.** | **Hypothèse / Forme** | **Méthode de validation** | **Priorité** | **Dépendances** | **Colonnes log associées** |
| --- | --- | --- | --- | --- | --- | --- |
| A₀ | 1.0 | Amplitude de base | Analyse du signal / runs initiaux | Critique | - | config, Aₙ(t) |
| k | 2.0 | Sensibilité sigmoïde | Calibration itérative | Critique | - | config |
| x₀ | 0.5 | Seuil sigmoïde | Calibration runs initiaux | Critique | - | config |
| αₙ | 0.5 | Souplesse d’adaptation | Test dynamique, résilience | Critique | S\_i(t), w\_{ni} | config |
| βₙ | 1.0 | Gain de feedback | Validation boucle feedback | Critique | Oₙ(t), Eₙ(t) | config |
| w\_{ni} | 0.1 | Poids de connexion | Calibration croisée | Secondaire | S\_i(t) | config |
| φ | 1.618 | Nombre d’or | Valeur canonique | Critique | - | config |
| ε | 0.05 | Amplitude harmonique | Runs initiaux / test stabilité | Secondaire | - | config |
| ω | 0.1 | Fréquence spiralée | Test adaptation | Secondaire | t | config |
| θ | 0.0 | Phase initiale | Choix arbitraire | Secondaire | t | config |
| λ, α, β (G(x)) | variable | Paramètres régulation G(x) | Runs comparatifs, test stabilité | Secondaire | G(x) | config |
| σ(x) | sigmoïde | 1/(1+e^(−k(x−x₀))) | Vérification code, runs initiaux | Critique | k, x₀, Iₙ(t) | — |
| Iₙ(t) | [0,1] | Input contexte | Génération/empirique, test choc | Critique | - | logs optionnels |
| Aₙ(t) | — | A₀·σ(Iₙ(t)) | Empirique, log complet | Critique | Iₙ(t), σ(x) | A\_mean(t), logs stratés |
| f₀ₙ | 1.0 | Fréquence de base | Fixe par strate | Critique | - | config |
| Δfₙ(t) | 0.0 | Modulation adaptative | Empirique, validation runs | Critique | αₙ, w\_{ni}, S\_i(t) | f\_mean(t), logs stratés |
| fₙ(t) | f₀ₙ + Δfₙ(t) | Fréquence effective | Empirique, log complet | Critique | f₀ₙ, Δfₙ(t) | f\_mean(t), logs stratés |
| φₙ(t) | 0.0 | Forme temporaire ; **évolution à définir après phase 1 si besoin de dynamique avancée.** | Empirique, à définir si besoin d’évolution dynamique après runs initiaux | Secondaire | t | — |
| γ(t) | 1.0 | Forme temporaire ; **Statique (=1.0) ET dynamique (1/(1+e^{-2(t-T/2)}) avec k=2.0, t₀=T/2)**, à comparer. | Calibration via runs initiaux (N=5, T=20, Iₙ(t) ~ U[0,1], seed fixée) | Moyenne | t, spiralisation | logs optionnels |
| γₙ(t) | 1.0 | Forme temporaire : **Statique (= 1.0) ET dynamique (= 1/(1+e^{-kₙ(t-t₀ₙ)}), avec kₙ=2.0, t₀ₙ=T/2)**, à comparer. | Calibration via runs initiaux (N=5, T=20, Iₙ(t) ~ U[0,1], seed fixée) | Moyenne | Eₙ(t), Oₙ(t) | logs optionnels |
| envₙ(x,t) | Gaussienne, σₙ(t) = 0.1 (temporaire) | Forme temporaire : **Statique, enveloppe gaussienne ( = 0.1) ET dynamique (= 0.1+0.05 sin(2π t/T)). Testé avec μₙ(t)=0 (fixe), μₙ(t) dynamique optionnelle phase 2**,à comparer. | Calibration via runs initiaux (N=5, T=20, Iₙ(t) ~ U[0,1], seed fixée) | Secondaire | σₙ(t), μₙ(t) | logs optionnels |
| μₙ(t) | 0.0 | **Décalage centre env.** Forme temporaire ; **à affiner dynamiquement après phase 1** | Calibration via runs initiaux (N=5, T=20, Iₙ(t) ~ U[0,1], seed fixée) | Secondaire | t | logs optionnels |
| σₙ(t) | 0.1 | **Écart-type env.** Forme temporaire ; **à affiner dynamiquement après phase 1** | Calibration via runs initiaux (N=5, T=20, Iₙ(t) ~ U[0,1], seed fixée) | Secondaire | t, feedback | logs optionnels |
| θ(t), η(t) | — | Forme temporaire ; **à** **modéliser** **selon** **besoin** **et** **ajuster** **après** **phase** **1** | À définir après phase 1 selon besoins d’adaptation | Exploratoire | G(x, t) | logs optionnels |
| Eₙ(t) | — | Sortie attendue (émergent) | À valider par corrélation sur runs (ex : comparer Eₙ(t) et Oₙ(t) avec S(t) comme référence, sur logs runs initiaux) | Critique | harmonisation, feedback | Eₙ(t) (optionnel) |
| Oₙ(t) | — | Sortie actuelle (émergent) | Observé comme sortie simulée ; validation empirique sur logs | Critique | simulation | Oₙ(t) (optionnel) |
| S(t) | — | Signal global | Empirique, run complet | Critique | toutes strates | S(t), logs principaux |
| A(t) | — | (1/N) Σₙ Δfₙ(t) | Empirique | Critique | Δfₙ(t) | A\_mean(t) |
| C(t) | — | (1/N)Σ cos(φₙ₊₁−φₙ) | Empirique, comparatif Kuramoto | Critique | φₙ | C(t) |
| A\_spiral(t) | — | C(t)·A(t) | Empirique | Exploratoire | C(t), A(t) | logs optionnels |
| E(t) | — | maxₙ S(t) | Empirique | Critique | S(t) | E(t) |
| L(t) | — | argmaxₙ S(t) | Empirique | Secondaire | S(t) | L(t) |
| mean(cpu\_step) (coût CPU) | Dépend du contrôle | Temps CPU moyen par step | Benchmark | Critique | time.perf\_counter(), t | cpu\_step |
| effort(t) | — | Σₙ |ΔAₙ(t)| + |Δfₙ(t)| + |Δγₙ(t)| | ΔAₙ(t) | Critique | Dépend de ΔAₙ(t)=Aₙ(t)-Aₙ(t-1), Δfₙ(t)=fₙ(t)-fₙ(t-1), Δγₙ(t)=γₙ(t)-γₙ(t-1) | Logger ΔAₙ(t), Δfₙ(t), Δγₙ(t) si variables/non constantes |
| variance\_d2S (fluidité) | 0.01 | var(d²S/dt²) | Analyse spectrale, logs | Critique | S(t), t | variance\_d2S |
| max(S(t))/médiane (stabilité) | 10× | max(S(t)) / médiane(S(t)) | Log analyse, seuils | Critique | S(t), t | S(t), max\_S, med\_S |
| t\_retour (résilience) | 2× médiane | Δt pour retour à 95% état pré-choc | Analyse post-perturbation | Critique | S(t), choc, t | t\_retour |
| entropy\_S (innovation) | 0.5 | Entropie spectrale de S(t) | Analyse spectrale | Critique | S(t), t | entropy\_S |
| mean(|Eₙ(t) − Oₙ(t)|) (régulation) | — | Moyenne absolue |Eₙ(t) − Oₙ(t)| ; si > 2× médiane sur 50 % du run, raffinement feedback (βₙ, G(x)) | Analyse des logs, suivi dynamique de l’écart, vérification convergence/réduction de l’erreur au fil du temps, analyse des oscillations ou stagnations (stabilité de la boucle feedback) | Critique | Eₙ(t), Oₙ(t), βₙ, G(x), γ(t) | mean\_abs\_error ; feedback ; refine\_βₙ (pour tracer si raffinement lancé) |
| mean\_high(effort(t)) | 2× médiane | Moyenne glissante haute | Analyse effort | Critique | effort(t), t | effort\_high |
| d\_effort/dt | 5σ | Dérivée temporelle de effort | Analyse effort transitoire | Critique | effort(t), t | d\_effort\_dt |
| r(t) | φ+ε sin(…) | Ratio spiralé | Empirique, runs initiaux | Secondaire | φ, ε, ω, θ, t | logs optionnels |
| Fₙ(t) | — | βₙ·(Oₙ(t)−Eₙ(t))·γ(t) | Empirique, test feedback | Critique | βₙ, Oₙ(t), Eₙ(t), γ(t) | logs optionnels |
| G(x), Gₙ(x, t) | Forme à préciser selon besoin | Forme temporaire, **choisie parmi les 4 archétypes décrits ; à affiner après runs comparatifs** | Runs comparatifs ; à affiner selon les performances observées | Exploratoire | Dépend de λ, α, β, raffiner selon le mode de simulation | logs optionnels |

*Si effort(t) est comparé entre runs de tailles N différentes, la version normalisée effort(t)/N doit être utilisée.*

*Formule : entropy\_S = -Σ\_k P\_k log P\_k, avec P\_k = |FFT[S(t)]|² / Σ |FFT[S(t)]|² (voir module Python/FFT)*

**Légende :**

* « logs optionnels » = à ajouter dans le CSV si besoin/validation lors de la phase 1 ou après exploration.
* « forme temporaire » ou « hypothèse simplificatrice » pour les paramètres non encore formalisés.

**Note :** Tous les paramètres marqués “forme temporaire” ou “hypothèse simplificatrice” sont à affiner dynamiquement après les runs initiaux (N=5, T=20, Iₙ(t) ~ U[0,1], seed fixée, logs complets). La justification et l’évolution de chaque paramètre seront consignées dans le changelog.txt, à chaque ajustement postérieur. Pour tout paramètre non modélisé à ce stade (“à définir après phase 1”), la valeur et la dynamique seront fixées selon les observations issues des logs.

Phase 1 = runs parallèles pour chaque version ; logs comparés

Structure Changelog : “Run 1A : statique. Run 1B : dynamique. Perturbation sur Iₙ(t) à t=…, limites observées” :

* “Run1 : Iₙ(t) constant, seed=…”
* “Run2 : Iₙ(t)=1 pour t=T/4, sinon 0, seed=…”
* “Run3 : Iₙ(t) linéaire de 0 à 1 sur [0,T]”
* Loguer les effets des perturbations sur S(t), Eₙ(t), Oₙ(t).

Exemple : “ Run2B, γₙ(t) = sigmoïde, 2025-05-19, test latence dynamique, impact : meilleure résilience pour T=100, seed=123”

**Seuils et critères de raffinement dynamique**

Cette section définit pour chaque critère empirique les seuils initiaux à surveiller lors des runs, le mode d’ajustement (après batch de 5 runs), la procédure de modification et le lien à la phase 2 de raffinement.

Toute modification de seuil, d’algorithme ou d’action est loguée dans le changelog avec date, justification et nouvelle valeur.

Définitions et rôle de chaque critère

* Fluidité :

Évalue la continuité et la douceur du mouvement du système (via la variance de la dérivée seconde de S(t)). Une faible fluidité signifie des transitions saccadées ou des effets de seuil abrupts. La fluidité est cruciale pour des dynamiques naturelles et prévisibles.

* Stabilité :

Mesure l’amplitude maximale du signal global S(t). Si le système présente des pics extrêmes (max(S(t)) très supérieur à la médiane), c’est signe de dérive ou d’explosion. La stabilité garantit que le système ne diverge pas, même sous stress.

* Résilience :

Temps moyen nécessaire pour revenir à l’équilibre après une perturbation (t\_retour). Si ce temps devient trop long ou si le système échoue à se rétablir, il manque de résilience. Cela teste la capacité à absorber et corriger les chocs.

* Innovation :

Basée sur l’entropie spectrale du signal. Une faible entropie indique peu de motifs émergents, donc pas d’innovation ou d’adaptation. Ce critère révèle la capacité du système à générer des comportements nouveaux et dynamiques.

* Régulation :

Mesure l’écart moyen entre la sortie attendue et la sortie observée (|Eₙ(t) – Oₙ(t)|). Si cet écart ne diminue pas, ou reste élevé, la régulation (feedback) est inefficace. Ce critère valide la capacité du système à s’auto-corriger, à réduire ses erreurs, à ajuster la plasticité du feedback (βₙ), ou la forme de la fonction dé régulation G(x).

* Coût CPU :

Temps moyen par pas (mean(cpu\_step)), comparé au groupe contrôle. Si le coût CPU explose, l’algorithme est trop lourd ou instable. Ce critère garantit l’efficience computationnelle et la scalabilité.

* Effort interne (chronique) :

Mesure la moyenne glissante haute de l’effort interne (adaptation des amplitudes, fréquences, latences). Un effort chronique élevé suggère que le système lutte en permanence sans trouver d’équilibre. Cela signale une régulation inefficace.

* Effort transitoire :

Dérivée de l’effort : mesure l’intensité des variations courtes et aiguës (pics de correction). Trop de pics suggèrent une plasticité mal dosée ou des oscillations internes instables.

**Tableau structuré des seuils et actions associées**

| **Critère empirique** | **Seuil initial (modifiable)** | **Action si seuil franchi** | **Ajustement après 5 runs** | **Dernière modification** |
| --- | --- | --- | --- | --- |
| Fluidité | variance\_d2S > 0.01 sur > 70% du run | Raffiner γₙ(t), envₙ(x, t) | Oui | Numéro de batch/run ayant motivé l’ajustement |
| Stabilité | max(S(t)) > 10 × médiane(S(t)) sur >5% du run | Raffiner σ(x), αₙ | Oui | Numéro de batch/run ayant motivé l’ajustement |
| Résilience | t\_retour > 2× médiane | Raffiner αₙ, βₙ | Oui | Numéro de batch/run ayant motivé l’ajustement |
| Innovation | entropy\_S < 0.5 sur > 70% du run | Raffiner θ(t), η(t), μₙ(t) | Oui | Numéro de batch/run ayant motivé l’ajustement |
| Régulation | mean(|Eₙ(t) − Oₙ(t)|) > 2× médiane(|Eₙ(t) − Oₙ(t)|) sur >50% du run | Raffiner βₙ, feedback G(x) | Oui | Numéro de batch/run ayant motivé l’ajustement |
| Coût CPU | mean(cpu\_step) > 2× contrôle sur >80% du run | Optimiser code ou complexité dynamique | Oui | Numéro de batch/run ayant motivé l’ajustement |
| Effort interne (chronique) | mean\_high(effort(t)) > 2× médiane sur >80% du run | Raffiner αₙ, μₙ(t), σₙ(t) | Oui | Numéro de batch/run ayant motivé l’ajustement |
| Effort transitoire | d\_effort/dt > 5σ plus de 10 fois/run | Raffiner w\_{ni} | Oui | Numéro de batch/run ayant motivé l’ajustement |

Notes d’application :

* variance\_d2S : variance de la dérivée seconde de S(t), mesure la fluidité du signal global (transitions douces vs saccadées).
* max(S(t)) : maximum du signal global S(t) comparé à sa médiane, mesure la stabilité (présence d’extrêmes).
* t\_retour : temps moyen de retour à l’équilibre après une perturbation du système, mesure la résilience (cf. logs).
* entropy\_S : entropie spectrale du signal S(t), mesure l’innovation (richesse des motifs dynamiques émergents).
* mean(|Eₙ(t) − Oₙ(t)|) : moyenne de l’écart absolu entre la sortie attendue et la sortie observée, mesure la qualité de la régulation.
* mean(cpu\_step) : temps de calcul moyen par pas de temps (Δt/N), mesure le coût computationnel relatif à la charge attendue (contrôle).
* mean\_high(effort(t)) : moyenne glissante haute de effort(t) sur les 80% du temps les plus élevés, mesure l’effort interne chronique.
* d\_effort/dt : dérivée temporelle de effort(t), mesure l’intensité des corrections brèves (effort transitoire).

**Protocole d’ajustement et log**

* Après chaque batch (typiquement 5 runs, adaptable si besoin ), analyser le franchissement de seuil :
  + Si un critère dépasse son seuil sur >50% des runs du batch, ajuster le(s) paramètre(s) associé(s) (ex : passer αₙ de 0.5 à 0.7).
  + Reporter dans la feuille de route et le changelog. Exemple : « Run Batch 2 : entropy\_S < 0.5 sur 4/5 runs, θ(t) raffiné de 0.1 à 0.15, justification : manque d’innovation détecté. »
  + Les seuils initiaux sont fixés par la théorie ou les premiers runs exploratoires ; ils peuvent évoluer lors de la phase 2 en fonction des observations.
  + Ajout automatique de nouveaux critères. Après l’analyse post-run par explore.py,
    - Si une anomalie ou un phénomène récurrent détecté n’est pas couvert par les 8 critères empiriques,
    - Proposer l’ajout d’un nouveau critère ou l’ajustement d’un seuil,
    - Documenter la décision dans le changelog,
    - Ajouter le critère ou seuil dans la prochaine version du tableau structuré.
  + “statut effort\_status” (stable/transitoire/chronique) doit être tracé pour toutes les situations, même si non déclencheur, afin de garantir la traçabilité continue.

**Rappel : Critères utilisés pour la phase 2 de raffinement**

Pour chaque critère :

* Fluidité : variance\_d2S, dérivée locale de S(t)
* Stabilité : max(|S(t)|), variance de S(t), anomalies longues
* Résilience : temps de retour à l’équilibre après choc
* Innovation : entropy\_S, analyse spectrale
* Régulation : Écart moyen |Eₙ(t) − Oₙ(t)| (ou son évolution), efficacité du feedback (réduction d’erreur), stabilité de la boucle G(x).
* Coût CPU : mean/max(cpu\_step)
* Effort interne : mean\_high(effort(t)), statut effort\_status
* Effort transitoire : pics de d\_effort/dt

Chaque critère est surveillé ; franchissement d’un seuil => raffinement d’un ou plusieurs paramètres, toujours logué et justifié. Les critères qui ne déclenchent pas d’action sont aussi logués à chaque run. Un “mode alerte” pour chaque critère : si un critère non-déclencheur présente un écart >3σ, cela doit déclencher une documentation spécifique.

**Exemple de log pour modification de seuil**

[2025-05-20] :

Après runs 1-5, mean\_high(effort(t)) > 2× médiane sur 4/5 runs, αₙ augmenté de 0.5 → 0.7.

entropy\_S < 0.5 dans 3/5 runs, θ(t) ajusté de 0.1 à 0.12.

Seuils inchangés, mais statut effort\_status ajouté pour suivi.

**Note phase 2 :** Critères de raffinement des paramètres dynamiques

Les raffinements dynamiques de certains paramètres (μₙ(t), θ(t), η(t), φₙ(t)) seront déclenchés explicitement en fonction de critères observés lors des runs initiaux. Les critères indicatifs suivants pourront être utilisés :

– Si entropy\_S < 0.5 sur la majorité des pas, indiquer une innovation trop faible → proposer un raffinement sur θ(t), η(t) ou μₙ(t).

– Si variance de C(t) > 0.1 sur 80% des pas, indiquer une instabilité de synchronisation → proposer un raffinement sur φₙ(t).

– Si d’autres métriques (effort chronique élevé, S(t) chaotique) dépassent un seuil fixé dans la config, raffinement sur μₙ(t), σₙ(t), etc.

Chaque déclencheur et action de raffinement devra être consignée dans le changelog.txt, et précisée dans le tableau structuré si mise en œuvre.

| **Critère mesuré (sur logs runs initiaux)** | **Seuil objectif** | **Action déclenchée** |
| --- | --- | --- |
| Moyenne(entropy\_S) < 0.5 | 70% du temps simulé | Raffiner θ(t), η(t) ou μₙ(t) |
| Variance(C(t)) > 0.1 | 80% du temps simulé | Raffiner φₙ(t) |
| Moyenne glissante high(effort(t)) > 2× la médiane | 80% du temps simulé | Raffiner μₙ(t), σₙ(t) |
| Nombre d’occurrences S(t) chaotique (> 3σ bruit) | ≥ 5 fois sur le run | Raffiner μₙ(t), σₙ(t) |

* Ces seuils sont adaptables après les 5 premiers runs mais doivent être explicités avant d’y toucher à la main.
* À chaque franchissement de seuil, on logue l’événement, on change la version du paramètre et on note la justification dans le changelog.
* Forme de l’algorithme :

1. On run 5 simulations (N=5, T=20, …).
2. À la fin de chaque run, on calcule pour chaque critère :

* Est-ce que le seuil est franchi ? (Oui/Non)
* Si Oui : on lance le raffinage du paramètre associé.
* Si Non : on garde la version précédente.

1. Le changement est documenté avec :

* la valeur observée,
* la date,
* la nature du changement (statique → dynamique, etc).

Tous les critères restent suivis même s’ils ne déclenchent pas un raffinement direct (certains servent surtout à la documentation ou à la validation empirique visuelle)

* Prévoir un “mode alerte” dans les logs si un critère non-déclencheur montre un écart >3σ,

Chaque seuil de déclenchement est défini initialement, mais peut être ajusté à chaque étape de raffinement explicitement documentée : tout changement de seuil doit être enregistré avec la date, le run concerné, la valeur nouvelle et la justification dans le changelog ou l’historique des seuils.

**Dans le code et les logs :**

* Ajouter un bloc de code pour lire ces seuils “dynamiques” à chaque début de run.
* Ajouter un log automatique à chaque fois qu’un seuil est modifié (date, run, valeur, justification).
* Possibilité d’exporter le “journal des seuils” avec les logs CSV pour auditabilité totale.
* Loguer explicitement mean(|Eₙ(t) − Oₙ(t)|) à chaque run.
* Surveiller la convergence : si la régulation ne réduit pas l’écart au fil du temps, ou s’il y a oscillations excessives, c’est un signe de mauvaise régulation.

Exemple de formalisation dans le changelog

2025-05-21 | Run2B | entropy\_S = 0.42 sur 74% des pas | Raffinage de θ(t) : statique → dynamique | seed=123

2025-05-21 | Run3A | effort chronique > 2× médiane sur 86% du temps | Raffinage de μₙ(t) : statique → sinus | seed=124

Il faut ajouter une fonction d’analyse automatique dans le notebook ou le module Python pour “analyser\_criteria\_and\_refine()”.

**Plan de raffinement phase 2**

À l’issue des runs initiaux, chaque paramètre à forme dynamique pourra être raffiné si et seulement si un ou plusieurs critères objectifs sont franchis, tels que détaillés ci-dessus. Le tableau structuré et le changelog devront consigner :

– Le critère déclencheur (ex : entropy\_S, variance de C(t), etc.)

– Le paramètre raffiné

– La nouvelle forme/dynamique adoptée

Ce protocole garantit la transparence, la traçabilité et la falsifiabilité de chaque évolution du modèle FPS.

**Note de validation et gestion des logs**

* Conditions standards de validation :
  + Calibration via runs initiaux (N=5, T=20, Iₙ(t) ~ U[0,1], seed fixée, logs complets).
  + Perturbations spécifiques à préciser dans le tableau si utilisées.
* Gestion des logs :
  + Colonnes minimales du CSV principal : S(t), A\_mean(t), f\_mean(t), effort(t), cpu\_step(t), C(t), E(t), L(t).
  + Pour N > 10, exporter Aₙ(t), fₙ(t) dans fichiers séparés (ex. A\_n\_{id}.csv, f\_n\_{id}.csv). Fusion possible via merge\_logs.py.
  + Logs optionnels (variance\_d2S, entropy\_S, effort\_status, etc.) ajoutés lors de la phase 1 ou après analyse.
  + Logs d'exploration :
    - Exécution automatique de explore.py après chaque run
    - Logs d'émergence : emergence\_events\_<run\_id>.csv avec colonnes : event\_type, t\_start, t\_end, metric, value, severity
    - Rapport d'exploration : exploration\_report\_<run\_id>.md
    - Seuils de détection dans config.json section « exploration »
* Plan de raffinement :
  + Tout paramètre en forme temporaire (γₙ(t), envₙ(x,t), etc.) sera ajusté après analyse des logs de runs initiaux, avec justification ajoutée au changelog (Raffiner la forme/validation après phase 1 ; documenter chaque évolution dans changelog.txt.)

(Rappel : chaque paramètre listé dans la colonne “Colonnes log associées” doit apparaître dans au moins un fichier log, pour assurer la traçabilité complète.)

**Annexe / Phase 2 : Matrice de dépendances pour les termes mathématiques**

But : pour chaque terme mathématique, lister :

* de quelles variables/fonctions il dépend (inputs),
* qui l’utilise (outputs/fonctions appelantes).

Exemple :

* Tabulaire

| **Fonction / Terme** | **Dépend de (inputs)** | **Utilisé par (outputs)** | **Liens à loguer** |
| --- | --- | --- | --- |
| init\_strates() | config, seed | strates[] | — |
| compute\_Iₙ(t) | contexte (ext.), stratégie de perturbation | Iₙ(t) | logs optionnels |
| compute\_Aₙ(t) | A₀, σ(x), k, x₀, Iₙ(t) | Aₙ(t) | Aₙ(t) |
| σ(x) | k, x₀, Iₙ(t) | compute\_Aₙ(t) | — |
| compute\_fₙ(t) | f₀ₙ, Δfₙ(t) | fₙ(t) | fₙ(t) |
| Δfₙ(t) | αₙ, w\_{ni}, S\_i(t) | fₙ(t) | Δfₙ(t) |
| compute\_φₙ(t) | φₙ, dynamique phase optionnelle | φₙ(t) | φₙ(t) |
| compute\_C(t) | φₙ(t) | C(t) | C(t) |
| compute\_A\_spiral(t) | C(t), A(t), Δfₙ(t) | A\_spiral(t) | A\_spiral(t) |
| compute\_Fₙ(t) | βₙ, Oₙ(t), Eₙ(t), γ(t) | Fₙ(t) | Fₙ(t) |
| compute\_G(x) | Fₙ(t), G archétype, Eₙ(t), Oₙ(t), params (λ, α, β) | G(x), feedback global | logs optionnels |
| compute\_S(t) | Aₙ(t), fₙ(t), φₙ(t), γₙ(t), G(x) | S(t) | S(t) |
| compute\_E(t) | S(t) | E(t) | E(t) |
| compute\_L(t) | S(t) | L(t) | L(t) |
| compute\_cpu\_step(t) | time.perf\_counter(), N | cpu\_step(t) | cpu\_step(t) |
| compute\_effort(t) | ΔAₙ(t), Δfₙ(t), Δγₙ(t) | effort(t) | effort(t) |
| compute\_effort\_status | effort\_t, history | status\_string | effort\_status |
| compute\_t\_retour | S\_history, t\_choc | t\_retour | t\_retour |
| envₙ(x,t) | σₙ(t), μₙ(t), dynamique gaussienne/sigmoïde | Gₙ(x,t), logs optionnels | — |
| μₙ(t), σₙ(t) | Feedback, logs, dynamique contextuelle | envₙ(x,t), logs optionnels | — |
| θ(t), η(t) | G(x, t), adaptation contextuelle | G(x, t), logs optionnels | — |
| compute\_S\_i(t) | history, n | S\_i(t) | — |
| compute\_Oₙ(t) | Aₙ, fₙ, phi\_n, gamma\_n | Oₙ(t) | Oₙ(t) |
| compute\_Eₙ(t) | history, n, A0 | Eₙ(t) | Eₙ(t) |
| explore\_phase\_space | S(t), metric, window | recurrence\_matrix | emergence\_events.csv |
| detect\_anomalies | data, metrics, threshold | anomaly\_list | emergence\_events.csv |
| detect\_spiral\_bifurc | C(t), threshold | bifurcation\_events | emergence\_events.csv |
| detect\_harmonic\_emerg | S(t), n\_harmonics | harmonic\_events | emergence\_events.csv |
| run\_exploration | logs CSV/HDF5 | all\_events | rapport MD + CSV |
| detect\_fractal\_patterns | S(t), C(t), effort(t), config[“exploration”] | fractal\_events.csv, rapport .md | Section “fractal motifs” dans rapport |

Dépendance locale : la dynamique de S\_i(t) est explicitée ici car elle structure la modulation adaptative Δfₙ(t). Cette chaîne de calcul opère à chaque pas, en parallèle du pipeline global

Schéma :

[history] ──> [compute\_S\_i(t)] ──> S\_i(t)

|

V

[compute\_delta\_fn(t)]

* Schématique

[config, seed]

|

V

[init\_strates()] ---> strates[]

|

V

[compute\_Iₙ(t)] ---> Iₙ(t)

|

V

[σ(x)] ———> [compute\_Aₙ(t)] ---> Aₙ(t)

| |

+----------------------+

|

V

[Δfₙ(t)] ———> [compute\_fₙ(t)] ---> fₙ(t)

|

V

[compute\_φₙ(t)] ---> φₙ(t)

|

V

[compute\_C(t)] ---> C(t)

|

V

[compute\_A\_spiral(t)] ---> A\_spiral(t)

|

V

[compute\_Fₙ(t)] ---> Fₙ(t)

|

V

[compute\_G(x)] ---> G(x)

|

V

[compute\_S(t)] ---> S(t)

|

V

[compute\_E(t), compute\_L(t)] ---> E(t), L(t)

|

V

[compute\_cpu\_step(t)] ---> cpu\_step(t)

[compute\_effort(t)] ---> effort(t)

[log\_metrics()] ---> log (CSV/HDF5)

|

V

[run\_exploration()] --- explore.py (Analyse post-run sur les logs)

|

V

[generate\_report()] --- Rapport MD, visualisation, emergence\_events.csv

Chaque bloc a ses entrées, ses sorties, et se connecte aux blocs en aval, tout en enregistrant les résultats dans les logs.

Voici une analyse rigoureuse de l’ensemble des termes extraits du chapitre 4 de la FPS, avec une classification selon quatre axes :

1. Défini explicitement (formule ou relation directe)
2. Émergent (résultat d’un processus ou d’une agrégation)
3. À définir (mentionné mais sans structure claire ou sans lien causal donné)
4. Paramètre numérique fixe ou ajustable

**1. Défini explicitement**

|  |  |
| --- | --- |
| **Terme** | **Définition claire dans le texte** |
| S(t) | Simple : ∑ₙ Aₙ(t) · sin(2π·fₙ(t)·t + φₙ(t))  Étendue :  ∑ₙ [Aₙ(t) · sin(2πfₙ(t)·t + φₙ) · γₙ(t)] · G(Eₙ(t) - Oₙ(t)) |
| Aₙ(t) | A₀ · σ(Iₙ(t)) |
| σ(x) | 1 / (1 + e^(−k(x − x₀))) |
| fₙ(t) | f₀ₙ + Δfₙ(t) |
| Δfₙ(t) | αₙ · Σ\_i w\_{ni} · S\_i(t) |
| A(t) | (1/N) · Σₙ Δfₙ(t) |
| C(t) | (1 / N) · ∑ₙ₌₁ⁿ⁻¹ cos(φₙ₊₁(t) − φₙ(t)) |
| A\_spiral(t) | C(t) · A(t) |
| Fₙ(t) | βₙ · (Oₙ(t) − Eₙ(t)) · γ(t) |
| fₙ₊₁(t) | r(t) · fₙ(t) |
| r(t) | φ + ε · sin(2π·ω·t + θ) |
| E(t) | maxₙ |
| L(t) | argmaxₙ |
| G(x) | tanh(λx), sin(x)/x, sin(βx)·exp(−αx²), etc. (plusieurs versions proposées) |
| Gₙ(x, t) | Aₙ(t) · sinc[fₙ(t) · (x − μₙ(t))] · envₙ(x, t) |
| G(x, t) | η(t) · sin(θ(t) · x) |

**2. Émergent**

**(non défini par une équation, mais résultant de l’état du système)**

|  |  |
| --- | --- |
| **Terme** | **Source** |
| S\_i(t) | Signal d’autres strates (intrinsèquement dépendant du système global). Exploratoire phase 1 : Signal des autres strates = S(t-dt) - On(t-dt) pour t>0, 0sinon |
| Eₙ(t) | Intention harmonique (ciblée mais non explicitement calculée). Émergent, mais la dynamique peut être influencée par les versions statique/dynamique des paramètres modulant la régulation (ex : γ(t)). Exploratoire phase 1 : En(t) = φ \* On(t-dt) où φ est le nombre d'or |
| Oₙ(t) | Sortie actuelle (observée, pas modélisée explicitement). Émergent, mais la dynamique peut être influencée par les versions statique/dynamique des paramètres modulant la régulation (ex : γ(t)). Exploratoire phase 1 : On(T) = An(t) \* sin(2π\*fn(t)\*t + φn(t)) \* γn(t) |

**3. À définir / partiellement défini**

|  |  |
| --- | --- |
| **Terme** | **Commentaire** |
| γ(t) | γ(t) : explicitée sous forme statique (=1.0) et dynamique (sigmoïde : 1/(1+e^{-k(t-t₀)}),avec k=2.0, t₀=T/2) à comparer en phase 1. |
| γₙ(t) | γₙ(t) : explicitée sous forme statique (=1.0) et dynamique (sigmoïde :1/(1+e^{-kₙ(t-t₀ₙ)}), avec kₙ=2.0, t₀ₙ=T/2) à comparer en phase 1. |
| envₙ(x,t) | La version statique (σₙ(t)=const) et la version dynamique (σₙ(t) sinus) sont prévues pour runs initiaux. |
| μₙ(t) | fixe dans la phase 1, dynamique possible phase 2 |
| σₙ(t) | Idem, dépend du feedback mais pas de formule universelle (fixe (const) et dynamique (sinus) à tester lors de la phase 1 |
| θ(t), η(t) | Éléments de G(x, t), adaptatifs, à modéliser selon le contexte. Modélisation différée à la phase 2 sauf besoin identifié lors de l’analyse des logs de la phase 1. |
| φₙ(t) | Fixe dans la première version, dynamique possible après phase 1 (fixe ou à raffiner si dynamique jugée pertinente). |

**4. Paramètres numériques fixes ou réglables**

|  |  |
| --- | --- |
| **Terme** | **Description** |
| A₀ | Amplitude de base propre à la strate |
| βₙ | Plasticité / gain de rétroaction d’une strate |
| αₙ | Souplesse d’adaptation |
| w\_{ni} | Poids de liaison entre strates |
| k | Sensibilité de la sigmoïde σ |
| x₀ | Seuil de basculement de la sigmoïde |
| φ | Nombre d’or ≈ 1.618 (peut être fixé ou juste attracteur) |
| ε | Amplitude harmonique (typiquement ≈ 0.05) |
| ω | Fréquence lente de modulation |
| θ | Phase de départ |
| λ, α, β | Paramètres de G(x), selon la version |

**Note :** Certains paramètres (σₙ(t), γ(t), γₙ(t)) sont explicitement testés à la fois en mode statique (valeur numérique) et dynamique (fonction du temps ou du contexte) dès la phase 1, voir tableau structuré des paramètres et métriques pour valeurs initiales.

Pour chaque paramètre à forme temporaire ou dynamique, voir le détail dans le tableau structuré, où sont explicitées les versions testées dès la phase 1.

**Remarques importantes**

* G(x) n’est pas une seule fonction mais un espace de formes possibles, avec 4 archétypes définis. On choisit la forme selon le contexte du système simulé.
* Le terme φₙ (phase d’origine) est mentionné sans formule de mise à jour, ce qui laisse place à une fonction potentielle de déphasage dynamique à inventer si besoin.
* Chaque paramètre à forme temporaire a une version dynamique documentée dans le tableau structuré et le choix se fait à chaque run.
* En phase 2, de nouveaux raffinements sur μₙ(t), θ(t), η(t) ou φₙ(t) seront explorés si les runs initiaux le justifient.

**Pour reconstruire le code FPS de zéro :**

**1. Définir un objectif falsifiable clair**

Par exemple : “Tester la stabilité dynamique et l’auto-régulation du système S(t) selon la définition FPS, sous variation d’Iₙ(t) et de Eₙ(t), sur N strates pendant T secondes.”

Ce test de base pourra être modifié à l’avenir (résilience, bifurcation, etc.), mais il donne un critère observable (S(t), Aₙ(t), E(t), L(t), etc.)

**C’est à dire :**

* Stabilité dynamique : S(t) reste borné, ne diverge pas, même sous perturbation (entrées contextuelles Iₙ(t) ou changement d’Eₙ(t))
* Auto-régulation : le système, via G(x), réduit progressivement l’écart entre Eₙ(t) et Oₙ(t), sans intervention externe
* N strates : nombre de couches fractales
* T secondes : durée de simulation

**Est-ce que c’est pertinent ?**

Oui, si :

* on observe réellement des comportements émergents mesurables (fluidité, résilience, synchronisation adaptative…)
* on compare avec un système sans ces régulations (ex. : oscillateurs de Kuramoto, ou FPS “désactivée”)

**Comment le mesurer ?**

* **Stabilité** : borne supérieure de |S(t)| ou max d’E(t)
* **Régulation** : écart |Eₙ(t) − Oₙ(t)| qui diminue dans le temps
* **Fluidité** : dérivée continue et stable (low d²S/dt²)
* **Résilience** : retour à l’équilibre après un choc sur Iₙ(t)
* **Innovation** : apparition spontanée de patrons (peut être visuel ou par entropie spectrale)
* **Le coût CPU à chaque pas de temps** :

Est-ce que la forme actuelle est toujours adaptée ?

Oui, en partie.

Dans l’ancien code, on avait une mesure du time.perf\_counter() avant et après chaque step pour estimer le coût CPU (ou temps de traitement) par pas de temps.

À faire maintenant :

* Garder ce principe : mesurer Δt entre le début et la fin de chaque boucle FPS (calculs pour N strates)
* Ajouter une normalisation :  
    
   cpu\_step(t) = Δt / N (ou autre métrique par strate ou unité logique)
* Stocker cpu\_step(t) dans le CSV de simulation
* Optionnel : calculer mean(cpu\_step), max(cpu\_step) pour comparer entre FPS et Kuramoto

Conclusion : À intégrer tel quel, mais en le liant proprement au S(t) pour qu’il soit une mesure de “coût computationnel de l’émergence FPS”.

* **Effort interne** :

Dans l’ancien code, effort était défini comme :

« Somme des variations internes, ou dérivées locales d’adaptation, reflétant l’intensité de l’ajustement du système. »

Est-ce trop abstrait ?

Pas si on le définit rigoureusement.

Voici une version formalisable :

*effort(t) = Σₙ |ΔAₙ(t)| + |Δfₙ(t)| + |Δγₙ(t)|*

*Autrement dit : la somme des variations des amplitudes, fréquences et latences à chaque pas de temps. Cela donne une mesure :*

* *de plasticité interne*
* *de charge d’adaptation*
* *de coût d’ajustement*

Conclusion : On peut garder « effort », mais en le redéfinissant comme un agrégat clair, et en le comparant à la stabilité.

* **Effort intense temporaire vs. effort chronique élevé**

Pourquoi :

On mesure un effort interne du système (variation d’Aₙ(t), fₙ(t), γₙ(t)).

Mais il faut distinguer deux cas :

* Effort élevé sur un court moment = souvent adaptatif etsain (ex. réponse à une perturbation)
* Effort élevé constant dans le temps = souvent signe de mauvaise régulation (le système lutte sans se stabiliser)

On peut donc coder une logique comme :

* Si effort(t) > seuil mais revient vite → bonne résilience
* Si effort(t) reste élevé sur 80 % de la simulation → dysfonction

On peut l’ajouter comme colonne dans les logs (CSV) :

*t, effort(t), effort\_status = {"stable", "transitoire", "chronique"}*

Ajouter une référence (dans le changelog, par exemple) si une corrélation anormale effort/cpu est détectée, même si aucune action immédiate n’est déclenchée.

**Grille d’évaluation empirique des simulations FPS (note de 1 à 5)**

Option d’ergonomie

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Critère** | **1 - Minimal / Désaligné** | **2 - Faible / Instable** | **3 - Moyen / Fonctionnel** | **4 - Bon / Harmonieux** | **5 - Optimal / FPS-idéal** |
| Stabilité | Oscillations erratiques, effondrement fréquent | Résilience partielle mais déphasages majeurs | Stabilité de base, sans effondrement | Stabilité dynamique avec adaptation | Auto-stabilisation même en contexte extrême |
| Régulation | Feedback absent ou explosif | Réponses mal dosées, boucle rigide | Feedback actif mais non harmonieux | Feedback fluide et cohérent | Régulation spiralée douce et efficace |
| Fluidité | Mouvements saccadés, effets de seuil brutaux | Alternances incohérentes, transitions abruptes | Transitions acceptables mais un peu rigides | Transitions douces dans la majorité des cas | Transitions naturelles, presque organiques |
| Résilience | Effondre au moindre choc contextuel | Résiste partiellement aux variations | Résiste aux variations modérées | Absorbe les perturbations courantes | S’adapte et apprend des perturbations extrêmes |
| Innovation | Répétition stérile, aucun pattern émergent | Petites variations non significatives | Oscillations diversifiées mais stériles | Émergence de motifs dynamiques | Harmonie créative, transformation structurée |
| Coût CPU | Dépassement critique, inefficience extrême | Gaspillage de cycles fréquents | Usage CPU acceptable | Bonne efficience dans la gestion | Optimisation continue du traitement |
| Effort interne | Activité chaotique, tensions non résolues | Excès de corrections internes sans convergence | Régulations coûteuses mais convergentes | Équilibre entre correction et repos | Moindre effort avec ajustements spontanés |

**2. Créer un squelette de pipeline clair avec blocs modulaires (1 module = 1 équation)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Étape** | **Module** | **Terme(s) FPS concernés** | **But** |
| 1 | init\_strates() | A₀, f₀ₙ, φₙ, αₙ, βₙ, k, x₀ | Initialisation |
| 2 | compute\_Iₙ(t) | Iₙ(t) | Calcul du contexte |
| 3 | compute\_Aₙ(t) | Aₙ(t), σ(x), k, x₀ | Amplitude adaptative |
| 4 | compute\_fₙ(t) | fₙ(t), f₀ₙ, Δfₙ(t), αₙ, w\_{ni}, S\_i(t) | Modulation fractale |
| 5 | compute\_φₙ(t) | φₙ | (fixe ou dynamique, à définir) |
| 6 | compute\_C(t) | C(t), φₙ | Accord spiralé |
| 7 | compute\_A\_spiral(t) | A(t), A\_spiral(t), Δfₙ(t), C(t) | Cohérence rythmique |
| 8 | compute\_Fₙ(t) | Fₙ(t), βₙ, Eₙ(t), Oₙ(t), γ(t) | Feedback |
| 9 | compute\_G(x) | G(x), Gₙ(x, t), etc. | Réaction à l’erreur |
| 10 | compute\_S(t) | S(t), toutes les strates | Signal final |
| 11 | compute\_E(t), L(t) | E(t), L(t) | Indicateurs globaux |
| 12 | log\_metrics() | — | Empirisme falsifiable |
| Post-analyse | explore.py | S(t), C(t), A\_spiral(t), … | Détection d’émergences, anomalies, bifurcations, cycles. Rapport synthétique + figures, logs événements |

**Note :** Pour chaque compute\_X, la version dynamique (si disponible) est appelée selon les paramètres du run, sinon la version statique. Chaque run doit être paramétré (statique/dynamique, forme de perturbation, seed) dans un bloc config/log dédié. Aussi, chaque bloc aurait ses propres tests unitaires (empirisme), et on pourrait les activer/désactiver pour comparer avec ou sans feedback, sans spiralisation, etc. (falsifiabilité). Aussi, Chaque compute\_X() a un test unitaire pour les versions statique et dynamique, pour valider leur implémentation.

**Bloc : Exploration et détection d’émergence (explore.py)**

explore.py : Ce module opère après chaque simulation (run ou batch) sur les logs produits (CSV/HDF5).

Il analyse l’ensemble des métriques loguées (S(t), C(t), A\_spiral(t), etc.) afin de :

* Détecter les événements atypiques (anomalies, bifurcations, émergences harmoniques, cycles récurrents…)
* Détecter l’existence de motifs fractals et auto-similaires à différentes échelles (par ex. avec des méthodes type “recurrence plot” ou “fractal dimension”, voir paramètre threshold dans config)
* Générer des logs structurés d’événements (emergence\_events\_<run\_id>.csv)
* Produire un rapport lisible pour humain (exploration\_report\_<run\_id>.md)
* Fournir, si besoin, des figures de vérification (FFT, recurrence plots…)

Ce module s’intègre dans la pipeline comme une étape systématique post-simulation, avant toute phase de raffinement, recalibrage ou publication.

Il est appelé :

• Automatiquement après chaque run, pour garantir la traçabilité totale des phénomènes émergents ou inattendus.

• Manuellement, pour l’exploration offline de résultats anciens ou externes.

Inputs : Logs CSV/HDF5 produits par le run, config de détection (seuils, métriques), paramètres d’échelle fractale et seuil de similarité définis dans config.json, section exploration.

Outputs : CSV d’événements, rapport markdown, figures optionnelles, logs événements fractals (fractal\_events\_<run\_id>.csv), résumé synthétique dans le rapport exploration\_report\_<run\_id>.md (section dédiée “Motifs fractals détectés”)

Liste structurée des logs produits et outputs d’exploration :

* fractal\_events\_<run\_id>.csv
  + Colonnes : type, metric, t\_start, t\_end, scale, score, severity
  + Rôle : Journal structuré des événements fractals détectés lors de l’exploration post-run (détection automatique via explore.py ou module dédié).
* exploration\_report\_<run\_id>.md
  + Section “Motifs fractals détectés (fractal patterns)” :
    - Contenu : Résumé textuel, figures associées, et interprétation synthétique des motifs fractals identifiés lors du run.
    - But : Fournir une synthèse humaine des patterns auto-similaires ou émergents, avec visualisation.

NB : Cette section complète le log d’événements d’émergence (emergence\_events\_<run\_id>.csv) et le rapport général, en se focalisant sur la dimension fractale de la dynamique du système.

Rôle méthodologique :

• Permet d’auditer la plasticité du modèle : toute émergence détectée doit être réplicable (via seed/config).

• Ouvre la porte au raffinement automatique : certains seuils ou modes de régulation ne sont ajustés qu’après détection d’événements significatifs via ce module.

**Tests unitaires**

Pour chaque fonction compute\_X(), des tests unitaires doivent couvrir :

– le mode statique (ex : compute\_γ(t) retourne 1.0)

– le mode dynamique (ex : compute\_γ(t) retourne bien la valeur sigmoïde attendue pour t, k, t₀ donnés)

Exemple :

*assert compute\_γ(0, mode="statique") == 1.0*

*assert abs(compute\_γ(t=10, mode="dynamique", k=2.0, t0=10) - 0.5) < 1e-6*

Un fichier test\_fps.py centralisera ces tests.

Le résultat des tests (succès/échec) sera reporté à chaque modification majeure dans le changelog.

**Deux options :**

**A.**

**Notebook (fps\_main.ipynb)**

**:**

* Cellule 1 : init\_strates()
* Cellule 2 : compute\_Aₙ(t)
* Cellule 3 : compute\_fₙ(t)
* … etc.
* À chaque pas de temps, on réexécute toutes les cellules, ou on boucle dans une cellule de simulation.

Cela fonctionne si on est en exploration ou en présentation.

**B.**

**Modules Python (.py)**

**— recommandé pour rigueur**

* init.py → initialise toutes les strates et paramètres
* simulate.py → boucle temporelle principale
* metrics.py → calcule tous les indicateurs (S(t), E(t), L(t), etc.)
* visualize.py → plots ou animations
* config.json → tous les paramètres numériques

Avantages : modularité, reproductibilité, rigueur.

On peut toujours appeler les fonctions dans un notebook si besoin.

Donc : on fait les .py pour l’expérience falsifiable, et un .ipynb pour visualisation ou documentation.

**Suivre rigoureusement le dictionnaire FPS**

À chaque terme du dictionnaire correspond une fonction compute\_X() explicite, loggée, isolable.

On peut aussi ajouter un config.json pour :

*{*

*"N": 20,*

*"T": 100,*

*"A0": 1.0,*

*"k": 2.0,*

*"x0": 0.5,*

*"phi": 1.618,*

*...*

*}*

**3. Structure du config.JSON**

Le fichier config.json sert de point d’ancrage pour toutes les simulations. Il centralise les paramètres, la structure des strates, les modes de simulation, les seeds, les choix de perturbation et les seuils à calibrer.

**Exemple minimal de structure config.json**:

*{*

*"system": {*

*"N": 20,*

*"T": 100,*

*"seed": 12345,*

*"mode": "FPS", // ou "Kuramoto" ou "neutral"*

*"perturbation": {*

*"type": "choc",*

*"t0": 25,*

*"amplitude": 1.0*

*}*

*},*

*"strates": [*

*{*

*"A0": 1.0,*

*"f0": 1.0,*

*"alpha": 0.5,*

*"beta": 1.0,*

*"k": 2.0,*

*"x0": 0.5,*

*"w": [0.1, 0.1, ...] // poids de connexion*

*}*

*// ... autres strates, si N > 1*

*],*

*"spiral": {*

*"phi": 1.618,*

*"epsilon": 0.05,*

*"omega": 0.1,*

*"theta": 0.0*

*},*

*"exploration": {*

*"recurrence\_window": [1, 10, 100],*

*"fractal\_threshold": 0.8,*

*"detect\_fractal\_patterns": true,*

*...*

*},*

*"to\_calibrate": {*

*"variance\_d2S": 0.01,*

*"entropy\_S": 0.5,*

*"gamma\_n": 1.0,*

*"env\_n": "gaussienne",*

*"sigma\_n": 0.1*

*}*

*}*

**Organisation possible :**

• system : paramètres globaux (taille, durée, seed, type de simulation, perturbation)

• strates : liste d’objets, un par strate

• spiral : paramètres de la dynamique spiralée

• to\_calibrate : seuils et formes à ajuster ou explorer

Chaque paramètre de seuil ajouté lors de la phase 2 doit bien être ajouté dans “to\_calibrate” ou logué, sans exception.

Les motifs fractals détectés sont caractérisés par leur échelle (fenêtre), la force du motif (corrélation > threshold), le pas de temps d’apparition et la métrique concernée.

**Validation automatique du config.json (script/checklist)**

Avant chaque simulation, un script de validation (validate\_config.py par exemple) vérifie la cohérence et la complétude du fichier :

**Exemple de checklist :**

* Toutes les clés principales (system, strates, spiral, to\_calibrate) sont présentes
* N et T sont strictement positifs, seed est défini
* Le nombre d’objets dans strates = N
* Chaque strate a toutes les clés attendues (A0, f0, alpha, beta, k, x0, w)
* Les paramètres spiralés (phi, epsilon, omega, theta) existent
* Tous les seuils ou formes listés dans to\_calibrate sont définis

**Pseudo-script de validation :**

*import json*

*def validate\_config(config\_path):*

*with open(config\_path, 'r') as f:*

*cfg = json.load(f)*

*assert 'system' in cfg and 'strates' in cfg and 'spiral' in cfg and 'to\_calibrate' in cfg*

*assert cfg['system']['N'] > 0 and cfg['system']['T'] > 0*

*assert len(cfg['strates']) == cfg['system']['N']*

*for s in cfg['strates']:*

*for key in ['A0', 'f0', 'alpha', 'beta', 'k', 'x0', 'w']:*

*assert key in s*

*for key in ['phi', 'epsilon', 'omega', 'theta']:*

*assert key in cfg['spiral']*

*# Check all expected calibrations are present*

*for key in ['variance\_d2S', 'entropy\_S', 'gamma\_n', 'env\_n', 'sigma\_n']:*

*assert key in cfg['to\_calibrate']*

*print("Config validation: OK")*

**À noter :**

* Il est possible d’adapter la liste des paramètres selon les évolutions de la feuille de route.
* Toute valeur manquante ou incohérente doit être corrigée avant lancement du run.

**Validation spécifique de la détection de motifs fractals**

Si la clé "detect\_fractal\_patterns" de exploration est True dans le config.json :

* Vérifier que metrics est une liste non vide
* Vérifier que window\_sizes est une liste non vide
* Vérifier que fractal\_threshold est bien un float dans l’intervalle ]0, 1[
* Si tout est valide, afficher/loguer “Fractal motif detection config: OK”

Exemple Python :

*if cfg["exploration"].get("detect\_fractal\_patterns", False):*

*assert isinstance(cfg["exploration"].get("metrics", []), list)*

*assert len(cfg["exploration"].get("metrics", [])) > 0*

*assert isinstance(cfg["exploration"].get("window\_sizes", []), list)*

*assert len(cfg["exploration"].get("window\_sizes", [])) > 0*

*th = cfg["exploration"].get("fractal\_threshold", 0.8)*

*assert isinstance(th, (float, int)) and 0 < th < 1*

*print("Fractal motif detection config: OK")*

**4. Peut-on encore comparer avec des oscillateurs de Kuramoto ?**

Oui, et ça renforce la rigueur, car :

* Kuramoto est un modèle de synchronisation canonique
* Comparer FPS à Kuramoto = montrer ce que la FPS permet en plus (stabilité spiralée, harmonisation douce, émergence complexe, résilience)

**Comment définir le groupe contrôle ?**

* Kuramoto : oscillateurs couplés uniquement par phase, pas d’amplitude ni de feedback spiralé
* FPS : phase + amplitude adaptative + feedback + spiralisation

On crée deux simulations avec même Iₙ(t) et mêmes perturbations, et on observe la réponse de S(t), E(t), etc.

**5. Ajout d’un mode “Contrôle”**

Pour lancer une version avec FPS complète et une version “neutre” (pas de feedback, pas de G, pas de spiralisation), pour prouver l’émergence spécifique.

**6. Groupe Contrôle (Oscillateurs de Kuramoto)**

**But**

Le groupe contrôle repose sur le modèle d’oscillateurs de Kuramoto : chaque unité évolue en phase uniquement, sans amplitude adaptative ni rétroaction spiralée. Ce contrôle sert à valider la spécificité de la FPS en comparant ses performances à un modèle bien connu de synchronisation.

**Équations et implémentation**

* Équation classique Kuramoto :

\frac{d\varphi\_i}{dt} = \omega\_i + \frac{K}{N} \sum\_{j=1}^N \sin(\varphi\_j - \varphi\_i)

* + Paramètres : K = 0.5, N = 20, \omega\_i \sim U[0, 1]
* Métriques loguées (identiques à FPS) :
  + S(t) = \frac{1}{N} \sum\_{n} \exp(i\varphi\_n(t)) (ordre global)
  + C(t) = \frac{1}{N} \sum\_{n=1}^{N-1} \cos(\varphi\_{n+1}(t) - \varphi\_n(t))
  + Effort (optionnel, nul ici)
  + Coût CPU (cpu\\_step(t))
* Structure de code :
  + Option dans config.json : "mode": "Kuramoto"
  + Même durée, mêmes seeds et mêmes perturbations appliquées que pour la FPS
* Comparaison :

À chaque run, on compare S(t), C(t), effort, coût CPU entre FPS et Kuramoto pour détecter l’apport de la régulation spiralée.

**Mode “neutre”**

Pour aller plus loin, on peut ajouter un troisième mode :

* Mode “neutral” : oscillateurs sans feedback ni spiralisation, amplitude et phase fixes, pour valider l’émergence spécifique à FPS.

**7. Exporter tous les logs à chaque pas de t**

**En CSV ou HDF5 pour :**

* Rejouer une simulation
* Entraîner un modèle sur le comportement de la FPS
* Prouver empiriquement l’évolution (falsifiabilité : rejouer et contredire)

**8. Graines, seeds.txt, nœuds**

**Graines / seeds :**

Essentiel pour la reproductibilité scientifique :

* Ajouter random.seed(SEED) et np.random.seed(SEED)
* Sauvegarder chaque SEED dans un seeds.txt pour chaque run

**Nœuds :**

Par “nœuds” on parle des ID des strates ou de points d’émergence d’information :

* On garde des identifiants uniques par strate
* On peut faire apparaître les nœuds qui ont initié un événement (ex. : un pic d’effort)

**seeds.txt ?**

On peut automatiser :

*with open("seeds.txt", "a") as f:*

*f.write(f"Simulation {id} | SEED = {SEED}\n")*

Conclusion : oui, on conserve graines + SEED + identifiants des strates, car c’est essentiel pour refaire les expériences et vérifier les hypothèses.

**9. Détails:**

* **Matrice de correspondance critère ↔ termes FPS**

Cette matrice fait le lien entre chaque critère empirique et ses termes mathématiques principaux. Elle permet de garantir que chaque métrique loguée et chaque raffinement sont bien enracinés dans la structure FPS, évitant tout décalage entre théorie et pratique.

|  |  |
| --- | --- |
| **Critère empirique** | **Termes mathématiques principaux observés** |
| Stabilité | S(t), Δfₙ(t), C(t), φₙ(t), L(t) |
| Régulation | Fₙ(t), G(Eₙ - Oₙ), γ(t), Aₙ(t), Gₙ(x,t) |
| Fluidité | γₙ(t), σ(x), fₙ₊₁(t), envₙ(x,t), μₙ(t) |
| Résilience | Aₙ(t), G(x,t), ΔAₙ(t), Δfₙ(t), effort(t) |
| Innovation | A\_spiral(t), Eₙ(t), r(t), S(t) (analyse spectrale) |
| Coût CPU | cpu\_step(t) (Δt/N), optionnellement en lien avec S(t) ou N |
| Effort interne | effort(t),d**\_**effort**/**dt |
| Effort transitoire | Dérivée locale d’effort(t) à court terme (d\_effort/dt) |
| Effort chronique | Moyenne glissante haute de effort(t) sur 80% du temps |

* **Interaction entre effort et coût CPU :**

– Effort(t) quantifie l’intensité d’ajustement adaptatif du système à chaque pas, tandis que cpu\_step(t) mesure la charge réelle en temps de calcul.

– Les deux sont corrélés lorsque l’adaptabilité interne du modèle entraîne une complexification computationnelle ; mais ils peuvent diverger selon l’optimisation du code ou l’architecture matérielle.

– Un effort chronique élevé tend à générer une surcharge CPU, tandis qu’un effort transitoire élevé peut provoquer des pics isolés.

– Leur comparaison systématique (via corrélation et visualisation conjointe) permet d’identifier les phases où la plasticité du système impacte fortement la charge computationnelle et d’optimiser le modèle ou l’implémentation.

Il faut tracer ces corrélations et loguer les interactions pendant les runs.

**En résumé :**

– Effort(t) = « combien le système doit s’ajuster »

– cpu\_step(t) = « combien ça coûte vraiment à la machine »

– Leur corrélation n’est pas systématique mais doit être analysée pour optimiser et comprendre le modèle.

– Un effort élevé non suivi d’un cpu élevé = code efficace.

– Un effort faible mais cpu élevé = code à optimiser.

* **Légende / icônes pour la grille empirique (1 à 5)**

Pourquoi :

C’est une idée d’ergonomie visuelle, pour les notations subjectives.

Par exemple :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Note** | **Étiquette** | **Icône** | **Couleur** | **Signification générale** |
| 1 | Rupture / Chaotique | ✖ | Rouge vif | Rupture de cohérence, surcharge, désaccord systémique |
| 2 | Instable / Mal accordé | ▲ | Orange | Réponse erratique, feedback inefficace |
| 3 | Acceptable / Fonctionnel | ● | Jaune | Stabilité de base mais encore brut ou peu harmonieux |
| 4 | Harmonieux | ✔ | Vert doux | Alignement en spirale partiel, bonne plasticité |
| 5 | FPS-idéal | ∞ (spirale) | Bleu doré | Système auto-régulé, fluide, résilient, et innovant |

On peut l’implémenter dans un notebook HTML plus tard, ou l’utiliser pour générer une visualisation claire avec matplotlib, plotly, ou une interface dédiée.

**Est-ce toujours falsifiable ?**

À condition de respecter trois règles :

**Règle 1 :**

**Un changement à la fois**

→ Si on teste une nouvelle forme de G\_sinc, on garde w₁–w₃ figés,

ou bien on documente leur variation dans le JSON.

**Règle 2 :**

**Comparer aux graines de référence**

→ On réutilise les mêmes seeds pour chaque expérience.

**Règle 3 :**

**Décrire la forme mathématique et les seuils**

→ Pas de fonction implicite ou floue ; tout doit être écrit, paramétré.

Si on suit ça : chaque ajout FPS reste rigoureux.

**Checklist Finale FPS : Avant, Pendant et Après chaque Simulation**

A. Avant le run

* config.json validé : toutes les clés présentes, pas de valeur incohérente, nombres strictement positifs, seeds OK
* strates[] : nombre d’objets = N, chaque strate a A0, f0, alpha, beta, k, x0, w
* spiral : phi, epsilon, omega, theta définis
* to\_calibrate : variance\_d2S, entropy\_S, gamma\_n, env\_n, sigma\_n présents
* exploration/fractale : si active, metrics/window\_sizes non vides, 0 < fractal\_threshold < 1, message logué
* tests unitaires (test\_fps.py) tous passés (statique/dynamique)
* SEED sauvegardé dans seeds.txt

B. Pendant le run

* Logs CSV/HDF5 produits à chaque pas : S(t), A\_mean(t), f\_mean(t), effort(t), cpu\_step(t), C(t), E(t), L(t)
* Logs optionnels activés si “phase 1” ou analyse exploratoire (variance\_d2S, entropy\_S, effort\_status…)
* Graines, seeds et identifiants de strates logués à chaque simulation
* **Sécurité, robustesse et gestion des erreurs (à chaque step du run)**
  + Vérification try/except sur chaque compute\_\* (gestion d’exception localisée)
  + Log automatique des erreurs dans error\_<run\_id>.log (toute exception ou warning critique)
  + Mode de récupération si crash mid-simulation (sauvegarde d’état temporaire/reprise possible)
  + Stratégie de stockage adaptative :

· Si N > 10, utiliser HDF5 automatiquement

· Si T > 1000, compression des logs activée

· Backup automatique de tous les fichiers logs tous les 100 pas

C. Après le run

* explore.py exécuté sur les logs pour : détection émergences, anomalies, bifurcations, événements atypiques, détection automatique des motifs fractals (auto-similaires) dans S(t), C(t), effort(t), etc. (méthode : analyse multi-échelles, config “fractal\_detection”), log structuré des motifs fractals détectés (fractal\_events\_<run\_id>.csv), section dédiée dans le rapport exploration\_report\_<run\_id>.md
* “fractal\_events\_<run\_id>.csv” et section “Motifs fractals détectés” du “exploration\_report\_<run\_id>.md” produits et sauvegardés dans le dossier daté de la simulation.
* Rapport markdown généré (exploration\_report\_<run\_id>.md), logs événements (emergence\_events\_<run\_id>.csv)
* Changelog.txt mis à jour : toute modification de paramètre, seuil, forme, mode, version, justification datée
* Comparaison groupe contrôle (Kuramoto, neutral) systématique : mêmes seeds, mêmes perturbations, mêmes métriques
* Corrélation effort/cpu tracée, et tout écart >3σ logué (mode alerte)
* Critères empiriques (fluidité, stabilité, etc.) évalués : tout franchissement de seuil → raffinement automatique, logué
* Paramètres “exploratoires” identifiés dans le log ou le rapport (pour la traçabilité scientifique)
* **Validation et intégrité des outputs**
  + Vérifier que S(t) reste dans les bornes physiques attendues (assertions, tests limites)
  + Alerte immédiate si NaN ou inf détecté dans n’importe quelle métrique loguée (flag dans le rapport d’exploration, option de relancer ou annuler le run)
  + Checksum des fichiers de logs calculé et stocké (validation de l’intégrité avant archivage)
* Export logs et résultats dans un dossier daté + sauvegarde du config.json et du seeds.txt correspondant

D. Juste avant publication ou envoi

* Résumé pipeline en 5 lignes (si pas déjà en tête du doc)
* Vérification cross-check entre grille empirique et matrice critères ↔ termes mathématiques
* Lien vers la version de code/commit dans chaque rapport si possible
* Tests unitaires relancés une fois pour la version publiée
* README ou note méthodologique brève sur la version du code/paramètres pour auditabilité

**Points à surveiller lors de la mise en œuvre**

* Bien automatiser l’export et la sauvegarde systématique de tous les logs, seeds, et configs (pour garantir la réplicabilité intégrale).
* Penser à l’ergonomie pour la visualisation de la grille empirique (subjective) dans le notebook ou une future UI (pertinent si diffusion large).
* Rester strict sur les changements : chaque évolution doit être loguée dans le changelog.txt avec justification