## 1. Objectif et hypothèse principale

Nous voulons tester, sur un réseau d'oscillateurs de Kuramoto, si la règle FPS (c'est-à-dire un couplage K(t), une amplitude A(t) et une latence γ(t) gouvernés par l'indicateur de confort C(t) ) augmente simultanément :

- (i) la fluidité tout en maintenant une cohérence stable
- (ii) la résilience après un choc de phase
- (iii) l'exploration entropique (capacité d'innovation)

le tout pour un surcoût computationnel inférieur à +20 % par rapport au modèle témoin à K constant. Le temps CPU moyen par pas ne devra pas dépasser 1,2 × (le temps témoin) sur la même machine et la même version de Python.

L'hypothèse sera considérée confirmée si, sur au moins deux de ces trois axes, la différence moyenne FPS – témoin est significative (intervalle de confiance à 95 % excluant 0) tout en respectant la contrainte de coût.

## 2. Modèles comparés

Témoin : réseau Kuramoto classique (N = 30, couplage fixe $K_0$ = 1, pas de boucle FPS).

FPS : même réseau mais :

- G(x) est la combinaison pondérée de trois noyaux — tanh, sinc, spirale logarithmique.
- Le couplage K(t), l'amplitude A(t) et la latence γ(t) s'ajustent via C(t) comme dans le prototype de code.

Tous les autres paramètres (fréquences naturelles, pas de temps, durée de simulation) sont identiques entre les deux conditions.

## 3. Paramètres fixés avant l'exécution

- Nombre d'oscillateurs N = 30.
- Durée T = 200 s, pas dt = 0,1 s → 2000 pas.
- Distribution initiale des phases : uniforme [0, 2π].
- Fréquences naturelles $\omega_i$ tirées d'une loi normale μ = 2π, σ = 0,1 × 2π.
- Amplitudes :
    - $A_0$ = 1,0 (valeur de base).
    - α_A = 0,1 (facteur de lissage).
- Couplage : K_max = 5, α_K = 0,1.
- Latence : γ_initiale = 1,5, α_γ = 0,1.
- Indicateur de confort :
    - $C(t) = \tanh(w_1 H\_norm + w_2 (1 - R) + w_3 |dR/dt|)$ avec $w_1 = w_2 = w_3 = 1/3$.
- Perturbation résilience : à t = 100 s, décalage de phase de π/2 appliqué à 20 % des oscillateurs.

### 3 bis. Méthode d'intégration

##### *Intégrateur numérique et gestion des phases*

*La dynamique $\dot{\phi} = f(\phi)$ est intégrée par **Euler explicite** :*

$$\phi_i \leftarrow \phi_i + dt \cdot f_i(\phi)$$

*Aucune autre méthode (par exemple Runge–Kutta) n'est utilisée dans cette étude.*

*Après chaque pas, chaque phase est normalisée dans l'intervalle [0, 2π[ :*

$$\phi_i \leftarrow (\phi_i \bmod 2\pi)$$

*En Python : `phi = np.mod(phi, 2*np.pi)`.*

*Cette opération garantit des phases bornées et des mesures d'entropie stables.*

## 4. Variables mesurées

- Cohérence globale R(t) et son aire intégrale $\int_0^T R(t)\,dt$.
- Fluidité : écart-type de dR/dt.
- Entropie de phase H_norm(t) puis moyenne temporelle ⟨H_norm⟩.
- Temps de retour t_recovery pour que R et A reviennent dans ±5 % de leur valeur pré-choc. R_ref et A_ref sont la moyenne de R(t) et A(t) sur la fenêtre [95 s, 100 s] juste avant le choc.
- Coût computationnel :
    - (i) Temps CPU moyen par pas : obtenu en chronométrant toute la boucle principale (time.perf_counter) puis en divisant par « steps ».
    - (ii) Coût_effort : Cost_effort = (1 / (N · steps)) · Σ_{k=1}^{N} Σ_{t=1}^{steps} | K(t) · G_comb(Δ_k(t)) |
        - où N = 30 et steps = 2000.

## 5. Plan expérimental

1. Générer 30 graines aléatoires (indices 0 – 29) : les 30 entiers aléatoires sont écrits dans un fichier seeds.txt poussé dans le dépôt ; le notebook lit ce fichier, puis fixe np.random.seed(seed) avant chaque run.
2. Exécuter la condition témoin puis la condition FPS pour chaque graine.
3. Sauvegarder pour chaque run un fichier CSV contenant les traces de R, dR/dt, H_norm, A, γ, K, coût.
4. Calculer les métriques listées §4 → un tableau (30 lignes × métriques).
5. Analyse : différence FPS – témoin pour chaque graine, bootstrap (10 000 resamples) sur la moyenne, en reportant l'intervalle de confiance à 95 %.

## 6. Diffusion et réplication

- Dépôt GitHub public créé avant l'exécution : notebook, scripts, paramètres, SHA gelé.
- Données CSV brutes poussées après chaque run dans data/raw/.
- Rapport d'analyse (analysis.ipynb) générant les graphiques et les intervalles.
- Annonce postée sur un forum open-science avec invitation à re-lancer le notebook Colab. Deux validations indépendantes suffiront pour considérer la réplication atteinte.

## 7. Définitions opérationnelles des variables, paramètres et mesures

Ce chapitre dresse la liste exhaustive des objets mathématiques, des constantes numériques et des métriques statistiques utilisés dans le protocole Tests de la FPS. Chaque entrée comporte :

- la formule ou l'algorithme exact ;
- le fragment de code Python/NumPy minimal correspondant ;
- une description fonctionnelle sans métaphore.

---

### 7.1 Variables d'état simulées

| Symbole | Formule / Algorithme | Code minimal | Description fonctionnelle |
|---|---|---|---|
| $\phi_k(t)$ | Phase de l'oscillateur k à l'instant t. | phi[k] | Variable d'état centrale du modèle de Kuramoto. |

| | | | |
|---|---|---|---|
| R(t) | $begin:math:text$ R(t)=\Bigl | \frac1N\sum_{k=1}^{ N}e^{iϕ_k(t)}\Bigr | $end:math:text$ |
| dR/dt | (R(t)-R(t-dt))/dt | dR = (R - prev_R)/dt | Variation instantanée de la cohérence. |
| H_norm(t) | Entropie de Shannon normalisée : H/ \log B avec B le nombre de classes. | voir code | Diversité de répartition des phases. |
| C(t) | $begin:math:text$ \tanh\!\bigl(w_1 H_\text{norm}+w_2( 1-R)+w_3 | dR/dt | \bigr) $end:math:text$ |
| A(t) | A(t)=A_0\cdot\frac{1 +C(t)}{2} lissé : A←α_A A_{t-1}+(1-α_A)A_\t ext{target} | code ci-dessous | Amplitude adaptative. |
| K(t) | Objectif K_\text{target}=1+C( t) puis lissage K←α_K K_{t-1}+(1-α_K)K_\t ext{target}; borné à K_max. | code | Couplage adaptatif. |
| γ(t) | Objectif γ_\text{target}=1+(1- C(t)) puis lissage γ←α_γ γ_{t-1}+(1-α_γ)γ_\te xt{target}. | code | Latence expressive. |

| | | | |
|---|---|---|---|
| G_comb(Δ) | w_tG_\text{tanh}(Δ) +w_sG_\text{sinc}(Δ )+w_lG_\text{log}(Δ) avec Δ = E–ϕ. | code | Noyau composite appliqué à chaque oscillateur. |

Fragments de code pour A, K, γ :

A_target  = (1 + C) / 2

A  = alpha_A * A_prev + (1 - alpha_A) * A_target

K_target  = 1 + C

K  = alpha_K * K_prev + (1 - alpha_K) * K_target

K  = min(K, K_max)

gamma_target = 1 + (1 - C)

gamma  = alpha_gamma * gamma_prev + (1 - alpha_gamma) * gamma_target

---

**7.2 Fonctions noyaux G**

| Nom | Expression | Code NumPy | Domaine |
|---|---|---|---|
| G_tanh(x) | \tanh(x) | np.tanh(x) | Saturation douce. |
| G_sinc(x) | \operatorname{sinc}(x/π) | np.sinc(x/np.pi) | Mémoire centrale étalée. |

| | | | |
|---|---|---|---|
| G_logspiral(x) | $begin:math:text$ \operatorname{sign}(x)\log(1+ | x | )\sin(x) $end:math:text$ |

Les pondérations contextuelles sont normalisées :

$$w_t=|dR/dt|;\;w_s=H_\text{norm};\;w_l=1-R;\;w_i\leftarrow w_i/\sum_j w_j.$$

---

### 7.3 Constantes du protocole

| Nom | Valeur | Rôle |
|---|---|---|
| N | 30 | Nombre d’oscillateurs. |
| dt | 0,1 s | Pas de temps. |
| T | 200 s | Durée totale de la simulation. |
| $A_0$ | 1,0 | Amplitude maximale. |
| α_A | 0,1 | Lissage d’A(t). |
| K_max | 5 | Limite supérieure du couplage. |
| α_K | 0,1 | Lissage de K(t). |

| | | |
|---|---|---|
| γ_init | 1,5 | Valeur initiale de γ. |
| α_γ | 0,1 | Lissage de γ(t). |
| $w_1, w_2, w_3$ | 1/3 chacun | Pondérations de C(t). |
| Perturbation | Δϕ = π/2 sur 20 % des nœuds à t = 100 s | Test de résilience. |

---

**7.4 Métriques d’évaluation**

| Nom | Définition précise | Calcul |
|---|---|---|
| Aire_R | \int_{0}^{T} R(t)\,dt (approx. somme discrète dt·R[t]) | R.sum()*dt |
| Fluidité | Écart-type de dR/dt sur [0,T] | np.std(dR_trace) |
| Entropie moyenne | Moyenne temporelle de H_norm(t) | H_trace.mean() |
| t_recovery | Première date ≥ perturbation où | R(t)-R_ref |
| Coût_CPU | Temps processeur/pas (profilage) | time.perf_counter() |
| Coût_effort | Moyenne de | K·G_comb |

---

**7.5 Contrôles de validité automatisés**

- Bornes :

  assert np.all((A_trace>=0)&(A_trace<=A0))

  assert np.all((C_trace>=-1)&(C_trace<=1))

- Variation maximale :

  assert np.max(np.abs(np.diff(A_trace))) <= (1-alpha_A)+1e-6

- Intervalle de confiance : bootstrap (10 000 tirages) sur la différence FPS–témoin ; l'IC95 % est [P_{2.5},P_{97.5}]. Si 0 ∉ IC, l'effet est déclaré significatif.

---

**7.6 vulgarisation**

**a. Ce qu'on simule : 30 oscillateurs de Kuramoto**

Un oscillateur est un angle ϕ (sa phase) qui tourne autour du cercle.

Dans Kuramoto :

*dphi_k/dt = ω_k + K/N * Σ_j sin(ϕ_j - ϕ_k)*

- ω_k : vitesse naturelle (on la tire d'une petite gaussienne autour de 2π).
- K : force de couplage, la même pour tous au même instant.
- N : nombre d'oscillateurs (ici 30).

**b. Pas de temps et boucle principale**

*import numpy as np, time, math*

*# ----------------- paramètres fixes -----------------*

```
N       = 30
dt      = 0.1         # 0,1 s
T       = 200         # 200 s au total
steps   = int(T/dt)
omega   = np.random.normal(2*np.pi, 0.1*2*np.pi, N)  # vitesses
phi     = np.random.uniform(0, 2*np.pi, N)          # phases initiales

# paramètres FPS
A0      = 1.0
alpha_A = 0.1
K_max   = 5.0
alpha_K = 0.1
gamma   = 1.5         # latence expressive (on l'utilisera plus tard)
alpha_g = 0.1

w1 = w2 = w3 = 1/3      # pondérations du confort

# stockage pour analyse
R_trace, A_trace, C_trace = [], [], []
```

**c. Mesures instantanées**

c.1 Cohérence R(t)

```
def coherence(angles):
    return np.abs(np.mean(np.exp(1j*angles)))
```

c.2 Variation dR/dt

On garde prev_R, on calcule la différence et on divise par dt.

c.3 Entropie normalisée H_norm(t)

Un moyen simple : on découpe le cercle en 36 bacs de 10 °, on compte combien d'oscillateurs par bac, puis on calcule l'entropie de Shannon ; enfin on divise par log(B) pour la normaliser entre 0 et 1.

```
def phase_entropy(angles, bins=36):
    counts, _ = np.histogram(angles, bins=bins, range=(0, 2*np.pi))
    p = counts[counts>0] / counts.sum()
    H = -np.sum(p * np.log(p))
    return H / np.log(bins)
```

**d. Indicateur de confort C(t)**

```
def comfort(H, R, dRdt):
    raw = w1*H + w2*(1-R) + w3*abs(dRdt)
    return np.tanh(raw)          # borné dans [-1, +1]
```

Pourquoi ces trois composantes ?

- H monte quand les phases sont variées → c'est bon pour l'exploration.
- 1-R monte quand la cohérence est faible → ça évite trop de rigidité.
- |dR/dt| monte quand R bouge brutalement → ça détecte les secousses.

On fait la moyenne, puis tanh écrase les valeurs extrêmes pour que C reste borné.

**e. Amplitude A(t), Couplage K(t), Latence γ(t)**

1. Cible à atteindre en fonction de C(t)

```
A_target = A0 * (1 + C) / 2          # 0 à A0
K_target = 1 + C                     # 0 à 2
K_target = min(K_target, K_max)      # borne haute
gamma_target = 1 + (1 - C)           # 1 à 2
```

2. Lissage exponentiel (c'est le « lissage » ou α dont on voit les constantes ; il évite les à-coups en faisant une moyenne pondérée avec l'état précédent)

*A = alpha_A * A_prev + (1-alpha_A) * A_target*

*K = alpha_K * K_prev + (1-alpha_K) * K_target*

*gamma = alpha_g * gamma_prev + (1-alpha_g) * gamma_target*

- alpha_A = 0.1 → l'amplitude atteint ≈ 63 % de sa cible après ~10 pas.
- Même logique pour K et γ.

**f. Noyau G_comb**

À chaque pas, pour chaque oscillateur, on applique un noyau composite :

```
def G_tanh(x):  return np.tanh(x)
def G_sinc(x):  return np.sinc(x/np.pi)          # sinc normalisée
def G_log(x):   return np.sign(x)*np.log1p(abs(x))*np.sin(x)

def G_comb(delta, w_t, w_s, w_l):
   return (w_t*G_tanh(delta) +
           w_s*G_sinc(delta) +
           w_l*G_log(delta))
```

Les poids sont recalculés à chaque pas :

*w_t = abs(dRdt)*

*w_s = H*

*w_l = 1 - R*

*w_sum = w_t + w_s + w_l*

*w_t, w_s, w_l = w_t/w_sum, w_s/w_sum, w_l/w_sum*

**g. Mise à jour des phases**

```
for step in range(steps):
    # 1) mesures
    R = coherence(phi)
    if step == 0:
        dRdt = 0.0
    else:
        dRdt = (R - prev_R) / dt
    H = phase_entropy(phi)
    C = comfort(H, R, dRdt)

    # 2) boucles adaptatives
    A_target   = A0 * (1 + C) / 2
    K_target   = min(1 + C, K_max)
    gamma_target = 1 + (1 - C)

    A     = alpha_A * A_prev + (1-alpha_A) * A_target
    K     = alpha_K * K_prev + (1-alpha_K) * K_target
    gamma = alpha_g * gamma_prev + (1-alpha_g) * gamma_target

    # 3) noyau pondéré
```

```
w_t = abs(dRdt)

w_s = H

w_l = 1 - R

w_sum = w_t + w_s + w_l

w_t, w_s, w_l = w_t/w_sum, w_s/w_sum, w_l/w_sum


# 4) champ global E(t) = moyenne des phases

E = np.mean(phi)


# 5) mise à jour des phases

for k in range(N):

    delta = E - phi[k]

    G     = G_comb(delta, w_t, w_s, w_l)

    phi[k] += dt * (omega[k] + K/N * A * np.sin(G + delta))

    phi[k] %= 2*np.pi        # remet l'angle dans [0, 2π[


# 6) perturbation à mi-parcours

if math.isclose(step*dt, 100, abs_tol=dt/2):

    idx = np.random.choice(N, size=int(0.2*N), replace=False)

    phi[idx] += np.pi/2        # choc


# 7) stocker pour analyse

R_trace.append(R)

A_trace.append(A)

C_trace.append(C)

prev_R, A_prev, K_prev, gamma_prev = R, A, K, gamma
```

**h. Tests de cohérence automatiques**

Juste après la boucle :

*assert np.all((np.array(A_trace) >= 0) & (np.array(A_trace) <= A0)), "A(t) hors bornes"*

*assert np.all((np.array(C_trace) >= -1) & (np.array(C_trace) <= 1)), "C(t) hors bornes"*

*assert np.max(np.abs(np.diff(A_trace))) <= (1 - alpha_A) + 1e-6, "Variation A trop brusque"*

Ces trois assertions garantissent que les formules sont bien respectées.

**j. Pourquoi ces constantes ?**

- A0 = 1 : amplitude maximale normalisée ; on ne veut pas dépasser l'enveloppe naturelle.
- α_A, α_K, α_g = 0,1 : règle empirique « temps de réponse ≈ 10 pas ».
- K_max = 5 : borne haute pour éviter la synchronisation forcée irréaliste.
- w1 = w2 = w3 : on donne la même importance à diversité, cohérence et douceur de variation — c'est un choix neutre.

Ces valeurs sont fixées avant l'expérience ; si une est changée après avoir vu les résultats, il faut ouvrir un « change log » et relancer toutes les graines. C'est ce qui rend le test falsifiable.

**k. Que faut-il tester et comment ?**

1. Condition témoin : même code mais on met K = K0 constant, on supprime A(t), γ(t) et G_comb (on garde la même boucle d'intégration pour être équitables).
2. Condition FPS : exactement le code ci-dessus.

Pour chaque graine, on calcule :

- l'aire sous R(t)
- l'écart-type de dR/dt
- la moyenne de H_norm
- le temps de retour t_recovery
- le temps CPU par pas
- la moyenne de abs(K * G) (effort)

Ensuite on fait la différence FPS – témoin sur chacune de ces métriques, on stocke le vecteur de 30 différences, et on applique le bootstrap pour obtenir l'intervalle de confiance à 95 % :

```
from sklearn.utils import resample

diffs = fps_values - control_values        # 30 échantillons
boots = [np.mean(resample(diffs)) for _ in range(10000)]
ci_low, ci_high = np.percentile(boots, [2.5, 97.5])
print("Δ =", np.mean(diffs), "  IC95% =", [ci_low, ci_high])
```

Si [ci_low, ci_high] ne contient pas zéro, l'effet est statistiquement établi au seuil 5 %.

**Environnement d'exécution**

Tests réalisés sous :

- Python 3.10 ou 3.11
- NumPy ≥ 1.26
- SciPy ≥ 1.12
- scikit-learn ≥ 1.5

Les dépendances minimales sont listées dans requirements.txt (racine du dépôt) et doivent être installées via pip install -r requirements.txt avant toute exécution.

---

Toutes les équations et constantes ci-dessus sont désormais figées ; tout changement ultérieur devra faire l’objet d’une révision de protocole datée.

---

Ce document constitue le cadre officiel préalable ; toute modification ultérieure des paramètres ou du plan devra être historisée par un nouveau commit estampillé « Protocol-change-YYYY-MM-DD ».