

Chapitre 5 : Le Synergistic Connection Network (SCN) : Architecture Générale

Chapitre 5 : Le Synergistic Connection Network (SCN) : Architecture Générale	413
5.1. Introduction et Enjeux.....	415
5.1.1. Contexte du Chapitre	415
5.1.2. Objectifs Principaux.....	417
5.2. Vision d'Ensemble de l'Architecture SCN.....	422
5.2. Vision d'Ensemble de l'Architecture SCN.....	423
5.2.1. Notion de “Noyau” vs. “Modules Adjoints”	423
5.2.1.2. Modules	426
5.2.2. Cycle de Vie du SCN	431
5.2.3. Exemples d'Organisation	436
5.3. Structures de Données pour la Matrice (ω)	448
5.3. Structures de Données pour la Matrice ω.....	449
5.3.1. Dense vs. Sparse	449
5.3.2. Indexation et Accès Rapide	451
5.3.3. Synchronisation en Cas de Distribution	455
5.4.1. Séparation du Calcul de Synergie	458
5.4.2. Méthodes d'Implémentation	460
5.4.3. Gestion du Coût	466
5.5. Module de Mise à Jour ω et Inhibition/Contrôle.....	470
5.5.1. Rappels des Règles DSL.....	470
5.5.3. Saturation	485
5.5.4. Recuit Simulé ou Bruit	491
5.6. Interfaces Entrée-Sortie et Gestion en Temps Réel.....	504
5.6. Interfaces Entrée-Sortie et Gestion en Temps Réel.....	505
5.6.1. Ajouter / Retirer une Entité	505
5.6.2. Mise à Jour Périodique vs. Événementielle.....	507

5.6.3. Extraction de Clusters / Sous-Réseaux	510
5.7. Distribution, Scalabilité et Architecture Modulaire.....	516
5.7. Distribution, Scalabilité et Architecture Modulaire.....	517
5.7.1. SCN Distribué sur Plusieurs Nœuds	517
5.7.2. Mise en Place d'un "meta-SCN"	523
5.7.3. Ingénierie Logicielle	529
5.8. Sécurité, Fiabilité et Vérifications	538
5.8. Sécurité, Fiabilité et Vérifications	539
5.8.1. Vulnérabilités au Bruit ou à l'Attaque	539
5.8.2. Robustesse Face aux Pannes.....	544
5.8.3. Contrôle d'Intégrité	548
5.9. Exemples d'Implémentations et Études de Cas	554
5.9. Exemples d'Implémentations et Études de Cas	555
5.9.1. SCN Multimodal	555
5.10. Conclusion et Ouverture	587
5.10. Conclusion et Ouverture	588
5.10.1. Bilan du Chapitre.....	588
5.10.2. Lien vers Chapitres 6, 7, 8.....	589
5.10.3. Conclusion Générale.....	590

5.1. Introduction et Enjeux

5.1.1. Contexte du Chapitre

5.1.1.1. Rappeler le Parcours : Après Avoir Défini (Ch. 3) la Représentation des Entités et (Ch. 4) la Dynamique d'Auto-Organisation, on Aborde Maintenant la “Structuration” d'un SCN au Niveau “Architecture Globale”

Le **Deep Synergy Learning (DSL)** progresse selon un fil conducteur qui, jusqu'à présent, a posé les **fondations conceptuelles** (chapitres 2 et 3) et la **dynamique** de l'auto-organisation (chapitre 4). Ces acquis, s'ils permettent de comprendre *comment* un **Synergistic Connection Network (SCN)** calcule la **synergie** et met à jour ses pondérations $\omega_{i,j}$, doivent à présent être prolongés par une **réflexion** plus large sur la **structuration logicielle**. En effet, il ne suffit plus de connaître la formule de mise à jour additive ou la manière d'introduire une inhibition compétitive ; il faut également clarifier la **façon** dont on organise les divers modules et flux de données pour gérer efficacement un SCN de grande taille ou relié à d'autres systèmes.

Le **chapitre 3** a introduit la **représentation** des entités $\{\mathcal{E}_i\}$ et a montré comment traduire, dans un espace sub-symbolique (embeddings, vecteurs, signaux) ou symbolique (règles, ontologies), la notion de *contenu* d'une entité. Le **chapitre 4** s'est concentré sur la **dynamique** : il a détaillé la **mise à jour** $\omega_{i,j}(t + 1)$ et le rôle crucial des **paramètres** η, τ ou γ (inhibition), révélant comment s'organisent peu à peu des **clusters** cohérents. Cette logique d'**auto-organisation**, essentielle à la philosophie du DSL, est désormais bien comprise : la synergie $S(i, j)$ détermine la vitesse à laquelle la pondération $\omega_{i,j}$ grandit ou diminue, conduisant à la formation de blocs d'entités fortement connectées.

Toutefois, disposer d'équations ou d'algorithmes de mise à jour ne garantit pas, à lui seul, la **maintenabilité** et la **scalabilité** du système. Lorsque l'on souhaite intégrer un SCN dans une application ou un cadre de recherche complexe (pouvant impliquer un grand nombre d'entités, des flux continus de données, ou une interconnexion avec d'autres briques logicielles), il faut régler plusieurs questions d'**architecture**. Le **chapitre 5** s'attachera donc à expliquer **comment** aller au-delà de la simple dynamique *algorithmique* pour parvenir à un **environnement modulaire**, où les blocs de calcul (mise à jour ω , calcul de synergie, module d'inhibition, etc.) sont clairement découplés. Ce faisant, on pourra gérer la **persistance** des données (stockage de ω sur disque ou en mémoire distribuée), la **synchronisation** (threads multiples, double-buffer, etc.) et l'**interface** avec des frameworks externes (traitement multimodal, robotique, systèmes distribués).

Ainsi, ce **chapitre 5** étendra la démarche amorcée dans les chapitres précédents, en montrant **comment** on fait vivre un SCN dans un *code* réaliste. On y retrouvera le **calcul** de **S**, la **dynamique** de ω , et on soulignera la nécessité de séparer en **modules** indépendants tout ce qui relève de la synergie (entités, scoring), de la mise à jour ω (logique de l'auto-organisation), et des mécanismes transverses (inhibition, recuit, gestion des ressources). Par ailleurs, la question de l'**observabilité** (possibilité de visualiser l'évolution de la matrice ω et l'apparition de clusters) ou celle de la **concurrency** (threading, verrous) seront détaillées, puisqu'elles conditionnent la faisabilité d'un SCN à grande échelle.

Ce **parcours** illustre la progression naturelle du DSL. Après avoir étudié **quoi** calculer (chap. 3, la représentation des entités) et **comment** l'actualiser (chap. 4, la dynamique ω), on doit maintenant se pencher sur **comment** organiser *logiciellement* ces éléments, de sorte qu'ils demeurent robustes, évolutifs et faciles à interfacer. C'est précisément la finalité du **chapitre 5**, qui vient *après* la dynamique d'auto-organisation et qui va *ouvrir* sur les chapitres suivants consacrés aux **extensions** (multi-échelle, multimodalité, etc.).

5.1.1.2. Expliquer la Différence entre la “Dynamique” (Vue Algorithme/Math) et l'Architecture (Vue Ingénierie, Mise en Œuvre Concrète)

La distinction fondamentale entre la dynamique, telle qu'elle est présentée au niveau mathématique et algorithmique, et l'architecture, qui représente la concrétisation logicielle et l'ingénierie d'un système, constitue un axe majeur pour la conception d'un **Synergistic Connection Network (SCN)** opérationnel. Au niveau de la **dynamique**, l'accent est

mis sur les équations de mise à jour des pondérations $\omega_{i,j}(t+1)$ et sur l'analyse théorique de leur comportement. Par exemple, nous avons vu que la règle additive classique s'exprime par

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta (S(i,j) - \tau \omega_{i,j}(t)),$$

ou que, dans une variante multiplicative, la mise à jour peut être formulée comme

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) \exp(\eta [S(i,j) - \tau \omega_{i,j}(t)]).$$

Ces formules, ainsi que les paramètres associés (η pour le taux d'apprentissage, τ pour la décroissance, et éventuellement γ pour des mécanismes d'inhibition), définissent l'**essence** du comportement du SCN : ils précisent « quoi » calculer à chaque itération, comment la pondération évolue en fonction de la synergie $S(i,j)$ et comment les liens se renforcent ou se décroissent. Cette vue mathématique permet d'étudier la convergence, la formation de clusters, et d'identifier des phénomènes tels que les oscillations ou la multi-stabilité. Elle se traduit le plus souvent par un pseudo-code qui, dans sa forme la plus simple, itère sur les entités pour actualiser $\omega_{i,j}$ selon la formule ci-dessus, fournissant ainsi un cadre théorique précis pour la dynamique d'auto-organisation.

En revanche, la **vue architecturale** se situe à un niveau d'implémentation concrète. Ici, il s'agit de concevoir une **infrastructure logicielle** capable d'exécuter la dynamique décrite tout en assurant la **maintenance**, l'**extensibilité** et la **scalabilité** du système dans un environnement réel. Cette infrastructure doit spécifier « comment » on organise les différents **modules** du SCN, c'est-à-dire la séparation claire entre le module de calcul de synergie, le module de mise à jour des pondérations et le module d'inhibition ou de recuit, ainsi que la manière de stocker la matrice ω dans des structures de données adaptées (par exemple, sous forme dense ou sous forme sparse). En outre, la gestion de la **parallélisation** et de la **synchronisation** entre les threads ou processus qui effectuent la mise à jour de ω devient cruciale quand le nombre d'entités n est élevé, car la complexité temporelle de la mise à jour est de l'ordre de $\mathcal{O}(n^2)$.

L'architecture se doit également d'offrir une interface robuste pour interagir avec d'autres modules du système, par exemple pour recevoir les modifications de la synergie $S(i,j)$ lorsqu'elle évolue au fil du temps (par exemple, en cas de mise à jour des embeddings ou de modifications des règles logiques), ou pour transmettre la matrice ω à des modules d'extraction de clusters ou d'optimisation. Des choix techniques tels que le double-buffering (pour séparer lecture et écriture des données), la gestion asynchrone des mises à jour et l'utilisation de verrous (ou de mécanismes lock-free) pour garantir la cohérence globale sont autant de considérations qui relèvent de la vue architecturale.

Ainsi, la **dynamique** représente le “**quoi**” : elle détermine, via des formules mathématiques précises, comment les pondérations $\omega_{i,j}$ doivent évoluer en fonction des synergies et des paramètres de stabilisation. C'est une approche purement algorithmique, qui se focalise sur l'analyse théorique, les conditions de convergence et les comportements attendus (par exemple, la convergence vers $\omega_{i,j}^* = \frac{S(i,j)}{\tau}$ dans un cas stationnaire). La **vue architecturale**, quant à elle, répond au “**comment**” : elle concerne la structuration du code, la gestion des ressources, la modularisation, la persistance des données et l'intégration du SCN dans un système d'envergure. Cette approche permet de passer de la théorie abstraite à une application concrète, dans laquelle la dynamique mathématique est encapsulée dans un environnement logiciel modulaire, scalable et interopérable.

En résumé, la différence réside dans le fait que la **dynamique** se concentre sur la formulation et l'analyse des règles de mise à jour (par exemple, la formule additive ou multiplicative, les paramètres η , τ , et l'étude de la convergence ou des oscillations), tandis que l'**architecture** s'intéresse à la manière d'implémenter ces règles de façon efficace et fiable dans un système logiciel réel. Cette dernière inclut la séparation des responsabilités en modules indépendants, la gestion de la mémoire (stockage de la matrice ω), la parallélisation des calculs, et l'interface avec des systèmes externes. Le Chapitre 5 développera en profondeur ces aspects d'ingénierie pour fournir un “contenant” logiciel robuste qui encapsule toute la dynamique théorique exposée au Chapitre 4. Cela permet non seulement de garantir la stabilité et la performance du SCN, mais aussi d'assurer sa maintenabilité et son adaptabilité dans des environnements complexes et évolutifs.

5.1.2. Objectifs Principaux

Au sein de ce **Chapitre 5**, nous ne restons plus seulement au niveau des **équations** ou des **concepts** (décrits en chapitres 2, 3, 4), mais nous nous focalisons sur la **mise en œuvre** concrète d'un **SCN** (Synergistic Connection Network). Nous allons préciser comment **concevoir** et **organiser** l'ensemble des modules et des données nécessaires à son fonctionnement, en gardant toujours à l'esprit des objectifs tels que la **clarté**, la **robustesse**, la **scalabilité** et l'**extensibilité**. Les grandes thématiques couvertes dans cette section (5.1.2) incluent :

5.1.2.1. Discuter Comment Organiser le SCN en Modules (Calcul de Synergie, Mise à Jour ω , etc.)

La conception d'un **Synergistic Connection Network (SCN)** opérationnel ne se limite pas à l'implémentation de la dynamique mathématique de mise à jour des pondérations $\omega_{i,j}(t+1)$. Pour que le système puisse être déployé à grande échelle, intégré dans des applications réelles et maintenu dans le temps, il est impératif de l'organiser en **modules distincts**. Cette approche modulaire, qui relève du principe de « separation of concerns », permet de dissocier la logique algorithme/mathématique – qui décrit « ce qu'il faut calculer » – de l'architecture logicielle – qui répond à la question « comment le code est structuré et géré dans un environnement d'exécution complexe ». Nous détaillerons ci-après les différents modules ainsi que les avantages qu'apporte cette organisation.

A. Raison d'Être de la Modularisation

La dynamique d'un SCN repose sur des formules telles que

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta \left(S(i,j) - \tau \omega_{i,j}(t) \right)$$

ou sa variante multiplicative, qui définissent l'évolution de la matrice ω en fonction de la **synergie** $S(i,j)$ entre les entités \mathcal{E}_i et \mathcal{E}_j . Ces équations, ainsi que leurs paramètres η , τ et éventuellement γ (pour l'inhibition), décrivent le comportement dynamique du système. Cependant, pour rendre ce comportement exploitable en pratique, il faut structurer le code de manière à isoler les différentes responsabilités. La modularisation permet notamment de séparer le **calcul de synergie** – qui peut reposer sur des données hétérogènes telles que des embeddings ou des règles logiques – de la **mise à jour** effective des pondérations ω . Cette séparation offre une meilleure lisibilité du code, facilite le débogage et permet de remplacer ou d'améliorer chaque composant sans avoir à repenser l'ensemble du système. En outre, elle rend possible une configuration dynamique des paramètres et une gestion efficace des ressources, deux aspects cruciaux quand le nombre d'entités n devient important.

B. Composition des Modules et Communication

Dans une implémentation modulaire d'un SCN, plusieurs blocs fonctionnels sont identifiés, chacun jouant un rôle spécifique :

- **Module « Calcul de Synergie »** : Ce module est responsable de la détermination de la fonction $S(i,j)$ qui quantifie la proximité ou la complémentarité entre les entités \mathcal{E}_i et \mathcal{E}_j . Pour des données sub-symboliques, il peut s'agir d'une similarité vectorielle (par exemple, une similarité cosinus ou une fonction de noyau gaussien) et, dans un contexte hybride, ce module peut combiner des scores issus d'embeddings et des scores de compatibilité logique. La fonction peut être exprimée, par exemple, sous la forme

$$S_{\text{hybrid}}(i,j) = \alpha S_{\text{sub}}(\mathbf{x}_i, \mathbf{x}_j) + (1 - \alpha) S_{\text{sym}}(R_i, R_j),$$

où α ajuste l'importance relative des composantes sub-symbolique et symbolique. Ce module reçoit en entrée les données des entités (embeddings, règles, etc.) et renvoie un score numérique exploitable par les autres modules.

- **Module « Mise à Jour de ω »** : Ce bloc implémente la dynamique propre aux pondérations. Il applique la règle de mise à jour, qu'elle soit additive ou multiplicative, et intègre les paramètres de contrôle tels que le taux d'apprentissage η et le facteur de décroissance τ . Par exemple, en mode additif, la mise à jour est donnée par

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta \left(S(i,j) - \tau \omega_{i,j}(t) \right).$$

L'objectif de ce module est de transformer, à chaque itération, la matrice $\omega(t)$ en une nouvelle configuration $\omega(t+1)$, en respectant les règles de la dynamique auto-organisée. En encapsulant cette fonction dans un module dédié, on peut facilement expérimenter différentes variantes (par exemple, intégrer des mécanismes de recuit ou de clipping) sans affecter le calcul de $S(i,j)$.

- **Module « Inhibition »** : Afin d'éviter une croissance excessive ou non sélective des pondérations, ce module applique des politiques d'inhibition, telles que l'inhibition compétitive locale ou globale. La règle peut prendre la forme d'un terme additionnel, par exemple

$$-\gamma \sum_{k \neq j} \omega_{i,k}(t),$$

qui pénalise l'activation simultanée de nombreux liens par une même entité. La modularisation de cette fonction permet de l'activer ou de la désactiver aisément, voire de la remplacer par une stratégie de sparsification, en fonction des exigences de l'application.

- **Module « Observer et Extraction de Clusters »** : Ce module a pour rôle de surveiller la matrice ω au fil des itérations, d'extraire des indicateurs de performance (tels que la modularité ou l'indice de silhouette) et d'identifier les clusters émergents. Il peut s'agir d'un outil de visualisation en temps réel, similaire à TensorBoard, qui permet de suivre l'évolution de la dynamique et d'ajuster les paramètres en conséquence.

La communication entre ces modules s'effectue via des interfaces bien définies : le module de mise à jour reçoit le score $S(i,j)$ calculé par le module de synergie, et transmet la nouvelle matrice ω aux modules d'inhibition et d'observation. Cette séparation garantit que l'évolution d'un module n'impacte pas directement les autres, permettant ainsi une maintenance aisée et une évolutivité du système.

C. Flexibilité et Extensibilité de l'Architecture

L'un des grands avantages d'une architecture modulaire est sa **flexibilité**. En isolant chaque composant, il est possible de modifier ou de remplacer une partie du système sans réécrire l'ensemble du code. Par exemple, si l'on souhaite passer d'une formule additive à une mise à jour multiplicative, il suffit d'ajuster le module de mise à jour de ω tout en conservant intact le module de calcul de synergie. De même, l'activation d'un module de recuit stochastique ou l'introduction de techniques de sparsification peut se faire par simple configuration, via des plugins ou des paramètres dans le gestionnaire de la dynamique. Cette approche modulaire rend également le système **scalable** : dans le cas où le nombre d'entités n devient très grand, la matrice ω peut être gérée à l'aide de structures de données adaptées (par exemple, des matrices creuses ou des bases de données distribuées), et le traitement peut être parallélisé de manière efficace grâce à un découpage des tâches entre plusieurs threads ou machines. Enfin, l'interopérabilité est facilitée puisque chaque module communique via des API standardisées, ce qui permet d'intégrer le SCN dans des environnements hétérogènes (systèmes multimodaux, simulateurs de robotique, etc.).

D. Exemple d'Organisation Modulaire

Pour illustrer, considérons un exemple hypothétique où le SCN est organisé en quatre modules principaux, orchestrés par un moteur central :

- **SCNCore** : Ce composant central détient la matrice ω et gère le cycle d'itération global. Il initie la boucle de mise à jour et coordonne les appels aux différents modules.
- **SynergyModule** : Responsable du calcul de $S(i,j)$, il reçoit les données d'entrée (embeddings, règles) et produit le score de synergie. Ce module peut être configuré pour utiliser différentes fonctions de similarité selon le contexte.
- **UpdateModule** : Ce module applique la règle de mise à jour de ω . Par exemple, il peut utiliser la formule additive

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta \left(S(i,j) - \tau \omega_{i,j}(t) \right),$$

ou une variante multiplicative, selon les paramètres fournis. Il reçoit le score $S(i, j)$ du SynergyModule et intègre les valeurs η et τ pour produire la nouvelle matrice.

- **InhibitionModule** : En tant que composant optionnel, il s'occupe d'appliquer des stratégies d'inhibition ou de sparsification, par exemple en limitant la somme des pondérations par ligne ou en imposant un clipping sur les valeurs extrêmes.
- **ObserverModule** : Ce module effectue des analyses de la matrice ω (par exemple, calcul de la modularité, détection de clusters) et fournit des visualisations ou des rapports pour surveiller la dynamique du SCN.

Chaque module communique avec SCNCore via des interfaces bien définies. Par exemple, le SynergyModule peut être appelé avec une fonction « computeSynergy(E[i], E[j]) » qui retourne $S(i, j)$, et le UpdateModule peut disposer d'une fonction « updateOmega(ω , S , η , τ) » qui produit $\omega(t + 1)$. Ce découpage facilite la maintenance et permet de tester indépendamment chaque composant.

E. Conclusion

L'organisation modulaire d'un SCN, telle que décrite dans ce point, permet de dissocier clairement la **dynamique mathématique** – qui s'exprime par des règles de mise à jour de ω en fonction de $S(i, j)$ – de l'**architecture logicielle** qui matérialise ces calculs dans un environnement réel. En isolant les modules de **calcul de synergie**, de **mise à jour de ω** et d'**inhibition**, ainsi que les outils d'**observation** et de **persistance**, on obtient un système **flexible, scalable et interopérable**. Cette structure permet non seulement de modifier aisément des aspects internes du SCN (comme le passage d'une règle additive à une règle multiplicative), mais aussi d'intégrer le SCN dans des environnements complexes, de gérer de grands volumes de données et de garantir une maintenance efficace. Le chapitre 5 approfondira ces concepts en décrivant l'architecture globale, les interfaces, et les stratégies de parallélisation et de persistance, assurant ainsi que la dynamique théorique du SCN se traduise par une infrastructure logicielle robuste et adaptable à divers cas d'usage.

5.1.2.2. Aborder la Persistance des Données, la Scalabilité, la Gestion d'Entités Hétérogènes dans un Seul Framework

Dans le contexte d'un **Synergistic Connection Network** (SCN), une fois que les modules essentiels de la dynamique – notamment le **calcul de la synergie** $S(i, j)$ et la **mise à jour** des pondérations $\omega_{i,j}(t + 1)$ – ont été définis, il apparaît indispensable de développer une infrastructure logicielle robuste qui intègre des mécanismes de **persistance**, assure la **scalabilité** pour un grand nombre d'entités et permette la gestion homogène d'entités hétérogènes dans un cadre unique. Cette section présente une approche intégrée, en insistant sur la formalisation mathématique des problèmes, sur les choix de structures de données et sur l'architecture logicielle qui facilitent l'implémentation opérationnelle d'un SCN.

A. Persistance des Données

La **persistance** des données constitue un enjeu majeur lorsque la matrice des pondérations, notée $\omega \in \mathbb{R}^{n \times n}$, peut devenir volumineuse, en particulier lorsque le nombre d'entités n croît de manière significative, entraînant une complexité mémoire de l'ordre de $\mathcal{O}(n^2)$. Dans un tel scénario, il est crucial d'enregistrer l'état du SCN à intervalles réguliers afin de permettre la reprise d'une simulation interrompue ou d'effectuer une analyse a posteriori de la dynamique. Par exemple, on peut définir une fonction de sauvegarde

$$\text{SaveState} : \omega(t) \rightarrow \text{Fichier (ou base de données)},$$

qui stocke l'état courant dans un format binaire ou au format HDF5. Cette approche permet non seulement de garantir la continuité des simulations en cas d'interruption, mais aussi d'enregistrer l'évolution de la dynamique pour une analyse ultérieure. En outre, lorsque la matrice ω est essentiellement creuse – c'est-à-dire que la majorité des liens $\omega_{i,j}$ restent faibles ou nuls après l'application de techniques de **sparsification** – l'utilisation de structures de données spécialisées telles que les matrices **sparse** contribue à optimiser la persistance et la rapidité d'accès, tout en réduisant l'empreinte mémoire.

B. Scalabilité

La **scalabilité** se réfère à la capacité du SCN à gérer efficacement un grand nombre d'entités, ce qui pose deux défis majeurs : le coût du stockage de la matrice ω et le temps de calcul associé à la mise à jour, qui implique potentiellement de traiter $\mathcal{O}(n^2)$ paires à chaque itération. Pour pallier ces difficultés, il est judicieux de recourir à des méthodes de **sparcification** qui consistent à filtrer les liens insignifiants. Par exemple, en définissant un opérateur de filtrage Π tel que

$$\tilde{\omega}_{i,j} = \Pi(\omega_{i,j}) = \begin{cases} \omega_{i,j}, & \text{si } \omega_{i,j} \geq \theta, \\ 0, & \text{sinon,} \end{cases}$$

on peut ainsi réduire le nombre total de liens stockés à $\mathcal{O}(nk)$, avec $k \ll n$. Par ailleurs, la parallélisation du calcul, en utilisant des techniques telles que le double-buffering et la distribution des tâches sur plusieurs threads ou machines (via MPI ou OpenMP), permet de diviser le travail de mise à jour de manière efficace. La stratégie consiste à partitionner la matrice ω en sous-blocs, de sorte que chaque thread traite indépendamment un ensemble de lignes ou de colonnes, réduisant ainsi considérablement le temps de calcul par itération.

C. Gestion d'Entités Hétérogènes dans un Seul Framework

Dans de nombreux cas d'usage, le SCN doit intégrer des entités de nature diverse – par exemple, des **entités sub-symboliques** issues d'embeddings neuronaux, des **entités symboliques** représentées par des ensembles de règles ou d'axiomes, et même des agents physiques comme des robots. Pour assurer une **intégration homogène**, il est essentiel de définir une **interface** abstraite, par exemple une classe **Entity**, qui spécifie les méthodes communes telles que `getRepresentation()` ou `computeSynergy(other)`. Chaque type d'entité (sub-symbolique, symbolique ou hybride) implémente cette interface en fournissant ses propres méthodes de calcul. Par exemple, dans un scénario où la fonction de synergie est définie par

$$S(i, j) = \alpha S_{\text{sub}}(i, j) + (1 - \alpha) S_{\text{sym}}(i, j),$$

les entités sub-symboliques et symboliques peuvent ainsi être traitées uniformément par le module de **Calcul de Synergie**, qui s'appuie sur les méthodes polymorphiques définies dans la classe **Entity**. Cette approche permet non seulement de faciliter l'extension du SCN à de nouveaux types d'entités, mais également de standardiser la manière dont les différentes représentations interagissent au sein du même réseau.

D. Interfaçage et API

Afin de garantir une intégration efficace du SCN dans un environnement logiciel plus vaste, il est impératif de développer des **API** claires et bien définies. Ces API doivent permettre l'accès aux données du SCN (la matrice ω et la fonction de synergie $S(i, j)$), ainsi que la modification dynamique des paramètres essentiels tels que η , τ et γ . Par exemple, une interface du type

$$\text{getOmega()} \quad \text{et} \quad \text{setParameters}(\eta, \tau, \gamma)$$

permet de communiquer avec d'autres modules ou systèmes, tels que des outils de visualisation, des systèmes de contrôle en temps réel ou des simulateurs de tâches complexes. Cette interopérabilité est cruciale pour que le SCN puisse fonctionner en tant que composant intégré dans des frameworks plus larges, notamment dans des applications de robotique ou d'apprentissage multimodal.

Conclusion

L'**architecture** d'un SCN opérationnel doit être conçue pour répondre simultanément aux exigences de persistance, de scalabilité et d'intégration d'entités hétérogènes. En mettant en œuvre un **module de persistance** robuste, le système peut sauvegarder et restaurer efficacement l'état de la matrice ω , ce qui est indispensable pour la continuité des simulations et l'analyse a posteriori. La scalabilité est assurée grâce à des techniques de **sparcification**, à une parallélisation judicieuse et à l'utilisation de structures de données adaptées, ce qui permet de gérer des réseaux de grande dimension tout en maintenant un temps de calcul raisonnable. Enfin, la gestion d'entités hétérogènes repose sur une abstraction orientée objet et sur des API standardisées qui garantissent une interopérabilité optimale entre les différents types d'entités et les modules de calcul. Ces stratégies, intégrées dans un cadre modulaire et extensible,

posent les bases d'un SCN capable de s'insérer dans des environnements complexes et évolutifs, tel que détaillé au Chapitre 5.

5.1.2.3. Préparer les Chapitres Suivants (Multi-Échelle, Optimisations Avancées, Etc.)

Dans les sections précédentes, il a été montré comment une architecture modulaire favorise la clarté et la flexibilité d'un **Synergistic Connection Network** (SCN). Les deux premiers objectifs, à savoir la séparation des rôles (calcul de synergie, mise à jour ω , inhibition, etc.) et la prise en compte de la persistance, de la scalabilité et de l'hétérogénéité des entités, répondent déjà à la plupart des exigences d'une mise en œuvre robuste. Néanmoins, cet agencement ne représente que la base d'un cadre plus large. Les chapitres suivants (6, 7, 8) détailleront des domaines où l'architecture doit être étendue ou adaptée à des situations plus sophistiquées : il peut s'agir de gérer plusieurs niveaux d'échelles, de recourir à des algorithmes d'optimisation plus élaborés, ou d'intégrer des canaux multimodaux variés.

Le chapitre 6 abordera la question de la **multi-échelle**, c'est-à-dire l'articulation du SCN selon des niveaux hiérarchiques, avec la possibilité que la synergie et la mise à jour ω se déclinent sous différentes granularités. On peut envisager des entités formant des micro-clusters locaux, enchevêtrés dans de plus grands macro-clusters destinés à un aperçu plus global. Cette hiérarchie demande souvent de disposer de structures de données ou de métadonnées permettant de résumer les sous-réseaux à des niveaux supérieurs, et de prévoir des règles distinctes à chaque palier. L'architecture modulaire telle qu'elle se dessine au chapitre 5 devra autoriser l'insertion de "meta-niveaux" ou "cluster-layers" dans la boucle de mise à jour. Les modules définissant la synergie ou la mise à jour ω se voient ainsi complétés par un gestionnaire multi-niveau supervisant l'échange d'informations entre échelles.

Le chapitre 7 se consacrera aux **optimisations avancées** et à l'**adaptation dynamique** des paramètres. L'idée d'adapter automatiquement η , τ ou γ en cours d'exécution, en fonction d'observateurs mesurant la vitesse de convergence ou le taux d'oscillation, nécessitera un composant "feedback" supplémentaire. Des algorithmes inspirés des approches globales (tels que le recuit simulé plus élaboré, ou un gradient global sur la matrice ω) pourront aussi être intégrés au SCN. Une architecture modulaire rend bien plus simple l'activation de routines spécialisées (par exemple, un "ParamManager" réajustant η selon le rythme d'évolution des liens ω), tout en préservant la mise à jour locale. Il pourra également être question d'une commande "top-down" ou d'un feedback direct injectant un contrôle externe sur certaines parties du réseau, pour dépasser la simple auto-organisation "bottom-up".

Enfin, le chapitre 8 portera sur la **fusion multimodale**. Il s'agira de démontrer la souplesse du **Deep Synergy Learning** (DSL) dans des domaines complexes (vision, langage, audio, etc.) où les entités représentent des canaux ou des descripteurs hétérogènes. On appréciera alors l'importance cruciale d'une architecture apte à recevoir, pour chaque paire (i, j) , plusieurs sources de synergie $S_1(i, j)$, $S_2(i, j)$..., correspondant à différents canaux (visuel, textuel, sonore). Ce cadre multimodal repose sur la cohabitation naturelle de multiples "entités" dans le SCN, ce que le chapitre 5 aura préparé, en s'assurant que la gestion polymorphe des entités et la persistance à grande échelle puissent se décliner de façon transparente, quel que soit le mode de calcul du score $S(i, j)$.

En somme, l'architecture décrite au chapitre 5 constituera le socle permettant de développer chacune de ces expansions. Il sera beaucoup plus aisé d'ajouter, par exemple, un module "multi-scale synergy" pilotant un niveau hiérarchique supplémentaire, ou un module "fusion visuo-textuelle" manipulant diverses modalités, lorsqu'un découpage modulaire (mise à jour ω , calcul de synergie, etc.) est déjà en place. Une mise en cohérence entre les chapitres 6, 7, et 8 permettra de montrer à quel point le **SCN**, initialement concentré sur l'auto-organisation locale des liens, peut s'adapter à des configurations hiérarchiques, des stratégies de contrôle top-down, ou des données multimodales abondantes. L'ensemble fera apparaître le **Deep Synergy Learning** comme un paradigme général et extensible, autorisant nombre d'expérimentations et d'applications pratiques, dès lors qu'on se dote d'une architecture logicielle rigoureuse et ouverte.

5.2. Vision d'Ensemble de l'Architecture SCN

5.2.1. Notion de “Noyau” vs. “Modules Adjoints”

- 5.2.1.1. **Noyau (Core)** : stocke la matrice $\{\omega_{i,j}\}$, exécute le cycle de mise à jour selon la formule du DSL.
- 5.2.1.2. **Modules** :
 - Module Synergie (calcule $S(i,j)$),
 - Module Inhibition/Contrôle,
 - Module d'Interface (ajout/suppression d'entités), etc.

5.2.2. Cycle de Vie du SCN

- 5.2.2.1. **Initialisation** : chargement des entités, création (ou non) d'une matrice ω initiale.
- 5.2.2.2. **Boucle d'itérations** : calcul de synergie, mise à jour ω , éventuelle inhibition ou saturation, extraction de clusters.
- 5.2.2.3. **Sortie / Flux continu** : quand stabilisé, ou en mode perpétuel si apprentissage continu.

5.2.3. Exemples d'Organisation

- 5.2.3.1. **Mono-module** minimal (tout dans un même bloc)
- 5.2.3.2. **Architecture modulaire** (chacun s'occupe d'une fonction)
- 5.2.3.3. **Architecture distribuée** (plusieurs sous-SCN coopérant)

5.2. Vision d'Ensemble de l'Architecture SCN

5.2.1. Notion de “Noyau” vs. “Modules Adjoins”

5.2.1.1. Noyau (Core) : Stocke la Matrice $\{\omega_{i,j}\}$ et Exécute le Cycle de Mise à Jour selon la Formule du DSL

Le **Noyau** représente la pièce maîtresse de l'architecture du *Synergistic Connection Network* (SCN) dans le cadre du **Deep Synergy Learning** (DSL). Il constitue la **colonne vertébrale** du système en assurant à la fois le **stockage** central de la matrice des pondérations $\Omega(t) \in \mathbb{R}^{n \times n}$ et le **pilotage** de la dynamique d'auto-organisation par la mise à jour itérative des poids $\omega_{i,j}(t)$. Cette section présente en détail le rôle du Noyau, sa formalisation mathématique, la gestion des aspects de stockage et de complexité, ainsi que son interaction avec d'autres modules du système.

A. Rôle Fondamental du Noyau

Le **Noyau** se définit avant tout comme le gardien de la matrice $\Omega(t)$, qui regroupe l'ensemble des pondérations $\omega_{i,j}(t)$ établies entre toutes les entités \mathcal{E}_i du réseau. En pratique, il doit garantir un **stockage efficace** de ces valeurs, que ce soit sous un format **dense** ou **sparse** afin de minimiser l'empreinte mémoire, notamment lorsque le nombre d'entités n devient très grand (par exemple, $n = 10^4$ à 10^5). Sur le plan **mathématique**, la matrice $\Omega(t)$ évolue selon une fonction de transition que nous pouvons écrire de manière générale :

$$\omega(t+1) = F(\omega(t), \mathbf{S}, \{\text{inhibition, bruit, etc.}\}),$$

où \mathbf{S} représente la matrice des **synergies** $S(i,j)$ entre les entités, et les autres termes (inhibition, bruit, etc.) viennent moduler l'évolution du système.

Sur le plan **algorithmique**, le Noyau agit comme une brique centrale qui orchestre l'ensemble du **cycle itératif** de mise à jour. Ce cycle repose sur la formule de mise à jour propre au DSL. Par exemple, dans le cas d'une mise à jour de type **additif**, nous avons :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)] - \gamma \sum_{k \neq j} \omega_{i,k}(t) + \dots,$$

où η représente le **taux d'apprentissage**, τ le **facteur de décroissance** qui module le coût de maintenir un lien élevé, et γ un coefficient d'**inhibition** appliqué sur les liens concurrents. Cette formule exprime l'essence de la **descente d'énergie** qui guide l'auto-organisation du réseau.

B. Stockage de la Matrice Ω et Gestion de la Complexité

Le Noyau doit assurer le **stockage** et la gestion de la matrice $\Omega(t)$, qui contient $O(n^2)$ éléments pour n entités. Lorsque n est élevé, le coût mémoire d'un stockage dense peut rapidement devenir prohibitif (pouvant atteindre des dizaines ou centaines de gigaoctets). Pour pallier ce problème, des techniques de **sparcification** telles que l'utilisation de structures de données *sparse* ou la limitation des liaisons à un nombre restreint de voisins (k-NN ou ϵ -radius) peuvent être mises en œuvre. Ainsi, le Noyau doit intégrer des routines d'accès rapide et de mise à jour efficaces, en exploitant par exemple des stratégies de **parallélisation** ou de **double-buffer** pour gérer les itérations.

Sur le plan **algébrique**, la mise à jour s'effectue selon la relation :

$$\omega(t+1) = F(\omega(t), \mathbf{S}, \{\text{termes de régulation}\}),$$

ce qui implique un parcours complet (ou partiel en mode sparse) sur tous les indices $(i,j) \in \{1, \dots, n\}^2$.

C. Cycle Itératif de Mise à Jour

Le fonctionnement du Noyau peut être décomposé en une série d'étapes séquentielles qui assurent la mise à jour correcte de la matrice Ω . Ce **cycle** se réalise généralement en six étapes clés :

1. **Lecture** : Le Noyau commence par lire la matrice des pondérations $\omega_{i,j}(t)$ depuis un buffer de lecture, souvent désigné par w_{current} .
2. **Acquisition de la Synergie** : Il sollicite un module externe, tel que le **Module Synergie**, afin d'obtenir les valeurs de $S(i,j)$ qui déterminent la proximité ou la complémentarité entre les entités.
3. **Calcul des Incréments** : Pour chaque paire (i,j) , le Noyau calcule l'incrément $\Delta\omega_{i,j}$ à partir de la formule de mise à jour, tenant compte des paramètres d'apprentissage, des termes d'inhibition et de toute composante stochastique.
4. **Écriture** : Les nouvelles valeurs $\omega_{i,j}(t+1)$ sont écrites dans un buffer d'écriture, appelé w_{next} .
5. **Traitement Postérieur** : Un module optionnel d'inhibition ou de post-traitement peut être appliqué pour ajuster les valeurs calculées, par exemple en imposant des limites ou en appliquant des règles de sparsification.
6. **Basculement** : Enfin, après vérification de la cohérence de l'itération, le Noyau effectue un swap entre w_{current} et w_{next} , marquant la transition de l'itération t à $t+1$.

Cette séquence garantit que la dynamique du DSL se réalise de manière ordonnée et cohérente, en appliquant à chaque itération la formule mathématique qui définit la descente d'énergie du système.

D. Interfaces et Modularité

Le Noyau se veut **minimaliste** dans ses responsabilités : il gère la matrice Ω et orchestre la boucle de mise à jour, tout en déléguant le calcul détaillé de $S(i,j)$ et des éventuels termes additionnels (comme l'inhibition) à des modules externes. Grâce à cette **modularité**, il est possible de modifier ou de remplacer ces modules indépendamment du cœur de la dynamique. Par exemple, il est envisageable de substituer le **Module Synergie** basé sur la distance euclidienne par un module exploitant la **co-information** ou d'autres critères, sans affecter le mécanisme central de mise à jour de ω . Cette séparation des préoccupations favorise l'**extensibilité** et la **maintenabilité** du système.

E. Conclusion sur le Noyau (Core)

En résumé, le **Noyau** représente la **colonne vertébrale** du SCN dans le cadre du DSL. C'est l'endroit où la **matrice** $\Omega(t)$ est **stockée** et **gérée**, et où la **boucle itérative** de mise à jour se réalise en appliquant la formule fondamentale, par exemple :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)] - \gamma \sum_{k \neq j} \omega_{i,k}(t) + \dots$$

D'un point de vue **mathématique**, c'est dans ce Noyau que se matérialise la **descente d'énergie** et la convergence vers des points fixes ou des attracteurs du système. Du point de vue **logiciel**, le Noyau constitue une brique modulaire essentielle, qui, grâce à des interfaces claires avec les modules de calcul de synergie, d'inhibition, et éventuellement de gestion d'états internes, assure l'auto-organisation du SCN dans un environnement évolutif et scalable. Ainsi, le **Noyau (Core)** n'est pas seulement responsable de la mise à jour des pondérations, il assure également la cohérence globale du système en orchestrant la séquence d'opérations qui fait converger la dynamique vers une configuration stable, tout en permettant l'intégration et la maintenance de modules complémentaires dans un cadre logiciel robuste et évolutif.

F. Le Noyau gérant également les États, Mémoires et Agents (complement)

Dans l'architecture précédente, on a insisté sur le fait que le **Noyau** (ou **Core**) d'un **Synergistic Connection Network** (SCN) se concentrait principalement sur le stockage de la matrice ω et l'exécution de la boucle de mise à jour du **Deep Synergy Learning** (DSL). Toutefois, lorsqu'une **entité** \mathcal{E}_i n'est pas un simple nœud statique, mais un **agent** susceptible de disposer d'un **état** interne (mémoire, variables locales, etc.), cette information doit être gérée d'une manière cohérente avec le **Noyau**. En pratique, il est fréquent que le **Core** stocke non seulement les pondérations $\omega_{i,j}$, mais aussi tout ou partie des **états** de ces agents, afin d'assurer un **cycle** de mise à jour unifié.

Si chaque entité \mathcal{E}_i possède un vecteur d'état $\mathbf{s}_i(t)$ (par exemple, un accumulateur ou même une cellule LSTM locale), on peut étendre la structure du **Noyau** pour mémoriser ces informations. Au lieu de se contenter d'une simple matrice $\omega(t) \in \mathbb{R}^{n \times n}$, le Noyau conserve également :

$$\mathbf{S}_{\text{etats}}(t) = \{\mathbf{s}_1(t), \mathbf{s}_2(t), \dots, \mathbf{s}_n(t)\},$$

chaque $\mathbf{s}_i(t)$ pouvant être un vecteur (ou un objet plus complexe) décrivant la **mémoire** courante de l'agent \mathcal{E}_i . On dispose alors d'une **dynamique** double : la mise à jour de $\omega_{i,j}$ (pondérations inter-agents) et la mise à jour de $\mathbf{s}_i(t)$ (mémoire ou état interne), laquelle peut être notée :

$$\mathbf{s}_i(t+1) = \Phi(\mathbf{s}_i(t), \{\omega_{i,j}(t)\}, \{\mathbf{s}_j(t)\}_{j \neq i}).$$

Le **Noyau** coordonne ainsi l'évolution simultanée de ω et \mathbf{s}_i , selon les règles du DSL enrichi (une équation couplée permettant aux agents de modifier leur état selon la synergie, et aux poids d'être ajustés en fonction des états et synergies).

Lorsqu'une entité \mathcal{E}_i représente un **agent** autonome, capable de prendre des décisions ou de gérer un comportement local, on veut généralement lui attribuer un espace d'état $\mathbf{s}_i(t) \in \mathbb{R}^d$ (ou un ensemble de variables plus symboliques). Le **Noyau** peut alors :

7. **Stocker** ces états dans une structure annexe, par exemple un tableau $\text{Etats}[1..n]$ où $\text{Etats}[i]$ pointe vers $\mathbf{s}_i(t)$.
8. **S'assurer** que, lors de chaque itération DSL, on exécute la mise à jour de $\mathbf{s}_i(t)$. Ceci peut être fait après ou avant la mise à jour de $\omega_{i,j}(t)$. Sur le plan formel, on définit un couple :

$$(\boldsymbol{\Omega}(t), \mathbf{S}_{\text{etats}}(t)) \mapsto (\boldsymbol{\Omega}(t+1), \mathbf{S}_{\text{etats}}(t+1)),$$

où la fonction de transition prend en compte à la fois la dynamique DSL sur ω et la dynamique interne sur \mathbf{s}_i .

On peut imaginer une **formule** conjointe :

$$\begin{cases} \mathbf{s}_i(t+1) = \Phi_i(\mathbf{s}_i(t), \{\omega_{i,j}(t)\}, \{\mathbf{s}_j(t)\}), \\ \omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta \left[S(\mathbf{s}_i(t), \mathbf{s}_j(t)) - \tau \omega_{i,j}(t) \right] + \dots \end{cases}$$

La **synergie** $S(\mathbf{s}_i, \mathbf{s}_j)$ dépend alors des **états** respectifs, ce qui justifie que le **Noyau** gère ou au moins coordonne l'accès à \mathbf{s}_i . En pratique, on peut loger \mathbf{s}_i dans le **Noyau** ou dans un "Module État" qui communique fortement avec le **Noyau**. L'important est que la **boucle** de mise à jour globale traite à la fois les poids $\omega_{i,j}$ et les états \mathbf{s}_i , maintenant une cohérence entre les deux.

D'un point de vue **architecture**, si l'on ne gérait que ω , le **Noyau** se limite à la "matrice de liens" et sa dynamique. Dès que des entités \mathcal{E}_i possèdent un état interne évolutif, il devient naturel d'étendre le **Noyau** pour inclure la **mémorisation** de ces états, ou au moins l'orchestration de leur mise à jour. Selon le degré de modularité voulu, le Noyau peut :

- Soit conserver physiquement les \mathbf{s}_i
- Soit appeler un "Module État" qui, à chaque itération, calcule $\mathbf{s}_i(t+1)$.

Dans tous les cas, la **synchronicité** de la mise à jour ω et \mathbf{s}_i doit être garantie : on évite que l'agent modifie son état selon un $\omega(t)$ déjà obsolète ou qu'il mette à jour ω avant d'avoir tenu compte du nouvel état. Les algorithmes de double buffer (pour ω) peuvent s'étendre à l'état \mathbf{s}_i s'il faut traiter la mise à jour de façon synchrone.

Conclusion

Il apparaît donc naturel que le **Noyau** n'héberge pas **uniquement** la matrice ω , mais puisse **inclure** (ou gérer étroitement) les **états** des entités/agents, si l'on souhaite un **DSL** où ces entités possèdent une mémoire ou un comportement local. Les formules mathématiques, dans ce cas, se présentent sous forme de couples $(\omega, \mathbf{s}) \mapsto (\omega', \mathbf{s}')$,

assurant la cohérence de la **dynamique** auto-organisée. L’architecture globale reste la même : le **Noyau** — désormais muni à la fois de Ω et de $S_{\text{états}}$ — exécute le **cycle** DSL, tandis que les **Modules** adjacents (calcul de synergie, inhibition, recuit simulé, etc.) fournissent les composantes nécessaires (valeurs ou compléments de formules), tout en déléguant la mise à jour itérative au **Noyau**.

5.2.1.2. Modules

Si le **Noyau (Core)** est la pièce maîtresse qui assure la conservation et la mise à jour de la matrice ω (voir 5.2.1.1), il n’opère pas tout seul. Dans une **architecture** SCN modulaire, nous identifions plusieurs **modules** “adjoints” (ou “satellites”) qui viennent se **connecter** au Noyau pour fournir des fonctionnalités supplémentaires :

1. **Module Synergie** (calcul de $S(i, j)$),
2. **Module Inhibition/Contrôle**,
3. **Module d’Interface** (ajout/suppression d’entités, configuration, etc.),
4. (Possiblement) d’autres modules : “Recuit simulé”, “Analyse/Observateur”, “Gestion paramétrique adaptative”, etc.

Chacun de ces **modules** sert un **rôle** précis et interagit mathématiquement ou logiquement avec le **Noyau** pour enrichir le comportement du SCN. Nous détaillons ci-après les trois modules majeurs cités.

A. Module Synergie (calcule $S(i, j)$)

Le **Module Synergie** constitue un élément fondamental dans l’architecture d’un **Synergistic Connection Network (SCN)**, car c’est lui qui fournit la **fonction** $S(i, j)$ permettant de mesurer la “distance”, la “similarité” ou plus généralement la **coopération** entre deux entités \mathcal{E}_i et \mathcal{E}_j . Dans le **Deep Synergy Learning (DSL)**, chaque pondération $\omega_{i,j}(t)$ se met à jour en tenant explicitement compte de la valeur $S(i, j)$, ce qui fait du Module Synergie un composant déterminant pour l’évolution du réseau.

Il est fréquent de poser, dans la forme la plus simple, une mise à jour additive du type

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)] - (\text{termes d’inhibition éventuels}),$$

où $S(i, j)$ désigne précisément la valeur fournie par ce **Module Synergie**. Sur le plan **mathématique**, on peut voir S comme une matrice $\mathbf{S} \in \mathbb{R}^{n \times n}$, mais sa construction ou son calcul peut être complexe, en fonction de la nature des entités et des traitements nécessaires pour en déduire la synergie.

L’objectif fondamental du Module Synergie est de **fournir**, pour chaque couple (i, j) , une quantité $S(i, j)$ qui sera ensuite exploitée par le Noyau (Core). Le Module Synergie peut être envisagé comme un “serveur” auquel le Noyau fait appel : lors de chaque itération du DSL, le Noyau “demande” la valeur $S(i, j)$ pour mettre à jour $\omega_{i,j}$. Selon la **typologie** des entités, la fonction S peut adopter diverses formules. Dans un cas sub-symbolique, on peut considérer une distance de type gaussien ou exponentiel :

$$S(i, j) = \exp(-\alpha \|\mathbf{x}_i - \mathbf{x}_j\|^2),$$

avec $\alpha > 0$, ou encore une similarité cosinus :

$$S(i, j) = \frac{\mathbf{x}_i \cdot \mathbf{x}_j}{\|\mathbf{x}_i\| \|\mathbf{x}_j\|}.$$

Dans un contexte plus symbolique ou hétérogène, le Module Synergie peut interroger une ontologie, combiner des scores de règles ou de co-occurrences, ou même implémenter une “co-information” statistique

$$S(i, j) = I(\mathcal{E}_i; \mathcal{E}_j),$$

et retourner ce score au Noyau. La **généralité** du DSL impose souvent que le Module Synergie soit écrit de manière suffisamment polymorphe ou extensible, afin que l'on puisse brancher différents calculs de $S(i, j)$ en fonction des types d'entités \mathcal{E}_i et \mathcal{E}_j .

D'un point de vue **implémentation**, le Module Synergie peut contenir des structures d'index (par exemple, KD-Tree, LSH) lorsqu'il s'agit de recherches k-NN ou d'autres formes de *voisinage*, mais peut aussi reposer sur un pré-calcul ou sur un appel direct à des fonctions (similitude de cosinus, distance euclidienne). La forme abstraite se conçoit aisément : à chaque itération, on parcourt les paires (i, j) actives ou autorisées, et on invoque une fonction

$$\text{synergyModule.getSynergy}(i, j)$$

qui renvoie la valeur $S(i, j)$. Sur le plan **algorithmique**, si l'on n'impose aucune parcimonie (k-NN ou ϵ -radius), on se confronte à un calcul en $O(n^2)$, ce qui peut être prohibitif pour de grands n . Les mêmes techniques de réduction de densité (Section 7.2.3.3) s'appliquent alors : on ne fait appel à `getSynergy` que pour un nombre restreint de paires.

Il convient de noter que le Module Synergie est **indépendant** de la mise à jour de $\omega_{i,j}$. Il ne modifie pas lui-même les pondérations : il se contente de calculer ou de fournir $S(i, j)$. Cette **séparation** clarifie l'architecture du SCN : le **Noyau** exécute la règle DSL, tandis que le Module Synergie "répond" aux requêtes $S(i, j)$. Sur le plan **mathématique**, la valeur de $S(i, j)$ peut dépendre directement des entités $\mathcal{E}_i, \mathcal{E}_j$ (par exemple de leurs vecteurs internes), ou s'appuyer sur des états que l'on stockerait dans un autre module (ex. mémoire interne de l'agent). Il reste que, pour chaque couple (i, j) , le Noyau reçoit une simple valeur scalaire, laquelle est injectée dans la formule de mise à jour $\omega_{i,j}(t + 1)$.

En conclusion, le **Module Synergie** occupe une place centrale dans la boucle DSL, assurant que chaque liaison $\omega_{i,j}$ se mette à jour proportionnellement à la "coopération" ou "affinité" mesurée entre \mathcal{E}_i et \mathcal{E}_j . Que l'on utilise une distance exponentielle, une corrélation, une co-information symbolique ou d'autres métriques, tout passe par ce Module Synergie, rendant la solution logicielle plus modulaire et plus souple : la nature concrète de $S(i, j)$ est encodée hors du Noyau, dans un composant spécialisé dont le rôle strict est de **calculer** la valeur de la synergie pour chaque paire active dans le SCN.

B. Module Inhibition/Contrôle

Le **Module Inhibition/Contrôle** joue un rôle complémentaire dans la **dynamique** d'un **Synergistic Connection Network (SCN)**, en ajoutant ou en modulant les termes d'**inhibition** (ou d'autres formes de régulation) pendant la mise à jour des pondérations $\omega_{i,j}$. Sur le plan **mathématique**, il s'agit de façonner la descente ou la dynamique du **Deep Synergy Learning (DSL)** au-delà du simple terme $\eta[S(i, j) - \tau \omega_{i,j}]$, de manière à imposer une "**compétition**" ou un "**budget**" aux liens sortants. La présentation qui suit précise la nature de ces fonctions et leur intégration au sein du SCN, où le **Noyau** s'en remet à un composant externe (ce Module) pour appliquer le contrôle nécessaire.

Dans de nombreuses variantes du DSL (cf. Chap. 4.2.2.2 ou 4.4.2), on considère une **inhibition** compétitive : lorsque $\omega_{i,j}$ tend à s'amplifier, on désire qu'une partie de cette croissance "pénalise" les autres liaisons $\omega_{i,k}$ pour $k \neq j$. L'idée courante est d'ajouter dans la **formule** :

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \Delta_{\text{update}}(i, j) + \Delta_{\text{inhibition}}(i, j),$$

où l'inhibition peut s'écrire, par exemple, sous la forme

$$\Delta_{\text{inhibition}}(i, j) = -\gamma \sum_{k \neq j} \omega_{i,k}(t).$$

Le **terme** $\gamma > 0$ règle l'intensité de la compétition : plus γ est grand, plus on pousse l'entité \mathcal{E}_i à **sélectionner** un (ou quelques) liens "gagnants", au détriment des autres. Sur le plan **algébrique**, on formalise alors une contrainte de type

$$\sum_j \omega_{i,j}(t) \leq \Omega_{\text{max}}$$

pour chaque i , dans une perspective saturante ou dans un style "winner-takes-most". Ce **contrôle** s'interprète comme une régulation additionnelle empêchant la croissance illimitée de multiples $\omega_{i,j}$ autour du même nœud \mathcal{E}_i .

Dans un SCN **modulaire**, le **Noyau** (Core) détient la boucle principale de mise à jour (cf. 5.2.1.1), mais ne gère pas directement les détails de l'inhibition (ou du contrôle plus vaste). Il se contente d'appeler un composant externe chaque fois qu'il doit ajouter l'effet de la régulation. On obtient un **module** dédié, qu'on peut désigner "Module Inhibition" ou "Module Contrôle" selon l'étendue de ses fonctions, chargé d'appliquer :

$$\Delta_{\text{inhibition}}(i, j) = \text{inhibitionModule.computeTerm}(i, j, \omega_{\cdot}(t)).$$

Ainsi, si l'on souhaite passer d'une inhibition "classique" ($-\gamma \sum_{k \neq j} \omega_{i,k}$) à un schéma plus sophistiqué (budget local, saturation, etc.), on modifie ce **Module** sans altérer la logique interne du **Noyau**. De la même façon, si l'on désire ajouter un terme de "bruit" ou d'autres contrôles (clamping, recuit, activation imposée), on étend simplement ce composant pour retourner la portion $\Delta_{\text{contrôle}}$.

Le terme "Inhibition" peut recouvrir différentes formes de **contrôle** :

Compétition Hebbienne : imposer un frein proportionnel à la somme des poids sortants $\sum_k \omega_{i,k}$.

Budget local : $\sum_j \omega_{i,j} \leq \Omega_{\text{max}}$. Cela se traduit parfois par un "coût" ou un "clip" imposé si la somme dépasse Ω_{max} . On peut écrire :

$$\Delta_{\text{budget}}(i, j) = -\lambda \left(\sum_m \omega_{i,m} - \Omega_{\text{max}} \right)_+,$$

où $(x)_+ = \max\{x, 0\}$.

Injection de Bruit / Recuit : pour franchir les minima locaux, on peut inclure un "terme stochastique" $\sigma(t) \xi_{i,j}$ comme décrit en recuit simulé. Alors, le **Module Contrôle** ajoute $\Delta_{\text{noise}}(i, j)$ pour réaliser la partie aléatoire.

Rétroaction top-down : dans certains schémas multi-niveaux, un "macro-niveau" envoie un retour (cf. chap. 6.4.1.2) : "renforce les liens entre \mathcal{E}_i et \mathcal{E}_j car ils appartiennent à un cluster macro", etc.

Ces divers processus se concentrent dans la même entité conceptuelle : le **Module Inhibition/Contrôle**, un "carré" dans l'architecture reliant le **Noyau** via des appels du style "*inhibition.apply(i,j)*".

La mise à jour globale s'écrit en un unique bloc :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \Delta_{\text{update}}(i, j) + \Delta_{\text{inhibition}}(i, j) + \Delta_{\text{noise}}(i, j) + \dots$$

ou, dans une version plus condensée,

$$\omega_{i,j}(t+1) = F_{\text{DSL}}(\omega_{i,j}(t), S(i, j)) + G_{\text{ctrl}}(\{\omega_{i,k}(t)\}_k, \{\dots\}).$$

La **séparation** modulaire signifie simplement que la partie **Inhibition/Contrôle** est **extraite** en un **Module**, afin que le code ou l'algorithme correspondant puisse être aisément remplacé ou ajusté. Cela se justifie dans un DSL **flexible** : on veut expérimenter différents **mécanismes** d'inhibition ou de régulation, sans retoucher la boucle itérative du **Noyau**.

En conclusion, le **Module Inhibition/Contrôle** occupe une place analogue au **Module Synergie** : il ne modifie pas ω de son propre chef (sauf dans la logique du cycle imposé par le Noyau), mais renvoie ou applique la **composante** $\Delta_{\text{inhibition}}(i, j)$ qu'il calcule. Les **règles** (inhibition compétitive, budgets locaux, bruit stochastique, etc.) sont encodées dans la formule $\Delta_{\text{inhibition}}$. Le **Noyau** intègre ensuite ce terme dans la mise à jour, assurant une **auto-organisation** plus riche, pilotée par plusieurs composantes (Synergie, Inhibition, Contrôle).

C. Module d'Interface (ajout/suppression d'entités, etc.)

Dans un **Synergistic Connection Network** (SCN) qui se veut adaptable ou évolutif, on peut être amené à **insérer** de nouvelles entités \mathcal{E}_{n+1} ou au contraire à **retirer** une entité devenant obsolète. Sur le plan **mathématique**, il s'agit de **changer** la dimension de la matrice des pondérations ω , passant d'un espace de taille $n \times n$ à $(n+1) \times (n+1)$ (ou

l'inverse) lorsqu'on introduit (ou supprime) une entité. Sur le plan **logiciel**, le **Noyau** (voir 5.2.1.1) n'a pas à gérer lui-même ces questions de "re-dimensionnement" : le **Module d'Interface** prend en charge ces opérations, assurant une **API** de manipulation externe tout en maintenant la cohérence du SCN.

Lorsqu'un SCN évolue de manière **dynamique**, divers scénarios justifient l'ajout ou la suppression d'entités. D'une part, on peut recevoir un **nouveau** flux d'informations conduisant à la création d'une entité inédite (par exemple, l'apparition d'un nouveau robot-agent dans un essaim, ou de nouvelles données dans un système multimodal). D'autre part, on peut devoir **retirer** une entité qui n'est plus pertinente ou qui s'avère défaillante. Formulons cela plus précisément :

$$\omega^{(n \times n)} \mapsto \omega^{((n+1) \times (n+1))},$$

dans le cas où l'on ajoute \mathcal{E}_{n+1} . Il faut alors initialiser les poids $\omega_{n+1,j}$ et $\omega_{j,n+1}$ pour les anciens j , ce qui revient à insérer une ligne et une colonne dans ω . Le **Module d'Interface** offre donc des fonctions de type

$$\text{addEntity}(\mathcal{E}_{\text{new}}),$$

garantissant l'allocation ou l'expansion du tableau des pondérations, avec une initialisation (par exemple $\omega_{\text{new},j} \approx 0$ ou un petit bruit). Inversement, la **suppression** se modélise comme

$$\omega^{(n \times n)} \mapsto \omega^{((n-1) \times (n-1))},$$

lorsqu'on enlève l'entité \mathcal{E}_q et qu'on retire les lignes et colonnes correspondantes. Le Module Interface propose alors une fonction du type

$$\text{removeEntity}(q),$$

pour "nettoyer" la matrice ω (ou un format sparse) et s'assurer que \mathcal{E}_q n'est plus référencée.

Le **Noyau** (Core) se concentre sur la mise à jour itérative de $\omega(t)$, tandis que le **Module d'Interface** assure le "point d'entrée" permettant aux applications externes ou aux administrateurs d'ajuster la configuration du SCN. Sur le plan **logiciel**, on peut envisager une API ou un service :

1. *addEntity*(Entity *e*):
Prépare un nouvel ID pour l'entité, insère une ligne et une colonne dans ω , initialise les pondérations $\omega_{\text{new},j}$ et $\omega_{j,\text{new}}$.
2. *removeEntity*(ID *eID*):
Retire l'entité *eID*, supprime la ligne et la colonne associées.
3. *setParameter*(paramName, *value*):
Modifie un paramètre ($\eta, \tau, \gamma, \dots$) en notifiant éventuellement le Noyau ou le Module Inhibition.

Cette modularité évite de *déranger* le code cœur du Noyau à chaque fois qu'on veut gérer de nouveaux agents ou entités. Sur le plan **mathématique**, la **dimension** n du SCN est ainsi rendue dynamique, ce qui peut être crucial dans un cadre de robotique ou de multimodalité évolutive, où le flux d'informations s'accroît ou se renouvelle fréquemment.

Le **Module d'Interface** doit, la plupart du temps, insérer ou retirer les entités *entre* deux itérations de mise à jour (plutôt que *pendant* une itération), afin de maintenir la cohérence. Techniquement, on peut concevoir que le Noyau, avant de lancer un "step" suivant, vérifie si des *opérations* en attente (addEntity, removeEntity) doivent être appliquées, met à jour sa structure ω , puis procède au calcul $\omega(t+1) = \dots$. C'est un **schéma** standard : on garantit la synchronisation en décalant l'"insertion" ou la "suppression" au début ou à la fin d'un cycle.

L'Interface ne se limite pas au *ajout/suppression* d'entités : dans bien des cas, on veut aussi **exposer** :

$$\text{setParameter}(\text{paramName}, \text{value}),$$

pour par exemple ajuster η (taux d'apprentissage) ou γ (facteur d'inhibition) en temps réel. Mathématiquement, cela revient à “déformer” la descente ou la mise à jour $\omega_{i,j}(t+1)$. Dans une architecture modulaire, le **Module d'Interface** transmet cette consigne au Noyau (ou au Module Inhibition), leur indiquant d'appliquer la nouvelle valeur à la prochaine itération. Ce mécanisme rend le SCN **adaptable** : on peut *augmenter* γ pour renforcer la compétition, ou baisser τ pour laisser les poids plus libres, sans redémarrer tout le système.

Dans des scénarios **massifs** ou temps réel, on peut imaginer un “dashboard” ou un “control center” reliant l'extérieur (l'utilisateur, un orchestrateur, un superviseur IA) au SCN. Cette passerelle d'interface se focalise sur le “comment ajouter tel agent ? supprimer tel agent ?” ou “comment régler η ?”, en interne. Le **Module Interface** se charge de traduire ces requêtes en modifications effectives de la structure Ω ou des paramètres.

Le **Module d'Interface** se révèle donc crucial pour un **SCN évolutif** ou pilotable. Il prend en charge :

1. L'**insertion** d'une nouvelle entité \mathcal{E}_{n+1} , en allouant les lignes/colonnes additionnelles ou en initialisant $\omega_{\text{new},j}$.
2. La **suppression** d'une entité obsolète, en retirant la ligne/colonne correspondante.
3. L'ajustement dynamique des **paramètres** de l'algorithme (comme η, τ, γ), qu'il transmet au Noyau ou aux autres modules.

Sur le plan **mathématique**, cela traduit la capacité à faire varier la dimension n du SCN et à reconfigurer la descente ou l'inhibition en temps réel, ce qui confère au DSL une plasticité véritablement **multi-scénario**. Le **Noyau**, quant à lui, reste focalisé sur l'exécution de la boucle de mise à jour, tandis que ce Module d'Interface gère la “capacité adaptative” : entrée/sortie d'entités, repositionnement des valeurs de η, τ , etc. Ainsi, l'architecture SCN reste **modulaire** : le **Noyau** ne se trouve pas alourdi par les contingences d'ajout ou de retrait, confiées au composant d'Interface qui orchestre ces changements structurels selon les besoins extérieurs.

D. Autres modules éventuels

En plus de **Synergie**, **Inhibition/Contrôle**, et **Interface**, d'autres **modules** peuvent être envisagés :

- **Module Recuit/Brut** (injection de bruit stochastique, planning de “température”),
- **Module d'Observation** (qui calcule la modularité ou extrait les clusters après chaque itération),
- **Module d'Optimisation paramétrique** (qui essaie diverses valeurs de η, τ, γ en cours de route), etc.

Chaque **extension** s'insère dans l'**architecture** par l'ajout d'un bloc additionnel, plutôt que de modifier le **Noyau** en profondeur.

Conclusion

Les **Modules adjoints** du SCN (au-delà du Noyau) apportent la **richesse** fonctionnelle et la **flexibilité** nécessaires :

Module Synergie : assure le **calcul** de $S(i,j)$ (en se basant sur des vecteurs, des règles, des données robot, etc.).

Module Inhibition/Contrôle : applique des **règles** complémentaires (inhibition compétitive, saturation, recuit), modulant la mise à jour ω .

Module d'Interface : gère l'**ajout** ou la **retrait** d'entités, la **configuration** de paramètres, et constitue une **API** pour l'extérieur.

Cette **architecture modulaire** sépare clairement les **concerns** : le **Noyau** traite la dynamique ω , tandis que chaque module s'occupe de son **domaine** (synergie, inhibition, interface). Mathématiquement, cela permet d'**incorporer** de nouvelles formules ou mécanismes (inhibition non linéaire, synergie binaire, etc.) sans devoir refondre la boucle centrale. En ingénierie, cela facilite la **maintenance**, les **expérimentations**, et la **scalabilité**.

5.2.2. Cycle de Vie du SCN

Au-delà de la simple **séparation** (Noyau vs. Modules) abordée en (5.2.1), il est crucial de voir **comment** un SCN (Synergistic Connection Network) vit **dans le temps** : de l'**initialisation** (qui crée ou charge la matrice ω , installe les entités) à la **boucle itérative** (où s'appliquent la synergie, la mise à jour, l'inhibition, etc.), jusqu'à l'**arrêt** ou la **poursuite** en flux continu. Cette section (5.2.2) détaille ce **cycle de vie**, en le reliant tant aux **aspects mathématiques** (comment on initialise ω , comment on sait qu'on est stabilisé) qu'aux **aspects ingénierie** (comment on charge/sauvegarde les entités, quels signaux déclenchent l'arrêt, etc.).

5.2.2.1. Initialisation : Chargement des Entités et Création (ou non) d'une Matrice ω Initiale

Un **Synergistic Connection Network (SCN)** opère toujours à partir d'un ensemble d'entités $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ et d'une matrice (ou structure) ω regroupant les liaisons $\omega_{i,j}$. La phase d'**initialisation** vise ainsi à introduire l'ensemble des entités dans le système (lecture, import, création dynamique), à attribuer des identifiants et à choisir la condition initiale pour la matrice ω . D'un point de vue **mathématique**, il s'agit de définir $\omega(0) \in \mathbb{R}^{n \times n}$ (ou une structure équivalente) et de **paramétrer** le SCN (fixer η, τ, γ , etc.) pour que la boucle d'auto-organisation puisse commencer dans de bonnes conditions.

Le **premier** acte de l'initialisation consiste à **rassembler** les entités $\{\mathcal{E}_i\}$. Selon l'application, on peut :

(1) Lire un jeu de données sub-symboliques $(\mathbf{x}_1, \dots, \mathbf{x}_n) \in \mathbb{R}^d$,

c'est-à-dire importer des vecteurs ou embeddings depuis un fichier, une base de données, etc.

(2) Importer des entités logiques (symboliques) (règles, ontologies, etc.),

ou

(3) Connecter un flot d'agents (p. ex. robots) (chaque agent devenant \mathcal{E}_i).

D'un point de vue **algorithmique**, on aboutit à un ensemble $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ d'entités numérotées ou identifiées. Sur le plan **mathématique**, on sait qu'on doit manipuler $\omega_{i,j}$ pour $i, j \in \{1, \dots, n\}$. Sur le plan **logiciel**, il faut attribuer une numérotation ou un ID à chaque entité, éventuellement en lien avec un stockage distribué (si le SCN est réparti sur plusieurs nœuds).

Dans un SCN **local**, il suffit de donner des indices $i = 1, \dots, n$ pour indexer la matrice ω . Si le SCN est **distribué** (voir plus loin, Chap. 5.7), on doit répartir les entités \mathcal{E}_i sur plusieurs machines ou partitions, se dotant d'un mécanisme d'allocation d'ID global (ou d'une table de routage). Cette initialisation inclut donc une "cartographie" des entités, permettant à chaque nœud de connaître les indices qu'il gère. D'un point de vue **mathématique**, la matrice ω peut alors être partitionnée par blocs.

Une **question** cruciale est de savoir comment initialiser $\omega_{i,j}(0)$. Plusieurs politiques sont possibles :

$$\omega_{i,j}(0) = \begin{cases} 0, & \text{(zéro total),} \\ \epsilon_{i,j}, & \text{(petit bruit pour casser la symétrie),} \\ \omega_{i,j}^{(\text{saved})}, & \text{(recharge d'un snapshot antérieur).} \end{cases}$$

Zéro total : on fixe $\omega_{i,j}(0) = 0$. C'est la solution la plus simple, mais il peut arriver qu'un SCN dont tous les liens sont à zéro ait besoin de plusieurs itérations pour "briser la symétrie" et commencer à se structurer.

Petit bruit : on affecte à chaque $\omega_{i,j}(0)$ une valeur $\epsilon_{i,j}$ aléatoire dans un intervalle $[-\delta, +\delta]$ ou $[0, \delta]$. Cette option est fréquente pour "réveiller" le réseau et éviter un démarrage trop homogène.

Recharge : si l'on dispose d'un état précédent, on le **charge** (avec persistance). Mathématiquement, il suffit d'initialiser $\omega(t = 0) \leftarrow \omega_{\text{saved}}$. On peut ainsi reprendre une simulation, garder le SCN "en mémoire" entre deux sessions, etc.

Dans l'**implémentation**, on peut décider du **format** : dense vs. sparse. Si l'on vise un SCN complet, on attribue une matrice dense de taille $n \times n$. Pour des réseaux clairsemés, on préfère une structure de liste d'adjacence. Le choix repose sur la taille de n et l'hypothèse de densité. Sur le plan **mathématique**, chaque $\omega_{i,j}$ se trouve dans un espace \mathbb{R} (ou \mathbb{R}^+ , si l'on impose non-négativité).

L'initialisation est aussi le moment de **fixer** ou **charger** les paramètres du DSL :

η (taux d'apprentissage), τ (décroissance), γ (inhibition), ω_{max} (saturation potentielle), etc.

On choisit par exemple $\eta = 0.1, \tau = 0.2$, etc. Si l'on veut un mode additif ou multiplicatif (Chap. 4.2.2.1), on le déclare ici. Sur le plan **mathématique**, c'est un jeu de variables dans la mise à jour $\omega_{i,j}(t + 1) = \dots$. Certains algorithmes autorisent une mise à jour **adaptative** de η ou τ , il faut donc prévoir une structure pour stocker ces variables et un moyen de les changer en cours de route.

Avant de démarrer la **boucle** d'auto-organisation, on exécute souvent un **contrôle** :

1. **Nombre** d'entités n cohérent.
2. **Taille** de la matrice ω adaptée.
3. **Paramètres** (ex. η, τ, γ) chargés.
4. **Modules** (Synergie, Inhibition...) opérationnels, pouvant calculer $S(i, j)$ ou $\Delta_{\text{inhibition}}(i, j)$.

C'est une sorte de "checkpoint" mathématico-logique : si tout est validé, on enclenche l'itération $t = 0 \rightarrow 1$. Sinon, on signale une erreur de configuration.

L'**initialisation** dans un SCN constitue donc la **première** phase du cycle de vie. On :

- **Charge** ou **crée** les entités $\{\mathcal{E}_i\}$,
- **Attribue** des IDs et un format (dense, sparse) pour ω ,
- **Choisit** ou **recharge** la matrice initiale $\omega(0)$,
- **Fixe** les paramètres DSL (η, τ, γ , etc.),
- **Contrôle** la cohérence globale.

À l'issue, on est prêt à entrer dans la **boucle** d'itérations, où la mise à jour $\omega_{i,j}(t + 1)$ (et éventuellement la mise à jour d'états internes) pourra démarrer, faisant évoluer le SCN vers la structure auto-organisée visée.

Conclusion :

La **phase d'initialisation** du SCN revêt une importance mathématique et pratique : c'est là qu'on **fixe** les entités $\{\mathcal{E}_i\}$, qu'on choisit la **matrice** $\omega(0)$ (zéro, bruit, ou rechargement), et qu'on **paramètre** la dynamique. D'un point de vue ingénierie, cela implique un module d'**Interface** (pour charger ou créer des entités), un possible **module Persistance** (pour recharger ω), et la configuration du **Noyau** (Core). Sans une initialisation cohérente, le SCN risque de partir en oscillations stériles (si η, τ sont mal calibrés) ou d'échouer à discriminer les entités (si on restait strictement à 0). Mathématiquement, c'est la condition initiale $\omega(0)$ et la "setup" paramétrique qui déterminent la trajectoire $\omega(t)$ subséquente.

5.2.2.2. Boucle d'Itérations : Calcul de Synergie, Mise à Jour ω , Éventuelle Inhibition ou Saturation, Extraction de Clusters

Après la **phase d'initialisation** (voir §5.2.2.1) qui a permis de rassembler un ensemble d'entités $\{\mathcal{E}_i\}_{i=1}^n$ et de définir la matrice $\omega(t=0)$ (ainsi que les paramètres $\eta, \tau, \gamma, \dots$), le **cœur** de la vie d'un **Synergistic Connection Network (SCN)** se déroule dans une **boucle d'itérations**. Cette boucle actualise, pas après pas, la matrice ω selon une **dynamique** de type **Deep Synergy Learning (DSL)**, en s'appuyant sur les **modules** éventuels (synergie, inhibition, observation). Sur le plan **mathématique**, on cherche à faire converger la suite $\omega(t)$ vers un point fixe (ou un cycle, ou un attracteur plus complexe) qui reflète la formation de **clusters** ou de sous-structures. Sur le plan **implémentation**, on exécute un enchaînement stable : calcul de la **synergie** $S(i, j)$, mise à jour de $\omega_{i,j}$, application d'**inhibition** ou de **contrôles** spécifiques, puis extraction ou observation des clusters résultants.

A. Logique Générale de la Boucle

La **boucle itérative** de mise à jour se formalise par l'entrée et la sortie d'un "pas" de temps. À l'itération t , on dispose d'une **matrice** $\omega(t) \in \mathbb{R}^{n \times n}$ (ou un autre format, par exemple sparse). On connaît aussi l'ensemble des **entités** $\{\mathcal{E}_i\}$ et leurs éventuelles représentations (sub-symboliques, symboliques). Les **paramètres** du DSL (η, τ, γ , etc.) sont fixés, ou adaptables si on a un mécanisme paramétrique. De plus, on a chargé les **modules** (synergie, inhibition, etc.) nécessaires. L'**issue** de chaque pas est la nouvelle matrice $\omega(t+1)$, potentiellement couplée à des informations d'observation (clusters, modularité, etc.), puis on recommence au pas suivant. On répète cette itération jusqu'à atteindre un **critère** (nombre maximal d'itérations, stabilisation, flux continu).

B. Calcul de la Synergie $S(i, j)$

Le **Module Synergie** joue un rôle crucial dans chaque itération : pour que la mise à jour $\omega_{i,j}(t+1)$ tienne compte de la **coopération** (ou distance, ou similarité) entre \mathcal{E}_i et \mathcal{E}_j , on doit évaluer $S(i, j)$. Sur le plan **mathématique**, on se dote d'une fonction

$$S: (\mathcal{E}_i, \mathcal{E}_j) \mapsto \mathbb{R}^+,$$

que l'on peut voir comme

$$S(i, j) = \mathcal{F}(\mathbf{x}_i, \mathbf{x}_j),$$

selon la nature des entités (vecteurs sub-symboliques, symboles avec ontologie, etc.). D'un point de vue **implémentation**, plusieurs approches sont possibles. On peut recalculer l'intégralité de $\{S(i, j)\}$ en $O(n^2)$ à chaque itération si n n'est pas trop grand, ou si la synergie dépend de variables qui changent à chaque pas. On peut aussi ne recalculer que les paires (i, j) actives dans un scénario parcimonieux (k-NN, ϵ -radius). Dans certains cas, $S(i, j)$ varie peu dans le temps si les entités \mathcal{E}_i ne changent pas de représentation, et l'on peut donc se contenter d'une mise à jour épisodique ou "à la demande".

Sur le plan **synchrone**, on peut figer la matrice S au début de chaque itération t , l'utiliser pour mettre à jour ω , puis éventuellement la régénérer à l'itération $t+1$. Cette **séparation** assure la cohérence : chaque "step" emploie la même table de synergies. Dans le cas asynchrone (plus rare), S peut fluctuer en continu, mais cela complique l'analyse mathématique de convergence.

C. Mise à Jour $\omega(t+1)$

Une fois $S(i, j)$ disponible (pour chaque paire (i, j) considérée), le **Noyau** applique la **formule** du DSL :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i, j) - \tau \omega_{i,j}(t)] + \dots$$

La partie ... peut inclure des termes de recuit simulé (ajout d'un bruit $\sigma(t) \xi_{i,j}$), d'inhibition compétitive, ou de budgets. Sur le plan formel, on peut réécrire :

$$\omega_{i,j}(t+1) = F_{\text{DSL}}(\omega_{i,j}(t), S(i, j)) + G_{\text{ctrl}}(\{\omega_{i,k}(t)\}_{k \neq j}),$$

où F_{DSL} recouvre la “descente locale” $\eta [S(i, j) - \tau \omega_{i,j}(t)]$, et G_{ctrl} est un terme de **contrôle** ou d’**inhibition**. L’application synchrone se met aisément en œuvre via un “double-buffer” : on crée une nouvelle matrice $\omega^{\text{next}}(t + 1)$ en lisant $\omega^{\text{current}}(t)$. D’un point de vue **complexité**, si l’on considère chaque paire (i, j) de $\{1, \dots, n\}^2$, on a un coût $O(n^2)$, qu’il est parfois nécessaire de réduire par parcimonie ou parallélisation.

D. Éventuelle Inhibition ou Saturation

De nombreux schémas de DSL prévoient une **inhibition** (chap. 4.2.2.2, 4.4.2) imposant une compétition entre les liens sortants de \mathcal{E}_i . Sur le plan **mathématique**, on ajoute un terme

$$\Delta_{\text{inhibition}}(i, j) = -\gamma \sum_{k \neq j} \omega_{i,k}(t),$$

faisant décroître $\omega_{i,j}(t + 1)$ proportionnellement à la somme des autres $\omega_{i,k}$. Il s’agit d’empêcher qu’un nœud \mathcal{E}_i entretienne simultanément trop de liaisons fortes. On peut également imposer une **saturation** ω_{max} , c’est-à-dire un clipping des valeurs pour éviter que $\omega_{i,j}$ ne devienne trop grande :

$$\omega_{i,j}(t + 1) \leftarrow \min(\omega_{i,j}(t + 1), \omega_{\text{max}}).$$

On peut aussi envisager un budget local $\sum_j \omega_{i,j} \leq \Omega_{\text{max}}$. Sur le plan **implémentation**, on applique souvent l’inhibition et la saturation comme un “post-traitement” de la mise à jour basique $\Delta_{\text{update}}(i, j)$. L’ordre d’application a un léger impact (on peut d’abord faire l’inhibition, puis clipper). Sur le plan **analyse** mathématique, on conceptualise cela comme une **composition** de fonctions inhib/cut :

$$\omega(t + 1) = \text{Clip} \left(\text{Inhib} \left(\omega(t) + \Delta_{\text{update}}(t) \right) \right).$$

Cet enchaînement rend la **dynamique** plus non linéaire, mais confère la possibilité d’une auto-organisation sélective.

E. Extraction de Clusters et Observations

L’**objectif** final d’un SCN est souvent de **révéler** la formation de clusters ou de sous-structures. À l’issue de chaque itération (ou toutes les quelques itérations), on peut :

1. **Seuiler** $\omega_{i,j}(t + 1)$: on considère les liens $\omega_{i,j} > \theta$ comme “actifs”.
2. **Chercher** les composantes connexes (si $\omega_{i,j}$ est vue comme un graphe), calculer une **modularité** ou un **score** de clustering.
3. **Visualiser** la structure ou fournir un retour dans un “dashboard” (par exemple, un affichage temps réel).

Sur le plan **mathématique**, cette extraction de clusters ou d’indices (silhouette, modularité, etc.) ne modifie pas la mise à jour ω , mais permet de **surveiller** la progression de l’auto-organisation. Il est parfois possible de s’arrêter si la matrice ω ne change plus beaucoup, ou si le score de clustering atteint un plateau.

Conclusion

La **boucle d’itérations** constitue le **processus vivant** d’un SCN, là où la matrice $\omega(t)$ prend forme et se réorganise au fil du DSL. Ce cycle comprend :

- Le **calcul** ou l’**accès** à la synergie $S(i, j)$,
- La **mise à jour** de base de $\omega_{i,j}(t + 1)$ (soit additive, soit multiplicative, ou enrichie de termes supplémentaires),
- L’application éventuelle de **mécanismes** : **inhibition** compétitive, **saturation** ou **budget**,
- Un **post-traitement** (extraction de clusters, scores, visualisations) selon la fréquence souhaitée.

Sur le plan **mathématique**, on décrit tout cela par l'opérateur

$$\omega(t+1) = \text{ExtractClusters} \left(\text{Clip} \left(\text{Inhib}(\omega(t), S(t)) \right) \right),$$

mais, en pratique, on sépare le cycle en plusieurs modules ou “étapes” pour la lisibilité et la souplesse. Tant qu'on ne détecte pas de critère d'arrêt ou que le système reste en flux continu, on répète ce cycle, laissant émerger et évoluer les **clusters** internes du SCN.

5.2.2.3. Sortie ou Flux Continu : Stabilisation, ou Mode Perpétuel pour l'Apprentissage Continu

Après la mise en place de la **boucle d'itérations** (§5.2.2.2) dans un **Synergistic Connection Network (SCN)**, plusieurs scénarios de déroulement se présentent quant à la **durée** de fonctionnement ou la **finalité** de l'exécution. Sur le plan **mathématique**, on se demande si la séquence $\omega(t)$ tend à un **point fixe** (ou à un cycle), auquel cas l'évolution peut être considérée comme convergente, ou si le SCN demeure en **flux continu** en raison de l'arrivée de nouvelles entités ou de la transformation perpétuelle des données. Sur le plan **ingénierie**, il s'agit de décider s'il convient de **stopper** le processus une fois atteint un certain critère de stabilisation, ou bien de **maintenir** le SCN actif en permanence, pour un apprentissage ininterrompu.

A. Sortie ou Arrêt après Stabilisation

Sur le plan **mathématique**, la dynamique d'un SCN se décrit par une équation du type

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] - \gamma \dots$$

ou sous forme plus générale $\omega(t+1) = F(\omega(t))$, éventuellement enrichie d'inhibition ou de budgets. Si les paramètres η, τ, γ sont choisis de manière à garantir la **stabilité**, il peut exister un **point fixe** ω^* tel que $\omega^* = F(\omega^*)$. Dans ce cas, la suite $\omega(t)$ tend vers ω^* pour $t \rightarrow +\infty$, au moins dans un régime local (ou global, si la non-linéarité est contrôlée).

Une façon d'**implémenter** cette convergence est de surveiller, à chaque itération, la **différence** entre $\omega(t+1)$ et $\omega(t)$. Par exemple, on peut calculer la norme

$$\| \omega(t+1) - \omega(t) \|_2$$

ou toute autre mesure de dissimilarité. Lorsque celle-ci passe sous un **seuil** ε (et le reste pendant quelques itérations), on juge que le SCN est **pratiquement convergé**. Un “Module d'Observation” (ou un simple mécanisme dans le **Noyau**) peut ainsi déclencher l'arrêt du programme.

Le code peut aussi être muni d'un **critère supplémentaire** (temps maximum, nombre d'itérations maximum) pour éviter une boucle sans fin si la convergence n'est jamais rigoureusement atteinte. Dans un usage **off-line**, typiquement, on exécute le SCN sur un lot d'entités, puis on s'arrête quand la structure ω s'est figée. Sur le plan **mathématique**, on aboutit à un ω^* qui reflète la répartition des entités en clusters, ou plus généralement la configuration stable.

Juste avant l'arrêt effectif, il est courant de **sauvegarder** la matrice ω^* (persistance), afin de pouvoir la réutiliser plus tard (rechargement) ou de l'exploiter dans un autre module (étiquetage de clusters, visualisation, etc.).

B. Mode Perpétuel ou Flux Continu (Apprentissage Continu)

Dans certains environnements (robotique multi-agent, systèmes de recommandation en ligne, traitement de flux de données), le SCN n'est pas censé s'achever après un certain nombre d'itérations, car les **données** ou les **entités** continuent à évoluer ou à arriver. On parle alors d'un **SCN en flux continu**, où la mise à jour $\omega(t+1)$ se poursuit indéfiniment, accompagnée de modifications potentiellement incessantes du calcul de synergie $S(i,j)$. D'un point de vue **mathématique**, la dynamique n'est plus strictement $\omega(t+1) = F(\omega(t))$ avec un opérateur fixe, mais peut dépendre du temps :

$$\omega(t+1) = F(\omega(t), S(\cdot, \cdot, t)),$$

car de nouvelles entités \mathcal{E}_{n+1} peuvent survenir, modifiant la taille de la matrice, ou parce que les entités existantes changent de représentation.

Cette **variabilité** implique qu'on n'atteint pas forcément un point fixe, mais plutôt qu'on "poursuit" un **équilibre** mobile ou une adaptation permanente. On peut y voir un système dynamique à paramètres variant dans le temps (non autonome). Sur le plan **implémentation**, on laisse la boucle itérative fonctionner en continu, et on traite les modifications (ajout/suppression d'entités, mise à jour de synergie, etc.) au fur et à mesure. Un "Module Interface" se charge d'**injecter** les nouvelles entités, en allouant la nouvelle ligne/colonne de ω , puis la dynamique reprend son cours, s'ajustant à ce changement.

Cette perspective rappelle l'**apprentissage en ligne** ou l'**apprentissage continu** : à chaque step, on effectue une mise à jour incrémentale de ω . Parfois, le SCN peut connaître des phases **quasi stationnaires** lorsqu'il n'y a pas de perturbation majeure, puis se réorganiser subitement quand un "choc" se produit (ex. un afflux de nouvelles entités ou un changement de représentation).

C. Bascule Entre Sortie et Flux Continu

Un même SCN peut, en pratique, être utilisé dans deux **modes** :

Mode "One-shot" ou "Offline" : on lance la boucle avec un ensemble fixe $\{\mathcal{E}_i\}$ et on arrête après convergence ou après un nombre de pas défini. On récupère alors ω^* et les clusters.

Mode "Online" ou "Continu" : le SCN tourne indéfiniment, prêt à accepter de nouvelles entités via le "Module Interface" (qui ajoute une ligne/colonne), et la matrice ω s'ajuste en temps réel. On ne parle plus d'arrêt, mais d'**écoulement** permanent, parfois ponctué de vérifications (par exemple, un mini-critère de convergence temporaire, avant qu'une nouvelle perturbation n'apparaisse).

Il est également possible de démarrer en "offline" pour amorcer une structuration, puis de **basculer** en "online" si l'application l'exige, c'est-à-dire laisser le SCN s'adapter aux évolutions ultérieures. Sur le plan **mathématique**, cette bascule revient à transformer un opérateur statique F en un opérateur dynamique dans le temps, sans rebooter la matrice ω . Sur le plan **implémentation**, il suffit de ne plus enclencher le "Critère d'Arrêt" et de laisser le SCN réagir aux notifications d'ajout/suppression d'entités.

Conclusion

La **phase finale** dans le cycle de vie d'un SCN ne se résume pas à un simple arrêt : selon l'objectif, on peut :

Terminer l'exécution après stabilisation (ou nombre maximal d'itérations), en récupérant la structure ω finale et en l'analysant (clusters, etc.). D'un point de vue **mathématique**, on se situe alors dans un cadre de **convergence** classique, $\|\omega(t+1) - \omega(t)\| \leq \varepsilon$.

Prolonger l'exécution en mode **flux continu**, où l'on n'a pas d'arrêt global. Le SCN reste actif, prend en compte les éventuelles modifications d'entités ou de synergie, et se reconfigure en continu. Ce "mode perpétuel" correspond à un système dynamique forcé, plus complexe à analyser, mais indispensable pour des applications en ligne (robotique, streaming, intelligence distribuée).

Ainsi, le **Noyau** et les **Modules** (Synergie, Inhibition, Interface) gardent un rôle identique : ils font la mise à jour $\omega(t+1)$. Seule la **politique** de sortie ou de poursuite diffère, assurant que la même **architecture** puisse servir aussi bien dans un scénario "offline" (stabilisation-clusters) que dans un cadre "online" (adaptation permanente).

5.2.3. Exemples d'Organisation

Au-delà du **cycle de vie** (5.2.2) et de la séparation **Noyau vs. Modules** (5.2.1), il est instructif de voir **comment** un SCN peut s'**organiser** dans des configurations variées, en fonction des **contraintes** et des **échelles**. Parmi les solutions les plus courantes :

Un **mono-module** minimal, où tout le code (calcul de synergie, mise à jour, inhibition, etc.) se trouve dans un **même bloc** (5.2.3.1),

Une **architecture** plus **modulaire**, où chaque fonction (synergie, mise à jour, interface...) est un module ou une classe distincte (5.2.3.2),

Une **architecture distribuée**, déployant plusieurs **sous-SCN** collaborant entre eux (5.2.3.3).

5.2.3.1. Mono-module minimal (tout dans un même bloc)

Lors de la mise en œuvre d'un **Synergistic Connection Network (SCN)**, il est possible de concevoir une **architecture** extrêmement simple et compacte, où toutes les fonctions nécessaires (stockage et mise à jour de ω , calcul de la synergie, inhibition, extraction de clusters, etc.) se retrouvent dans un **unique** module (ou script). Cette approche, qualifiée de “mono-module minimal”, vise la **simplicité** et la **rapidité** de développement pour des projets de petite taille ou des expérimentations rapides. Sur le plan **mathématique**, cela revient à écrire l'ensemble de la boucle d'itérations et des opérations connexes au sein d'un seul bloc de code, sans répartition en classes ou en modules séparés.

Dans ce “mono-module”, on se contente d'un **unique** script, classe ou notebook — éventuellement appelé *SCN_all_in_one.py* — qui regroupe toutes les étapes. Il initialise la matrice $\omega(t = 0)$, gère la boucle d'itérations $\omega(t + 1) = F(\omega(t))$, calcule la **synergie** $S(i, j)$, applique l'**inhibition** si nécessaire, et, s'il y a lieu, **extraît** les clusters (par un simple code de post-traitement) dans le même fichier. D'un point de vue **organisation**, on peut imaginer un pseudo-code :

(1) Définir entités $\{\mathcal{E}_i\}_{i=1}^n$, charger ou créer $\omega(t = 0)$,

(2) Pour $t = 1 \dots T_{\max}$:

$$\left\{ \begin{array}{l} \text{Calculer } S(i, j) \\ \omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta[S(i, j) - \tau \omega_{i,j}(t)] - \gamma \sum_k \dots \\ \text{(Application d'une saturation ou d'un clipping si besoin)} \\ \text{Extraction possible de clusters (optionnel)} \end{array} \right.$$

(3) Fin, sauvegarde ou visualisation.

Sur le plan **ingénierie**, tout cela est concentré dans un **même** bloc, sans appel externe à un “Module Synergie” ou un “Module Inhibition” dédié. Les variables globales η, τ, γ résident dans le même fichier, et le calcul de $S(i, j)$ s'inscrit dans une fonction locale.

Le premier intérêt de ce “**tout-en-un**” est la facilité de mise en place. Pour un **prototype** académique ou une démonstration dans un **projet réduit**, il suffit d'un script unique :

1. On déclare un tableau ou un vecteur pour chaque entité \mathcal{E}_i .
2. On crée $\omega(t = 0)$ en initialisant à zéro ou à un bruit aléatoire.
3. On code la boucle $\omega(t + 1) = \omega(t) + \eta[S(i, j) - \tau \omega(t)] + \dots$.
4. On insère, si nécessaire, quelques lignes pour l'inhibition (par ex., $-\gamma \sum_{k \neq j} \omega_{i,k}$).
5. On ajoute éventuellement un post-traitement pour détecter les clusters (un simple “threshold” sur $\omega_{i,j}$ et un parcours en composantes connexes).

Cette centralisation rend le script particulièrement **facile à comprendre** pour des novices qui veulent voir la **formule** de mise à jour en action. Sur le plan **mathématique**, tout reste sous la forme d'une unique “big loop” :

$$\text{for } t = 1 \dots T_{\max}: \quad \omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta[S(i, j) - \tau \omega_{i,j}(t)] - \gamma \sum_k \dots$$

et ainsi de suite.

Dès qu'on souhaite aller au-delà d'un **essai** isolé, la **structure** en un bloc unique devient handicapante. Sur le plan **évolutif**, chaque fois qu'on veut changer la définition de $S(i, j)$ (par exemple passer d'une distance euclidienne à une co-information symbolique), on doit altérer ce même code, parfois avec des embranchements (*if/else*). En outre, si l'on désire intégrer un **recuit simulé** ou un **module** d'ajout dynamique d'entités (pour un flux continu), on insère de plus en plus de blocs logiques dans le même script, risquant de produire un "**super-monolithe**" illisible.

Sur le plan **maintenance**, un gros bloc rend plus difficile la **collaboration** entre développeurs. Un collègue voulant modifier l'inhibition ne peut isoler un "fichier inhibition.py" : il doit naviguer dans des centaines de lignes d'un script commun. Sur le plan **test** unitaire, il est plus compliqué de tester chaque composant séparément : tout est couplé.

Sur le plan **performance** ou **scalabilité**, un script unique ne prévoit généralement pas d'architecture distribuée ou de parallélisation modulaire. En cas de passage à un $O(n^2)$ massif, il devient périlleux d'adapter ce code monolithique pour incorporer des techniques HPC ou des structures d'indexation complexes.

Rien, d'un point de vue **équations**, n'empêche de **fusionner** dans un même bloc :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \Delta_{\text{update}}(i,j) + \Delta_{\text{inhib}}(i,j)$$

et, si l'on souhaite, d'appeler en fin de boucle une fonction *extract_clusters(w)* qui identifie les composantes fortes. Sur un **petit** dataset (quelques centaines d'entités), c'est parfois la méthode la plus rapide pour un **proof of concept** : on code tout en une journée et on obtient un résultat. La **réduction** en modules n'est pas obligatoire du point de vue mathématique : c'est simplement un **confort** d'ingénierie pour la suite.

Les **petits projets** ou les **travaux pratiques** dans un cours peuvent recourir à ce style "all-in-one". L'étudiant ou le chercheur indique directement la **formule** :

$$\omega_{i,j} \leftarrow \omega_{i,j} + \eta [S(i,j) - \tau \omega_{i,j}] - \gamma \sum_{k \neq j} \omega_{i,k},$$

dans une boucle Python, invoque éventuellement un $\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|)$ pour la synergie, et récupère au final l'état ω . C'est idéal pour un "MVP" (Minimum Viable Product).

En revanche, une extension ultérieure (passer en mode streaming, ou introduire un nouveau calcul de S) demandera de **refactoriser** la base de code en modules séparés (cf. §5.2.3.2).

Conclusion

Le **mono-module minimal** consiste à mettre toutes les composantes du SCN (données, synergie, mise à jour ω , inhibition, etc.) dans un **unique** bloc de code. Cette approche se justifie par :

Simplicité pour des **petits** projets ou des démonstrations : une seule boucle, facile à lire.

Rapidité de prototypage : on écrit directement la formule du DSL sans structurer.

Toutefois, elle montre vite des **limites** pour un usage extensif (ajout de nouvelles entités, recuit, maintenance, test, distribution). On recommande donc cette formule "all-in-one" essentiellement dans des contextes académiques restreints ou pour valider un concept. Dès qu'on anticipe une évolution du code, il convient de se tourner vers une **architecture plus modulaire** (cf. §5.2.3.2) qui scinde la synergie, la mise à jour ω et les autres mécanismes en modules distincts.

5.2.3.2. Architecture Modulaire (Chacun s'occupe d'une fonction)

Lorsqu'un **Synergistic Connection Network (SCN)** a pour ambition d'être maintenu, étendu, ou intégré à des systèmes plus vastes, il est recommandé d'adopter une **architecture modulaire** plutôt que de concentrer tout le code et toutes les fonctionnalités dans un monolithe (cf. §5.2.3.1). Cette architecture consiste à **disséquer** le SCN en

plusieurs **modules** ou **composants**, chacun dédié à une tâche particulière. Sur le plan **mathématique**, on continue de manipuler la même dynamique $\omega(t+1) = F(\omega(t))$ (ou plus détaillée), mais on en sépare clairement les différents “blocs” de calcul. Sur le plan **ingénierie**, on met en place des **interfaces** et des classes (ou fichiers) distincts, afin de pouvoir tester, modifier ou réutiliser chaque composante sans retoucher l’ensemble.

A. Principe Général de la Modularisation

Il s’agit de **découpler** les responsabilités au sein du SCN en divers “modules”, afin de mieux maîtriser la complexité et d’**éviter** que tous les calculs (synergie, mise à jour ω , inhibition, interface, etc.) s’entassent dans un même fichier. On établit donc, au minimum, les blocs suivants :

Un **Noyau (Core)** qui gère la **matrice** ω , orchestre la **boucle** d’itérations et assure la **mise à jour** de base (Δ_{update}).

Un **Module Synergie** qui calcule la **fonction** $S(i, j)$ suivant la nature des entités (euclidienne, cosinus, co-information, etc.).

Un **Module Inhibition/Contrôle** qui introduit des mécanismes complémentaires (inhibition compétitive, saturation, budgets locaux, recuit...).

Un **Module d’Interface** pour gérer l’ajout/suppression d’entités, modifier les paramètres (η, τ, γ) , et connecter éventuellement le SCN à un système extérieur (tableau de bord, webservice, etc.).

(Optionnel) Des **modules** supplémentaires, comme un “Module Observateur” (détection de clusters, visualisation), un “Module RecuitSim” (injection de bruit pour franchir les minima locaux), un “Module Persistence” (sauvegarde/recharge de l’état ω).

Avantages

Dès qu’on s’éloigne d’un simple **prototype**, séparer en modules présente plusieurs atouts notables :

- **Maintenance** améliorée : si la forme de $S(i, j)$ doit évoluer, on ne touche qu’au Module Synergie ; si les règles d’inhibition changent, on modifie le Module Inhibition.
- **Extensibilité** : on peut ajouter un **nouveau** module pour un recuit simulé ou un mécanisme d’apprentissage hiérarchique sans devoir “recoder” le cœur de la boucle.
- **Clarté Mathématique** : chaque bloc de la formule $\omega_{i,j}(t+1) \leftarrow \omega_{i,j}(t) + \dots$ peut être isolé, testable, et documenté. On sait où se trouve $\Delta_{\text{inhibition}}$ et où se trouve Δ_{update} .
- **Test Unitaire** : on peut tester chaque module en isolement (par exemple, vérifier que le Module Synergie renvoie le score $S(i, j)$ correct pour un duo d’entités) avant de lancer l’ensemble.

Implémentation

Logicielle

On peut opter pour une approche **orientée objet** (OOP) en définissant, par exemple, une classe *SCNCore* où se trouvent la matrice ω et la boucle principale, ainsi que des classes *SynergyModule*, *InhibitionModule*, *InterfaceModule*, etc. Chacune possède des méthodes claires (ex. *getSynergy(i, j)*, *applyInhibition(w, i, j)*...), et le **Noyau** les appelle dans le bon ordre à chaque itération. Alternativement, on peut rester en mode **fonctionnel** en dispersant les fonctions dans différents fichiers, tout en ayant un “fichier central” qui enchaîne les appels.

Sur le plan **mathématique**, l’équation de la mise à jour globale reste :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \Delta_{\text{update}}(i, j) + \Delta_{\text{inhibition}}(i, j) + \dots$$

Le Module Synergie intervient pour fournir $S(i, j)$ à la partie $\Delta_{\text{update}}(i, j) = \eta [S(i, j) - \tau \omega_{i,j}(t)]$. Le Module Inhibition fournit $\Delta_{\text{inhibition}}(i, j)$. L’**ordre** d’application peut être séquentiel, reflétant la composition des opérateurs (cf. chap. 4.2.2.2).

B. Exemple de Flux Modulaire en une Itération

Pour illustrer la **séquence** dans un cycle d'itérations :

Pré-calcul ou accès de $S(i, j)$ via le Module Synergie. On peut générer une matrice \mathbf{S} , ou calculer $S(i, j)$ “à la demande” pour chaque (i, j) .

Mise à Jour de base : le Noyau applique l'équation additive ou multiplicative pour $\omega_{i,j}$. Ex. $\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i, j) - \tau \omega_{i,j}(t)]$.

Inhibition/Contrôle : on appelle alors le Module Inhibition qui peut réaliser $\omega_{i,j}(t+1) \leftarrow \omega_{i,j}(t+1) - \gamma \sum_{k \neq j} \omega_{i,k}(t)$, ou imposer un “budget”.

Saturation (si on veut clipper ω entre 0 et ω_{\max}). Ce peut être un sous-module du contrôle, ou un post-traitement.

Extraction de Clusters / Observations : un Module Observateur, si défini, peut lire la matrice $\omega(t+1)$ pour en déduire les composantes connexes, un score de modularité, etc.

Interface : en fin de boucle, le Module Interface peut éventuellement enregistrer la progression, traiter une requête d'ajout d'entité, ou modifier η .

Ainsi, on sépare clairement **qui** fait **quoi**. Les formules mathématiques s'assemblent comme des briques, et le code reflète ce découpage.

C. Comparaison avec l'Approche Mono-Module

Contrairement à un “**all-in-one**” script (voir §5.2.3.1), la modularisation semble plus lourde à configurer pour un très petit projet. Cependant, dès que l'on souhaite :

Ajouter un **nouveau** mode de synergie S (par exemple, un calcul de co-information au lieu d'une distance)

Varier l'**inhibition**

Intégrer un “**Module Recuit**” (ajout de bruit aléatoire pour franchir des minima locaux)

Gérer l'**arrivée** d'entités en flux continu

on réalise qu'une structure modulaire simplifie grandement la maintenance. Chaque module peut évoluer ou être remplacé sans perturber le **Noyau**. Sur le plan **mathématique**, cette séparation clarifie également l'analyse : la partie $\eta[S(i, j) - \tau \omega]$ diffère de la partie “inhibition”, on peut donc étudier leur impact respectif sur la convergence de manière indépendante, puis considérer leur composition.

D. Extensibilité et Scalabilité

Lorsqu'on parle de **mise à l'échelle** (plus grand nombre d'entités) ou d'**extension** (nouveau type d'entités, nouveau paradigme de synergie), l'architecture modulaire fournit un socle robuste. On peut brancher un “Module Parallélisation” ou un “Module Distribution” (voir §5.2.3.3) pour répartir la matrice ω sur plusieurs nœuds. Le **Noyau** demeure responsable de la logique de mise à jour, mais le “Module Distribution” peut implémenter un mécanisme d'échange ou de partition des données. Sur le plan **mathématique**, on reste fidèle à la même équation, mais on modifie la “plomberie” logicielle.

Cette capacité à empiler de nouveaux modules (Recuit, Observateur, etc.) est cruciale pour un SCN utilisé dans un cadre de recherche à long terme ou un projet industriel, où les besoins évoluent.

E. Conclusion

Une **architecture modulaire** pour un SCN implique :

- Un **Core** (Noyau) centré sur la matrice ω et la boucle d'itérations,
- Des **Modules** (Synergie, Inhibition/Contrôle, Interface, Observateur, etc.) qui viennent se **brancher** dessus et apporter des fonctions spécialisées.

Cette séparation rend la **structure** plus **solide** mathématiquement (chaque composant a des formules et un rôle précis) et **manœuvrable** au niveau ingénierie (chacun s'occupe d'une fonction). Par rapport au **mono-module** (5.2.3.1), on gagne en maintenabilité, extensibilité, et facilité d'évolution, au prix d'un effort de conception initial plus important. Pour des **projets** de recherche avancée ou des applications industrielles (gros volumes, scénarios évolutifs), c'est l'option qui offre le meilleur **équilibre** entre complexité et robustesse.

5.2.3.3. Architecture distribuée (plusieurs sous-SCN coopérant)

Lorsque le **nombre d'entités** devient très grand (n potentiellement de l'ordre de 10^5 à 10^7) ou que les entités elles-mêmes sont **réparties** sur plusieurs sites (multi-robots géographiquement dispersés, données multimodales venant de différentes sources, etc.), il n'est plus forcément possible ni souhaitable de gérer **un** unique SCN centralisé. On peut alors opter pour une **architecture distribuée** où plusieurs **sous-SCN** (chacun manipulant un sous-ensemble des entités ou un certain "domaine") collaborent pour former un ensemble plus vaste.

Cette approche répond aussi à des besoins de **scalabilité** (on ne peut stocker $O(n^2)$ pondérations sur un seul nœud) et de **résilience** (en cas de panne, on ne veut pas perdre tout le SCN). D'un point de vue mathématique, on se situe dans un **système** où la matrice ω est "morcelée" ou "partitionnée", et où les mises à jour dans chaque bloc coopèrent pour tendre vers une organisation globale.

A. Principe : plusieurs sous-SCN

Dans une **architecture distribuée** destinée à un **Synergistic Connection Network (SCN)**, le principe fondamental consiste à **partager** l'ensemble d'entités $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ en plusieurs **sous-ensembles** disjoints, chacun étant géré par un **sous-SCN** local. L'objectif est de répartir la **charge** de calcul et de stockage lorsque le nombre total n devient très grand (plusieurs centaines de milliers, voire des millions d'entités), ou lorsque les entités sont **géographiquement** dispersées (agents robotiques sur différents sites, flux multimédias provenant de régions variées, etc.). Sur le plan **mathématique**, on se retrouve avec une matrice ω qui, au lieu d'être stockée et actualisée de manière unique et centralisée, se trouve morcelée ou **partitionnée** en blocs (sous-matrices), chaque bloc traitant ses propres entités.

Pour mettre en place plusieurs **sous-SCN**, on commence par **diviser** l'ensemble $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ en plusieurs parties $\mathcal{V}_1, \dots, \mathcal{V}_m$, par exemple selon un découpage géographique, un découpage thématique, ou un simple partitionnement en blocs d'effectif à peu près égal. On obtient alors

$$\mathcal{V}_1 \cup \mathcal{V}_2 \cup \dots \cup \mathcal{V}_m = \{\mathcal{E}_1, \dots, \mathcal{E}_n\}, \quad \mathcal{V}_p \cap \mathcal{V}_q = \emptyset \text{ pour } p \neq q.$$

Chaque sous-SCN, noté SCN_p , gère localement les pondérations $\omega_{i,j}^{(p)}$ correspondant aux liens où $i, j \in \mathcal{V}_p$. Sur le plan **stockage**, cela signifie qu'il existe une (sous-)matrice $\omega^{(p)}$ que SCN_p manipule de manière analogue à ce qui a été décrit pour un SCN simple. Sur le plan **algorithmique**, SCN_p exécute sa **boucle** d'auto-organisation (calcul de synergie, mise à jour ω , etc.) au sein du bloc \mathcal{V}_p .

Si les entités i et j appartiennent à des blocs différents \mathcal{V}_p et \mathcal{V}_q , on se heurte à la question : comment traiter $\omega_{i,j}$? Dans un SCN **totale**ment distribué, il est a priori possible que \mathcal{E}_i dans \mathcal{V}_p soit fortement synergique avec \mathcal{E}_j située dans \mathcal{V}_q . On peut concevoir plusieurs schémas.

Le premier schéma consiste à **calculer** (ou au moins **approcher**) les pondérations inter-blocs, c'est-à-dire $\omega_{i,j}$ pour $i \in \mathcal{V}_p$ et $j \in \mathcal{V}_q$. On peut dans ce cas parler de "liens inter-SCN", que l'on stocke dans un "**module passerelle**" ou dans une base partagée. D'un point de vue **mathématique**, chaque SCN_p peut avoir besoin, lors de la mise à jour, d'un terme $\omega_{i,j}$ quand $j \notin \mathcal{V}_p$. Cela suppose des échanges de données entre SCN_p et SCN_q . On peut néanmoins décider d'**ignorer** ou de **sous-échantillonner** ces liens inter-blocs si la synergie est censée être trop faible ou trop rare, réduisant ainsi la complexité globale.

Une autre approche, moins coûteuse en communications, est de n’entretenir les liaisons inter-blocs qu’**épisodiquement** (synchronisation toutes les T itérations), ou de **négliger** explicitement les liens trop distants. On peut écrire, par exemple, un protocole de synchronisation dans lequel SCN_p et SCN_q échangent seulement la portion de ω relative aux entités qui semblent entretenir une synergie notable. Sur le plan **mathématique**, la mise à jour de $\omega^{(p)}(t+1)$ peut faire intervenir un terme de “dépendance inter-blocs” :

$$\omega_{i,j}^{(p)}(t+1) = F^{(p)}\left(\omega_{i,j}^{(p)}(t), S^{(p)}(i,j), \Delta_{\text{inter}}^{(p)}(t)\right),$$

où $\Delta_{\text{inter}}^{(p)}(t)$ regroupe les informations (indices de synergie, pondérations partielles) reçues des autres sous-SCN. Cela se rapproche d’un système **multi-agent** où chaque agent (ici, SCN_p) a sa propre mise à jour locale, mais s’échange parfois des données globales pour harmoniser le tout.

Pour aboutir à une organisation **globale**, il faut que ces sous-SCN ne travaillent pas totalement en vase clos. À intervalles réguliers, on peut effectuer une **coalescence** ou une **synchronisation partielle** : les différents SCN_p consolident les pondérations inter-blocs ou partagent des **vecteurs** d’état (par exemple, la somme $\sum_j \omega_{i,j}^{(p)}$ pour un i qui migrerait dans un autre bloc). D’un point de vue **mathématique**, cela se formalise comme une **itération distribuée** : chaque SCN_p gère les paires internes $\omega_{i,j}^{(p)}$, puis on superpose un opérateur $\text{InterSync}(\omega^{(1)}, \dots, \omega^{(m)})$ qui connecte ou rapproche les valeurs aux frontières. On peut également, dans certains scénarios, faire des “**clusters**” majoritairement internes, en acceptant que les liens inter-blocs soient minimales ou rarissimes.

Sur le plan **ingénierie**, chaque sous-SCN peut être un **processus** (ou un ensemble de threads) tournant sur un nœud différent. Les communications sur le “réseau” (au sens informatique) portent sur les indices d’entités, les bribes de matrice ω , etc. Sur le plan **mathématique**, on se rapproche d’un “réseau de SCN” ou d’un “réseau de nœuds DSL”, chacun pilotant un sous-ensemble.

Lorsqu’il s’avère nécessaire de distribuer le **SCN** pour gérer un **grand** nombre d’entités ou pour implanter le système sur des sites géographiquement dispersés, on procède en créant plusieurs **sous-SCN** SCN_1, \dots, SCN_m . Chacun manipule une portion des entités $\mathcal{V}_p \subset \{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ et stocke localement la sous-matrice $\omega^{(p)}$. Les liens inter-blocs (entités appartenant à des sous-ensembles différents) sont gérés au travers de **protocoles** de synchronisation ou de communications limitées. Sur le plan **mathématique**, on peut écrire la dynamique de chaque sous-SCN comme $\omega^{(p)}(t+1) = F^{(p)}\left(\omega^{(p)}(t), \Omega^{\text{inter}}(t)\right)$, où $\Omega^{\text{inter}}(t)$ regroupe les données échangées avec les autres sous-SCN. Cette approche **distribuée** permet de franchir les limites de mémoire et de calcul qu’impliquerait une matrice $O(n^2)$ trop imposante, tout en préservant une **cohérence** d’ensemble via les mécanismes de coopération entre blocs.

B. Organisation Logicielle d’une Architecture Distribuée

Lorsqu’il devient indispensable de gérer un **Synergistic Connection Network (SCN)** sur plusieurs machines ou serveurs, il est naturel de **distribuer** non seulement les données (entités et sous-matrices ω) mais aussi la boucle même d’auto-organisation du **Deep Synergy Learning (DSL)**. Cette mise en place “multi-nœuds” ou “multi-blocs” doit alors prendre en compte divers aspects : comment **partitionner** les entités, comment synchroniser les sous-SCN locaux, comment véhiculer les pondérations inter-blocs, et comment assurer la **scalabilité** dans un contexte où l’on peut avoir des centaines de milliers ou des millions d’entités globales.

Une architecture distribuée repose d’abord sur la définition de **nœuds** (ou “workers”, “serveurs”), chacun exécutant un **SCN local**. D’un point de vue **logiciel**, on peut imaginer que chaque nœud exécute :

1. Un **Core** local qui gère la sous-matrice $\omega^{(p)}$ correspondant aux entités de son bloc \mathcal{V}_p .
2. Des **Modules** spécialisés (Synergie, Inhibition, Interface) adaptés au bloc local.
3. Un mécanisme de **communication** (via TCP/IP, RPC, ou un bus de messages) pour échanger des données avec les autres nœuds.

Sur le plan **mathématique**, si l’ensemble global $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ est partitionné en $\mathcal{V}_1, \dots, \mathcal{V}_m$, le nœud p prend en charge toutes les liaisons $\omega_{i,j}$ où $i, j \in \mathcal{V}_p$. En local, il opère sa propre boucle “mise à jour $\omega_{i,j}^{(p)}(t+1) = \dots$ ”. Cela donne

naissance à des **sous-SCN** SCN_p . Pour gérer les liens inter-blocs ($i \in \mathcal{V}_p, j \in \mathcal{V}_q$), on recourt à un mécanisme d'échanges entre nœuds.

Le **rythme** des communications inter-nœuds détermine le type de **synchronisation** dans l'architecture distribuée. On discerne deux grandes approches :

1. Approche Sychrone

Tous les sous-SCN SCN_p réalisent leur mise à jour locale (pour leurs entités internes) sur un même “tour” ou “round”, puis s'arrêtent à une “barrière” de synchronisation pour **échanger** ou **agréger** les pondérations (ou au moins les informations pertinentes) concernant les liens inter-blocs. Par exemple, on peut imaginer un cycle :

- (i) Chaque SCN_p met à jour $\omega^{(p)}(t+1)$,
- (ii) Attente d'une barrière,
- (iii) Échange de données sur inter-blocs,
- (iv) Prochaine itération.

Cette approche facilite l'**analyse** mathématique : on peut considérer que chaque round complet aboutit à un opérateur global $G(\omega^{(1)}, \dots, \omega^{(m)})$. En contrepartie, on tolère une **latence** plus élevée (chaque nœud doit attendre les autres avant de poursuivre), ce qui peut ralentir le système pour un grand nombre de nœuds.

2. Approche Asynchrone

Ici, chaque sous-SCN tourne en continu et **demande** les valeurs $\omega_{i,j}$ ou \mathbf{x}_j (pour le calcul de synergie) au fur et à mesure à d'autres nœuds, sans synchronisation globale. Cela signifie qu'on peut avoir des liens $\omega_{i,j}$ “en retard” d'un ou plusieurs tours, donc des **incohérences** momentanées plus élevées. On gagne par contre en **efficacité** si le réseau est très large et que l'on n'a pas envie de bloquer tout le monde à chaque round. Sur le plan **mathématique**, on aboutit à une mise à jour $\omega^{(p)}(t+1) = F^{(p)}(\omega^{(p)}(t), (\text{informations inter-blocs potentiellement datées}))$. L'étude de convergence devient plus complexe, souvent liée aux techniques de systèmes dynamiques asynchrones ou aux algorithmes de type “Gossip”.

Inter-blocs : meta-SCN et résumé des synergies

Pour **connecter** logiquement les sous-SCN, on peut définir un “**meta-SCN**” au niveau supérieur : un graphe dont chaque nœud représente un **bloc** \mathcal{V}_p . Chaque arête représente un **résumé** des liens inter-blocs (par exemple la somme des pondérations $\omega_{i,j}$ pour $i \in \mathcal{V}_p, j \in \mathcal{V}_q$ supérieures à un certain seuil, ou la moyenne des synergies fortes). Ce meta-SCN n'a pas nécessairement le même rôle qu'un SCN classique, mais il peut **guider** l'échange de données : si la synergie inter-blocs $\mathcal{V}_p \leftrightarrow \mathcal{V}_q$ s'avère très faible, on peut réduire la fréquence de synchronisation entre SCN_p et SCN_q .

D'un point de vue **implémentation**, chaque sous-SCN peut posséder un “cache” local des informations sur les entités extérieures (\mathbf{x}_j si on calcule la distance) ou sur $\omega_{i,j}$ inter-blocs, mis à jour selon un protocole de communication. Les mathématiciens y verraient un “système dynamique couplé”, où chaque $\omega^{(p)}$ évolue au gré de données reçues de $\omega^{(q)}$.

Performance et Scalabilité

L'**avantage** principal d'une telle distribution est la **répartition** de la charge : si chaque nœud gère $|\mathcal{V}_p| \approx \frac{n}{m}$ entités, alors la sous-matrice interne $\omega^{(p)}$ représente $O\left(\left(\frac{n}{m}\right)^2\right)$ liens potentiels, ce qui peut devenir **faisable** si m est assez grand pour que $\frac{n}{m}$ demeure raisonnable. Cela permet de traiter un SCN global dont n serait impossible à stocker ou à traiter en un seul bloc mémoire (de l'ordre de $10^5 \sim 10^7$). Les **communications** inter-blocs conservent toutefois un coût non négligeable, que l'on essaie de contrôler (rafraîchissement épisodique, ignorance des synergies trop faibles, compression...).

L'**inconvenient** réside dans la difficulté à assurer la **cohérence** globale : si un agent $i \in \mathcal{V}_p$ veut entretenir une relation forte avec un agent $j \in \mathcal{V}_q$, cela requiert que SCN_p et SCN_q échangent suffisamment (calcul de la synergie, mise à jour conjointe de $\omega_{i,j}$). En mode asynchrone, on accumule d'éventuels retards rendant la convergence mathématique plus délicate. Sur le plan **pratique**, cette distribution reste souvent la seule solution lorsque $O(n^2)$ devient hors de portée sur un unique nœud.

Conclusion (B. Organisation logicielle d'une architecture distribuée)

Pour bâtir un **SCN** distribué, on déploie plusieurs **nœuds**, chacun contenant un **sous-SCN** local (matrice $\omega^{(p)}$, boucle DSL, modules adjoints). Les liens inter-blocs (entre entités se trouvant dans des partitions distinctes) sont gérés via des communications ponctuelles ou régulières, dans un cadre synchrone (barrière à chaque round) ou asynchrone (requêtes à la demande). Cette organisation autorise un **partage** de la mémoire et du calcul, accroissant la **scalabilité** et permettant de gérer un nombre massif d'entités. Elle implique toutefois une **complexité** supplémentaire, tant au niveau **implémentation** (protocole de synchronisation, gestion de la cohérence) qu'au niveau **mathématique** (systèmes dynamiques couplés). Ainsi, l'architecture distribuée prolonge la logique du SCN centralisé vers les **environnements** où la mémoire d'un seul nœud serait trop limitée ou où les entités sont naturellement réparties, tout en préservant, autant que possible, la **cohérence** globale de l'auto-organisation.

C. Scénarios d'Utilisation

Lorsque l'on envisage un **Synergistic Connection Network (SCN)** en mode distribué, plusieurs **scénarios** se présentent où la répartition des entités en différents sous-SCN devient non seulement utile, mais parfois indispensable. Au niveau **mathématique**, tous partagent l'idée d'un **partitionnement** des données ou des agents, chaque sous-SCN gérant localement ses pondérations $\omega^{(p)}$, tandis que les **liens** inter-blocs se manipulent au travers d'échanges. Sur le plan **ingénierie**, cela se traduit par divers contextes d'application (très grand nombre d'entités, systèmes multi-robots, hétérogénéité symbolique/sub-symbolique) nécessitant un **prototype** ou une **infrastructure** distribuée.

1. Très Grand n

Lorsqu'on manipule un **grand** nombre d'entités $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$, l'ordre de grandeur de $O(n^2)$ pour la matrice ω peut rendre impossible son traitement et son stockage en une seule machine. Les ressources (mémoire, CPU) sont vite saturées pour des valeurs de n s'élevant à plusieurs centaines de milliers, voire des millions. Dans ce contexte, on décide de **distribuer** l'ensemble des entités sur plusieurs serveurs (ou nœuds), chacun exécutant un **sous-SCN** local, noté SCN_p . Chacun de ces sous-SCN ne prend en charge que $O((n/m)^2)$ liens internes, ce qui peut devenir gérable si m (le nombre de blocs) est choisi de manière judicieuse.

Cette problématique s'inscrit dans la lignée des **algorithmes distribués** pour la **structure** de graphe ou la **clustering** à grande échelle (type "label propagation", "graph partitioning", etc.). Un **SCN** distribué se rapproche de ces approches, car on maintient des **pondérations** (ou "labels"/"scores") localement et on effectue des synchronisations pour converger vers un état global. Au niveau **mathématique**, on conçoit alors que la dynamique $\omega^{(p)}(t+1) = F^{(p)}(\omega^{(p)}(t), \Omega^{\text{inter}}(t))$ se déroule de façon **parallèle** sur chaque nœud, et l'on obtient une **scalabilité** accrue. Les résultats, en termes de convergence, sont plus complexes à analyser que dans un SCN centralisé, mais cette distribution demeure la seule voie dès que n dépasse largement la capacité d'une seule machine.

2. Contexte Multi-Robot ou Multi-Agents

Dans un cadre **multi-robot** (ou multi-agents, plus général), chaque robot peut être perçu comme un "**nœud**" autonome qui gère, localement, un **mini-SCN** représentant ses capteurs, ses états internes, ou les sous-ensembles d'entités qu'il connaît le mieux (en plus de quelques entités externes qu'il héberge symboliquement). Les **interactions** entre robots sont alors modélisées par des **liens** inter-blocs, réels ou potentiels, qui évoluent via messages d'événements ou mesures coopératives.

Sur le plan **mathématique**, on observe alors un **système** de sous-SCN : chaque robot SCN_p exécute la **boucle** DSL pour ses variables locales ($\omega^{(p)}$), et réalise parfois un échange de données (pondérations "intéressantes", vecteurs de capteurs, etc.) avec ses **voisins**. Les liens inter-robots $\omega_{i \in \mathcal{V}_p, j \in \mathcal{V}_q}$ se mettent à jour moins fréquemment ou selon un protocole asynchrone (messages d'événements). Sur le plan **implémentation**, cela correspond à un ensemble de robots

(ou d'agents virtuels), chacun muni d'un microprocesseur qui fait tourner son **Module Synergie** et sa **Mise à Jour** en local, et qui “**communiquent**” sur le réseau pour partager l'information relative aux liens inter-blocs.

Une telle **distribution** reflète l'autonomie partielle des robots : chaque entité (robot) est libre de se réorganiser en interne, tout en participant à une **organisation** plus large via la synchronisation inter-blocs. Sur le plan **ingénierie**, on peut ainsi construire des essaims robotiques où l'**intelligence** émerge de la coopération de sous-SCN dispersés, chacun actualisant $\omega_{i,j}$ localement tout en échangeant périodiquement les pondérations concernant des robots “voisins”.

3. Hétérogénéité des Domaines

Un autre scénario met en scène un **SCN** segmenté par **type** ou **domaine** d'entités. Par exemple, un sous-SCN $SCN_{subsymbol}$ dédié à des entités sub-symboliques (images, embeddings visuels) et un autre sous-SCN SCN_{symbol} consacré à des entités purement symboliques (règles logiques, concepts, ontologies). Ces deux blocs communiquent par l'intermédiaire d'un “**pont**” ou d'entités hybrides, assurant la **fusion** ou la **traduction** entre représentations sub-symboliques et logiques.

D'un point de vue **mathématique**, chaque sous-SCN gère sa dynamique $\omega^{(p)}$ avec une synergie $S^{(p)}$ adaptée (par exemple, un calcul de similarité cosinus pour le bloc sub-symbolique, et une co-information ou une compatibilité sémantique pour le bloc symbolique). Les **entités** hybrides \mathcal{E}_{hyb} se situent à la frontière : elles possèdent deux représentations (une sub-symbolique, une symbolique) ou sont capables de faire correspondre l'une et l'autre. On peut ainsi définir un **coup** de pont $\omega_{i,j}$ lorsque \mathcal{E}_i réside dans le bloc sub-symbolique et \mathcal{E}_j dans le bloc symbolique, la mise à jour de ce lien pouvant dépend d'un “Module Synergie” spécifique aux transitions visuel \leftrightarrow conceptuel.

Sur le plan **ingénierie**, on réalise deux “clusters” logiques de sous-SCN, chacun spécialisé dans un **domaine** (vision vs. sémantique, par exemple). Sur le plan **mathématique**, on obtient un **réseau** distribué où chaque sous-SCN gère un espace de représentation distinct, et où seuls les **entités-pont** facilitent la synchronisation inter-domaine. Cette hétérogénéité induit souvent des **protocoles** de communication adaptés (ex. on transfère un vecteur embedding quand on veut calculer une synergie cross-domaine). Cela permet au **SCN** global d'intégrer des données multiples (images, texte, logiques de règles) sans forcer un format unique.

Conclusion (C. Scénarios d'Utilisation)

Les **architectures distribuées** pour un SCN couvrent diverses situations : on peut les employer lorsque le **nombre** d'entités (n) dépasse largement la mémoire d'une seule machine, ou quand on opère dans un **contexte multi-robot** ou multi-agents, chaque agent local hébergeant un “sous-SCN” et communiquant pour les liens inter-blocs. Il est aussi possible de segmenter un SCN selon les **domaines** (sub-symbolique vs. symbolique) et de relier ces blocs via des **ponts** ou des entités hybrides. Dans tous ces cas, on préserve la philosophie du DSL (mise à jour de ω , synergie, inhibition éventuelle) tout en déportant une partie des calculs et des structures de données, pour **répondre** aux enjeux de **scalabilité**, de **distribution** géographique ou d'**hétérogénéité** des entités. Sur le plan **mathématique**, cela se traduit par un système dynamique **partiellement** couplé, où chaque bloc évolue localement, et où les **échanges** inter-blocs font émerger (ou non) une cohérence à l'échelle globale.

D. Points Mathématiques et Ingénierie Avancés

Lorsqu'un **Synergistic Connection Network (SCN)** est déployé de manière distribuée, la gestion de plusieurs sous-blocs $\{\omega^{(p)}\}$ et la coordination inter-blocs soulèvent des **questions avancées** à la fois sur le plan mathématique (convergence, synchronisation) et sur le plan d'ingénierie (implémentation de services, répartition des données, etc.). Le comportement global d'un tel SCN résulte alors d'un **système** d'équations ou d'itérations couplées, tandis que chaque sous-SCN local SCN_p maintient ses entités \mathcal{V}_p et ses pondérations internes $\omega^{(p)}$.

Convergence d'un SCN Distribué

Sur le plan purement **mathématique**, la convergence d'un SCN distribué peut s'analyser comme un **système dynamique** multi-blocs. Chaque bloc $p \in \{1, \dots, m\}$ met à jour $\omega^{(p)}$ suivant une équation du type

$$\omega^{(p)}(t+1) = F^{(p)}\left(\omega^{(p)}(t), \Omega^{\text{inter}}(t)\right),$$

où $\Omega^{\text{inter}}(t)$ regroupe les informations relatives aux entités ou pondérations situées dans d'autres blocs. Dans une **forme synchrone**, on suppose que tous les blocs procèdent par “rounds” : chaque SCN_p actualise ses liens internes $\omega^{(p)}$ en fonction de synergies $\mathbf{S}^{(p)}$ et de données inter-blocs reçues à la fin du round précédent. On parle alors d’une “barrière” ou d’un “rendez-vous” qui garantit la cohérence temporelle à chaque itération.

Lorsqu’on passe à une **forme asynchrone** (souvent dite “gossip” ou “push-pull”), chaque sous-SCN peut déclencher sa mise à jour $\omega^{(p)}(t + 1)$ quand bon lui semble, éventuellement en s’appuyant sur des valeurs de $\omega^{(q)}$ (pour $q \neq p$) reçues plus tôt. Le système s’apparente alors à un **système dynamique non autonome**, où la convergence se détermine par l’existence ou non d’une topologie d’échanges suffisamment riche et d’une fréquence de mise à jour inter-blocs assez élevée pour éviter la divergence. Sur le plan théorique, on retrouve des analyses comparables aux algorithmes de type “consensus” ou “gossip” dans les réseaux multi-agents, qui établissent parfois des résultats de convergence (avec ou sans retards) si la matrice d’adjacence globale satisfait certaines conditions de connexité ou de régularité.

Une **difficulté** réside dans le fait qu’un SCN distribué introduit des non-linéarités (inhibition, saturation, etc.), rendant la démonstration de convergence plus ardue qu’un simple consensus linéaire. On peut devoir imposer des **contraintes** (nombre minimal d’échanges par période, borne sur le retard asynchrone, etc.) pour aboutir à un état final stable ou quasi-stable. Les ingénieurs s’arrangent souvent pour trouver un compromis : on exige par exemple que chaque bloc synchronise (même partiellement) ses liens inter-blocs au moins toutes les K itérations, ou on restreint le SCN aux seuls liens inter-blocs au-dessus d’un certain seuil de similarité.

Implémentation et Organisation des Services

Sur le plan de l’**ingénierie**, une architecture distribuée se concrétise par plusieurs “**services**” ou “**modules**” s’exécutant sur des machines distinctes. Chacun possède son **sous-SCN** local, autrement dit une version de la “Core” (voir §5.2.1.1) et éventuellement des modules “Synergie”, “Inhibition”, etc. propres à son sous-ensemble \mathcal{V}_p . Les communications inter-blocs, indispensables pour maintenir la cohérence, s’implémentent via :

1. **Échanges explicites** (RPC, REST, gRPC) : quand un nœud SCN_p veut actualiser $\omega_{i,j}$ pour $i \in \mathcal{V}_p$ et $j \in \mathcal{V}_q$, il interroge un service sur SCN_q pour obtenir la représentation de \mathcal{E}_j ou la pondération précédente.
2. **Synchronisation par rounds** : si la mise à jour est synchrone, on attend la fin d’un “tour” local, puis on échange un “lot” de pondérations inter-blocs.
3. **Structures de cache** : parfois, on stocke (en local) une approximation de $\omega_{i,j}$ inter-bloc, actualisée périodiquement, afin de ne pas surcharger le réseau à chaque requête.

Côté **données**, on peut répartir les entités de façon relativement égale pour équilibrer la charge, ou bien on peut adopter un partitionnement sémantique (les entités “visuelles” sur un cluster, les entités “textuelles” sur un autre). Dans un déploiement massif, chaque bloc local peut gérer $\sim 10^4$ ou $\sim 10^5$ entités, permettant de maintenir $\left(\frac{n}{m}\right)^2$ liaisons internes dans la mémoire d’une seule machine. Les liens inter-blocs $\omega_{i \in \mathcal{V}_p, j \in \mathcal{V}_q}$ se gèrent soit par un stockage partiel (p. ex. un dictionnaire $(i, j) \mapsto \omega_{i,j}$ si $\omega_{i,j}$ est $>$ à un certain seuil), soit par un “recalcul” sur demande en recourant au “Module Synergie” distant.

Applications de l’Architecture Distribuée

Les **applications** d’un SCN distribué se déclinent en plusieurs domaines :

Dans un **Cloud** ou cluster HPC, on veut analyser un énorme graphe ou un volume considérable d’entités. On distribue alors les entités en sous-SCN, chaque sous-SCN tournant sur un ensemble de machines. Les mises à jour itératives de ω ou de $\omega^{(p)}$ progressent en parallèle, synchronisées de temps à autre. Cette configuration répond à un besoin de **scalabilité** : impossible de charger $O(10^{12})$ liaisons sur un unique serveur.

Dans un **environnement multi-robot** (flottille de drones, essaim de robots de service), chaque robot exécute localement un mini-SCN sur ses propres capteurs, ses propres représentations, et discute via messages d’événements ou protocoles ad hoc avec les autres. Les liens inter-robots $\omega_{i \in \mathcal{V}_p, j \in \mathcal{V}_q}$ s’actualisent au fil d’interactions tangibles

(rencontre physique, échange de données). De la sorte, le système complet forme un **SCN** global, potentiellement asynchrone, reflétant la coopération ou la co-information entre robots.

Dans un **contexte Edge computing**, on peut avoir des microcontrôleurs (capteurs, devices IoT) gérant en local un sous-groupe d'entités, puis communiquant avec un "hub" de plus haut niveau. Chaque microcontrôleur agit comme un "petit SCN" local, alors que le hub fait office de "meta-SCN" ou de "SCN central," agrégeant les informations inter-blocs.

Sur le plan **mathématique**, le fonctionnement distribué n'altère pas la logique fondamentale du DSL, mais la décline sous forme d'**itérations couplées**. Sur le plan **implémentation**, on construit un écosystème de **services** (chaque sous-SCN local + un ou plusieurs services de synchronisation) permettant l'apprentissage distribué.

Conclusion (D. Points Mathématiques et Ingénierie Avancés)

Dans une architecture distribuée pour le SCN, les **problématiques** de convergence exigent de considérer un **système** d'équations couplées, que l'on traite de manière synchrone (barrières, rounds) ou asynchrone (approches gossip). Sur le plan **ingénierie**, on met en place des **services** (un sous-SCN local par nœud, un mécanisme de communication inter-blocs) afin de gérer la répartition des entités et la synchronisation nécessaire. Les applications s'avèrent multiples : analyse de grands graphes en **cloud**, systèmes **multi-robots**, configurations **edge** hétérogènes. On apprécie ainsi la **flexibilité** du DSL à se déployer en mode distribué, moyennant des échanges partiels ou complets, et à s'adapter aux différentes topologies de communication. On fait toutefois face à des **défis** supplémentaires : complexité d'implémentation, incohérence passagère en mode asynchrone, et besoin d'un protocole pour gérer les liens inter-blocs. Sur le plan théorique, il reste souvent à **prouver** ou **assurer** la convergence (ou la quasi-convergence) sous certaines hypothèses de connectivité globale, de retard limité, et de fréquence suffisante d'échanges entre sous-SCN.

Conclusion (5.2.3.3)

L'**architecture distribuée** du SCN, fondée sur plusieurs **sous-SCN** coopérant, constitue la **solution** adéquate lorsqu'on atteint de grandes **dimensions** (n énorme), des **contraintes** de localisation (robots dispersés), ou un **besoin** de robustesse (chaque sous-SCN continue si un autre tombe). Sur le plan **mathématique**, on traite alors un système d'équations ou de mises à jour **couplées**, parfois avec synchronisations ponctuelles. Sur le plan **ingénierie**, cela se concrétise par :

- Des **modules** "sous-SCN" déployés sur différents nœuds (serveurs, robots),
- Des **protocoles** de communication (synchronisation, échanges de pondérations ou résumés),
- Une **possible** approche "meta-SCN" au-dessus, gérant les interactions inter-blocs.

Cette architecture s'avère plus **complexe** à mettre en place que les approches **mono-module** (5.2.3.1) ou **modulaire** centralisée (5.2.3.2), mais c'est la **clé** pour la **scalabilité extrême** et l'**adaptation** à des environnements distribués ou massivement parallèles.

5.3. Structures de Données pour la Matrice (ω)

5.3.1. Dense vs. Sparse

- 5.3.1.1. Avantages/Inconvénients d'une **matrice dense** (accès direct, simple) vs. **liste d'adjacence** si la plupart des (ω) sont faibles.
- 5.3.1.2. Critères de choix : taille (n), proportion de liaisons fortes.

5.3.2. Indexation et Accès Rapide

- 5.3.2.1. **Hashmap** ($(i,j) \mapsto \omega_{i,j}$).
- 5.3.2.2. Organisation en “voisinages” (garder les (k) plus forts liens).

5.3.3. Synchronisation en Cas de Distribution

- 5.3.3.1. Si la matrice (ω) est répartie sur plusieurs nœuds : protocole de communication (verrous, versioning).
- 5.3.3.2. Approche asynchrone vs. synchrone pour la mise à jour des liens inter-sous-SCN.

5.3. Structures de Données pour la Matrice ω

Dans un **SCN**, la matrice ω (de dimension $n \times n$) stocke les **pondérations** $\omega_{i,j}$ représentant la force du lien (ou la synergie actualisée) entre chaque entité \mathcal{E}_i et \mathcal{E}_j . Selon la **taille** n et la **densité** réelle des liens (beaucoup ou peu de valeurs non nulles), on choisira une structure de données appropriée (dense, sparse, etc.). Il s'agit d'un choix crucial, car il conditionne à la fois les **performances** (temps de mise à jour, accès mémoire) et la **flexibilité** (ajout/suppression de liens, extension à un mode distribué).

5.3.1. Dense vs. Sparse

Lorsqu'on parle de “**matrice dense**” vs. “**liste d'adjacence**” ou “**structuration sparse**”, on se réfère à des stratégies différentes pour **stocker** la grille $\omega_{i,j}$. Il en découle des **avantages** et **inconvénients** mathématiques et pratiques.

5.3.1.1. Avantages/Inconvénients d'une *Matrice Dense* (Accès Direct, Simple) vs. *Liste d'Adjacence* si la Plupart des ω sont Faibles

Dans le contexte d'un **Synergistic Connection Network (SCN)**, la **représentation** des pondérations $\omega_{i,j}$ devient cruciale lorsque le nombre d'entités n augmente. L'important défi consiste à choisir la structure de stockage la plus adaptée pour ces liaisons, notamment en considérant si la majorité des valeurs $\omega_{i,j}$ sont faibles ou nulles en raison de mécanismes d'**inhibition** ou de régularisation. Deux approches se distinguent dans ce cadre, l'**utilisation d'une matrice dense** et celle d'une **structure de stockage sparse**, telle qu'une **liste d'adjacence** ou un dictionnaire.

A. *Matrice Dense*

La représentation par **matrice dense** consiste à allouer un tableau bidimensionnel $W[n][n]$ dans lequel chaque élément $\omega_{i,j}$ est stocké de manière explicite à l'indice (i, j) . Chaque entité i correspond ainsi à une ligne complète de ce tableau, permettant un **accès direct** aux pondérations avec une complexité en temps de $O(1)$ pour toute lecture ou écriture. Par exemple, la somme des liens sortants d'une entité i se calcule rapidement par

$$\sum_{j=1}^n \omega_{i,j},$$

ce qui peut être optimisé par des techniques de vectorisation, telles que les instructions **SIMD**, ou en utilisant des bibliothèques spécialisées telles que **BLAS**.

Néanmoins, cette approche présente un **inconvénient majeur** : elle est extrêmement **consommatrice de mémoire**. En effet, le stockage d'une matrice dense requiert $O(n^2)$ éléments, ce qui peut rapidement conduire à une utilisation excessive de la mémoire lorsque n est grand (par exemple, quelques dizaines de milliers d'entités entraînant un espace de stockage de plusieurs gigaoctets). De plus, si une grande partie des pondérations reste négligeable (valeurs proches de zéro en raison d'un mécanisme d'inhibition), l'allocation mémoire se trouve en partie gaspillée pour stocker ces zéros, et le parcours complet de la matrice, également en $O(n^2)$, peut s'avérer prohibitif pour des mises à jour fréquentes.

B. *Liste d'Adjacence ou Stockage Sparse*

À l'opposé, le **stockage sparse** ne retient que les paires (i, j) pour lesquelles $\omega_{i,j}$ atteint une valeur significative, par exemple, supérieure à un certain seuil ou figurant dans le « top k » des liaisons pour chaque entité i . Plusieurs formats existent pour implémenter cette approche, notamment le format **CSR (Compressed Sparse Row)**, qui regroupe les liaisons non nulles par ligne, ou encore l'utilisation de dictionnaires associant chaque couple (i, j) à sa valeur correspondante $\omega_{i,j}$.

L'avantage principal d'une telle approche réside dans une **réduction substantielle** de l'utilisation mémoire. Si la densité effective des liaisons est faible, on peut passer d'une complexité en $O(n^2)$ à une complexité en $O(\rho n^2)$ où

$\rho \ll 1$ représente le taux de densité des liaisons significatives. Cette économie de mémoire est particulièrement avantageuse lorsque seules quelques connexions par entité sont réellement non négligeables. Par ailleurs, le parcours des voisins d’une entité i se limite aux entrées présentes dans la liste d’adjacence, ce qui peut considérablement accélérer les algorithmes qui ne nécessitent pas de balayer l’ensemble des n éléments.

Cependant, le **stockage sparse** comporte aussi des inconvénients. L’accès direct à une pondération spécifique $\omega_{i,j}$ n’est plus garanti en $O(1)$ dans tous les cas, en particulier lorsque l’implémentation repose sur une structure de type **hashmap** qui, bien que visant une complexité moyenne de $O(1)$, peut souffrir de surcoûts liés aux collisions ou au ré-hachage. Par ailleurs, lorsqu’on utilise une liste d’adjacence, l’accès à une pondération donnée dépend du nombre de voisins d_i de l’entité i , ce qui introduit une complexité en $O(d_i)$. De plus, la gestion dynamique de ces structures (insertion, suppression ou mise à jour) peut nécessiter des mécanismes supplémentaires, tels que des min-heaps pour conserver les k plus grandes liaisons, ce qui complexifie l’implémentation.

Conclusion (5.3.1.1)

Le choix entre l’utilisation d’une **matrice dense** et d’une **représentation sparse** doit être guidé par le **contexte** d’application. Lorsque n est de taille modérée et que la densité des liaisons fortes est élevée, la simplicité d’accès et l’efficacité d’une matrice dense – avec un accès direct en $O(1)$ – constituent des atouts indéniables. À l’inverse, pour des réseaux de grande taille (avec n atteignant voire dépassant 10^5) où la majorité des valeurs $\omega_{i,j}$ sont faibles ou nulles, l’utilisation d’un **stockage sparse** via liste d’adjacence ou dictionnaire devient prépondérante, permettant d’économiser considérablement de la mémoire et de focaliser les calculs sur les liens significatifs.

D’un point de vue mathématique, la matrice dense représente la solution la plus simple et directe pour accéder aux éléments $\omega_{i,j}$, tandis que la liste d’adjacence offre une efficacité en mémoire et en temps lorsque le réseau se caractérise par une faible densité effective, souvent induite par des mécanismes d’inhibition ou de régularisation. Le choix final doit également prendre en compte d’autres considérations telles que la facilité de persistance des données sur disque ou les exigences spécifiques du traitement des mises à jour. Il est ainsi recommandé que l’**architecture** décrite dans la suite du Chapitre 5 offre la flexibilité nécessaire pour basculer entre ces deux modes de stockage en fonction des besoins sans impacter significativement les modules responsables de la dynamique ou de la synergie.

Ce point de convergence entre **représentation dense** et **représentation sparse** constitue un aspect fondamental de la conception des SCN, et il sera examiné en profondeur dans les chapitres ultérieurs, notamment dans **Chapitre 5**, qui traitera de l’**architecture** générale du SCN, ainsi que dans les sections connexes qui discuteront de la **dynamique** d’auto-organisation et des stratégies d’optimisation des liaisons $\omega_{i,j}$.

5.3.1.2. Critères de Choix : Taille n , Proportion de Liaisons Fortes

Les réflexions portant sur la structure la plus adaptée pour stocker la matrice ω dans un **Synergistic Connection Network (SCN)** s’articulent principalement autour de deux **critères** : la **taille** du réseau (n entités) et la **densité** réelle des liaisons “fortes” (c’est-à-dire la proportion de valeurs $\omega_{i,j}$ restant significatives). Si la taille n demeure modeste ou si, au contraire, la synergie et l’inhibition conduisent à un fort pourcentage de liens non nuls, la **matrice dense** pourra s’avérer plus simple et avantageuse. En revanche, pour un très grand n ou un SCN où l’inhibition et la compétition rendent la plupart des liaisons négligeables, une **représentation sparse** (liste d’adjacence, dictionnaire) permettra d’économiser massivement la mémoire et d’accélérer certaines opérations.

A. Taille n

Lorsque le nombre d’entités n demeure **relativement petit** (typiquement jusqu’à quelques milliers, ou quelques dizaines de milliers), la mise en place d’une **matrice dense** de taille $O(n^2)$ reste tout à fait envisageable. Les mémoires contemporaines peuvent gérer sans difficulté quelques millions d’entrées, et un parcours complet en $O(n^2)$ par itération n’est pas nécessairement prohibitif si on peut paralléliser un peu le calcul (par exemple avec un *double-buffer* synchrone et quelques threads). Dans ce cas, la simplicité d’accès $\omega_{i,j}$ en $O(1)$ justifie largement l’emploi d’une matrice dense si la densité de liens s’avère élevée.

En revanche, si n franchit un seuil critique (quelques dizaines de milliers à plusieurs centaines de milliers), la structure dense devient difficile à stocker (les $O(n^2)$ éléments peuvent représenter des centaines de Go) et le coût de parcours complet en $O(n^2)$ à chaque étape peut s'avérer intenable. Dans un tel scénario, on tentera de **sparsifier** la matrice en ne préservant que quelques liens par entité (typiquement les k liaisons les plus fortes), ce qui rapproche la matrice d'une structure "liste d'adjacence" de taille $O(kn)$.

B. Proportion de Liaisons Fortes (Densité)

On note ρ la proportion de liens "non négligeables" dans le réseau, qu'on peut approximativement évaluer après inhibition ou après un filtrage des liens très faibles. Si ρ est proche de 1, cela signifie que la **plupart** des liaisons $\omega_{i,j}$ sont non nulles et qu'on a un réseau plutôt dense. Une **matrice dense** sera alors cohérente et évitera des surcoûts en recherche ou mise à jour. En revanche, si $\rho \ll 1$, c'est-à-dire que la quasi-totalité de $\omega_{i,j}$ tombent à des valeurs quasi nulles, un **format sparse** (liste d'adjacence, dictionnaire) se révèle bien plus économe en mémoire et plus efficace à parcourir si l'on ne souhaite traiter que les liaisons fortes.

C. Couplage des Deux Critères

Sur le plan **mathématique**, la taille mémoire d'une **matrice dense** est $O(n^2)$, tandis que la taille mémoire d'une **structure sparse** se déduit de $O(\rho n^2)$. Lorsque $\rho \approx 0.001$ (soit 0.1 % de densité), la structure sparse n'occupe qu'un millième de l'espace nécessaire à la dense. À l'inverse, dans un contexte où la densité réelle est $\rho \approx 1$ (ou 50 %), on retombe pratiquement à $O(n^2)$ d'entrées, ce qui enlève l'intérêt d'une représentation "sparse". Les accès aux éléments $\omega_{i,j}$ y deviennent plus complexes (listes, dictionnaires), sans gain notable en mémoire.

La **mise à jour** d'un SCN exige souvent de parcourir toutes les valeurs $\omega_{i,j}$ ou, au moins, toutes les liaisons fortes. Si le réseau demeure *très dense*, ce parcours pourrait rester $O(n^2)$. Dans un format sparse, un parcours complet demande $O(\rho n^2)$, ce qui est avantageux ssi ρ est faible. Pour l'accès "aléatoire" à $\omega_{i,j}$, une matrice dense offre un $O(1)$ direct, alors qu'une structure sparse génère un $O(\log d_i)$ ou $O(d_i)$ (selon les structures) pour retrouver un lien dans la liste d'adjacence.

Conclusion (5.3.1.2)

Le **choix** entre **matrice dense** et **liste d'adjacence** (ou stockage sparse) doit s'appuyer sur :

La **taille** n . En-dessous de quelques milliers à dizaines de milliers, une structure dense peut suffire et simplifier l'implémentation. Au-delà, c'est souvent impraticable d'avoir $O(n^2)$ entrées.

La **densité** ρ . Si la plupart des liaisons $\omega_{i,j}$ restent ténues ou nulles (après inhibition ou top- k forcé), une structure sparse sauvera la mémoire et le temps de calcul en ne stockant que les liens significatifs.

Les **schémas d'opérations** récurrents : un besoin fréquent d'accès aléatoire rapide pointe vers un format dense, un besoin de manipuler seulement les "voisins forts" milite en faveur d'une représentation sparse.

Dans un SCN en pratique, il n'est pas rare qu'au début les données soient manipulées de manière dense, puis qu'on introduise une "sparsification" au fil des itérations, conduisant à une taille $O(kn)$. Certains frameworks permettent même de jongler entre représentations selon l'état courant (chap. 5.3.2), assurant ainsi un usage optimal des ressources et l'adaptabilité du SCN.

5.3.2. Indexation et Accès Rapide

Même après avoir choisi un **format général** (dense vs. sparse) pour la matrice ω (cf. 5.3.1), on peut affiner l'**organisation** interne afin de faciliter les **accès** (lecture/écriture) aux pondérations $\omega_{i,j}$. Dans de nombreux cas, on a besoin d'opérations comme :

- `getWeight(i, j)` : accéder rapidement à la pondération d'un lien particulier,
- `sumOfWeights(i)` : sommer les liens d'une entité \mathcal{E}_i (pour l'inhibition ou la normalisation),

- $\text{topLinks}(i, k)$: retrouver les k plus gros liens sortants de i .

Selon la **densité** et la **nature** de la mise à jour (compétition locale, etc.), on peut adopter des **structures** plus sophistiquées, par exemple un **hashmap** associant $(i, j) \mapsto \omega_{i,j}$ (5.3.2.1) ou une **organisation** en “voisinages” (5.3.2.2).

5.3.2.1. Hashmap $(i, j) \mapsto \omega_{i,j}$

L’une des structures de données *sparse* envisageables pour stocker la matrice ω d’un **Synergistic Connection Network** (SCN) consiste à employer une **table de hachage** (ou *hashmap*) dont la clé est le couple (i, j) et la valeur est la pondération $\omega_{i,j}$. Cette représentation diffère d’un tableau dense ou d’une liste d’adjacence classique, puisqu’on ne stocke que les paires (i, j) jugées réellement significatives, tout en profitant, en moyenne, d’un accès $O(1)$ pour lire ou écrire une entrée dès lors que le hachage est bien conçu.

A. Principe

La matrice ω n’est plus représentée comme un bloc mémoire rectangulaire $W[n][n]$, ni comme un ensemble de listes par lignes, mais comme un **dictionnaire** (ou *hashmap*) associant toute paire $(i, j) \in \{1, \dots, n\}^2$ à la valeur $\omega_{i,j}$. Dans un langage comme C++, on peut employer `std::unordered_map<std::pair<int,int>, double>` ; en Python, un simple *dict* avec pour clé un tuple (i, j) . Les liens dont la pondération $\omega_{i,j}$ demeure faible (inférieure à un certain seuil) peuvent être omis du dictionnaire, réduisant ainsi la taille mémorielle lorsque la densité du SCN est faible.

Le système devient

$$H: (i, j) \mapsto \omega_{i,j}$$

où seules les paires (i, j) jugées “actives” (i.e. $\omega_{i,j}$ non négligeable) apparaissent dans la table. Cela diffère d’un **stockage dense**, qui crée $O(n^2)$ emplacements, et d’une **liste d’adjacence** structurée par ligne, qui assigne explicitement à chaque entité i sa liste de voisins.

B. Avantages

Un **avantage clé** de cette approche réside dans l’**accès direct** à $\omega_{i,j}$ en temps moyen $O(1)$. En effet, la **table de hachage** permet de retrouver la valeur associée à la clé (i, j) via un calcul de hachage, sans devoir chercher dans une liste d’adjacence. Cette propriété diffère notablement d’une liste classique, où l’on itère sur tous les voisins de i pour trouver si j y est présent.

De plus, la **suppression** ou l’**insertion** d’un lien $\omega_{i,j}$ (notamment en cas de mise à zéro ou de réactivation d’un lien suite à un changement de synergie) se fait également en $O(1)$ amorti. On bénéficie donc d’une grande **flexibilité** pour gérer les liens en dynamique : la représentation ne dépend pas de structures chaînées ni de tris partiels, et la table de hachage sait s’ajuster relativement aisément à l’ajout/suppression d’entrées.

Enfin, s’il n’existe qu’un nombre réduit de paires (i, j) réellement non nulles (faible densité ρ), la **mémoire** employée se limite à $O(\rho n^2)$. Cela devient un atout majeur dans les scénarios où l’inhibition ou les mécanismes de **sparsification** maintiennent la plupart des pondérations $\omega_{i,j}$ au seuil de zéro, rendant vaine une structure dense.

C. Inconvénients

Une **table de hachage** n’est pas sans **coût caché**. Le temps moyen d’accès en $O(1)$ s’accompagne d’une constante parfois significative, surtout si la structure devient très large (des millions de paires). Les “*re-hashings*” se déclenchent à mesure que le dictionnaire grossit, occasionnant des *slowdowns* sporadiques. Par ailleurs, contrairement à un tableau 2D, il n’existe pas de *layout* mémoire contigu pouvant tirer parti d’instructions vectorielles (SIMD) ou d’algèbre linéaire BLAS.

Un second souci touche au **parcours** par entité i . Dans une liste d’adjacence structurée, on dispose directement d’un vecteur listant les voisins de i . Dans un dictionnaire global $(i, j) \mapsto \omega_{i,j}$, on ne dispose pas en standard d’une répartition

par *buckets* regroupés. Parcourir l'ensemble des liens de i exige alors soit un balayage complet de la table (avec un filtrage sur la clé), soit le maintien en parallèle d'une structure secondaire (ex. un "adjIndex[i]" pointant vers une liste de clés (i, j)). Cela ajoute de la **complexité** de maintenance.

Enfin, si la densité ρ n'est pas si faible ou qu'elle évolue vers un SCN plutôt dense, la table de hachage devient lourde à manipuler. On perd alors l'avantage d'avoir éliminé $O(n^2)$ emplacements, et on hérite des surcoûts en dispersion de clés, en collisions, etc.

D. Analyse Mathématique et Cas d'Usage

On suppose que ρn^2 paires (i, j) soient actives. Le **coût mémoire** du dictionnaire sera $O(\rho n^2)$, plus un overhead lié aux structures internes de hash. En **accès direct**, on a un temps moyen $O(1)$: on exécute $\omega_{i,j} = H[(i, j)]$. Cette propriété rend la hashmap attractive pour les scénarios où la dynamique du SCN réalise souvent des "random access" sur des paires (i, j) . Toutefois, le "parcours complet" des liens sortants d'une entité i ne s'obtient pas nativement en $O(d_i)$, à moins de maintenir une structure secondaire ou un indice.

Les **contextes** où cette structure se défend bien sont ceux où la densité $\rho \ll 1$ (rendant la matrice dense peu rentable) et où l'on multiplie les accès ponctuels $\omega_{i,j}$ dans le code de mise à jour, tout en ajoutant ou supprimant régulièrement des liens au fil de l'évolution. De plus, l'ajout/suppression d'un couple (i, j) reste simple (opération *insert* ou *erase* en $O(1)$ amorti), contrairement à certaines implémentations de liste d'adjacence devant maintenir un tri par ordre de poids.

Conclusion

L'usage d'une **hashmap** $(i, j) \mapsto \omega_{i,j}$ constitue une **approche sparse** flexible, particulièrement utile pour un SCN si l'on pratique un *random access* fréquent sur de multiples paires, que l'on modifie la liste de liens actifs de manière dynamique, et que la densité demeure relativement faible. Les principaux **bénéfices** sont l'accès (moyennement) direct en temps amorti $O(1)$ et la liberté d'ajouter/supprimer des entrées. Les **limites** concernent le **parcours** par entité i (absence de structure locale) et la surcharge des mécanismes de hachage lorsque le nombre de paires (i, j) devient très grand. Dans une **architecture** modulaire (chap. 5), il est crucial de tenir compte de ces avantages et inconvénients au moment de sélectionner le format de stockage, tout en considérant la question de la densité, des types d'opérations dominantes (accès aléatoire vs. parcours par nœud), et de l'évolution potentielle du réseau au cours du temps.

5.3.2.2. Organisation en "Voisinages" (Garder les k plus forts liens)

Il est souvent observé dans un **Synergistic Connection Network** (SCN) que, pour chaque entité \mathcal{E}_i , seul un nombre relativement restreint de liaisons $\omega_{i,j}$ conservent une valeur suffisamment élevée au fil de la dynamique (additive, multiplicative, etc.). Dans ce cas, la **matrice** ω peut être considérablement épurée en se limitant, pour chaque i , aux k liens les plus forts sortants. Ce principe, parfois décrit sous l'appellation "**k plus forts liens**" ou "**voisinage restreint**", se rattache à l'idée d'une **sparsification** contrôlée : on impose par construction que chaque entité ne conserve qu'un nombre limité de connexions, ce qui réduit la complexité mémoire et facilite la mise à jour.

A. Principe : Conserver un Voisinage Restreint par Entité

Un SCN défini via $\omega_{i,j}(t)$ peut atteindre au plus $O(n^2)$ liens, ce qui devient impraticable si n s'accroît. L'**organisation en voisinage** pose alors la règle :

$$\forall i, \quad \text{voisins}(i) \subseteq \{1, 2, \dots, n\}, \quad |\text{voisins}(i)| \leq k,$$

où k représente un nombre fixé (ou un paramètre qui peut varier selon l'entité). Pour chaque i , on ne conserve que les $\omega_{i,j}$ les plus grands, éliminant les liens trop faibles. Cette notion s'applique de manière dynamique : si une liaison $\omega_{i,j}$ "surpasse" un des liens actuels au fil des itérations, elle peut intégrer le voisinage de i , tandis que le plus faible lien en sort.

B. Aspects Mathématiques et Logiques

La **dynamique** $\omega_{i,j}(t+1)$ continue de suivre les formules usuelles (par exemple additive), mais un “opérateur” TopK_i est appliqué périodiquement (ou à chaque itération) : on trie (ou sélectionne) les liens sortants de i en fonction de $\omega_{i,j}(t+1)$ et on coupe (met à zéro ou ignore) ceux qui ne figurent pas dans les k plus forts. Mathématiquement, cela revient à :

$$\omega_{i,j}(t+1) = \begin{cases} \omega_{i,j}'(t+1) & \text{si } j \in \text{TopK}_i(\omega_{i,j}'(t+1)), \\ 0 & \text{sinon.} \end{cases}$$

où $\omega_{i,j}'(t+1)$ est la mise à jour brute (sans coupes), puis on applique la *sélection* TopK_i . L’effet combiné se rapproche d’une **inhibition latérale** (les liaisons se “battent” pour figurer dans le top- k). Si k est raisonnable (typiquement un ordre de grandeur bien inférieur à n), la structure finale est fortement *sparse*, ce qui diminue le temps de parcours nécessaire pour des opérations (inhibition globale, calcul partiel, etc.).

C. Structure de Données pour le Voisinage

Afin de maintenir ce **voisinage** de taille $\leq k$ pour chaque entité i , on peut employer plusieurs approches :

Un **heap** de taille k conservant les liens sortants de i . Dès qu’un nouveau lien $\omega_{i,j}$ devient plus grand que le minimum du heap, on insère $(j, \omega_{i,j})$ et éjecte l’élément le plus faible.

Une **liste** maintenue triée, bien que les insertions/suppressions puissent devenir plus coûteuses ($O(k)$).

Un **arbre** équilibré (par ex. un *balanced tree*) de taille $\leq k$, permettant insertions et suppressions en $O(\log k)$.

Dans tous ces cas, la mise à jour du top- k pour $\omega_{i,j}(t)$ s’effectue en $O(\log k)$ lorsqu’on envisage d’ajouter ou de remplacer un lien. Ainsi, si l’on effectue un *random access* sur $\omega_{i,j}$ en dehors du top- k , on le considère nul (ou inexistant). Les liens $\omega_{i,j}$ contenus dans le voisinage sont ceux activement mis à jour et considérés comme “vivants”.

D. Bilan sur la Complexité

La **taille mémoire** globale devient $O(kn)$ à la place de $O(n^2)$. Ceci représente un gain appréciable dès lors que $k \ll n$. À chaque itération, chaque entité i manipule $O(k)$ liens dans son voisinage, et la mise à jour (notamment si on parcourt la liste des voisins pour calculer la somme ou vérifier l’inhibition) s’effectue en $O(k)$. Au besoin, si l’on examine la possibilité de faire “entrer” un nouveau lien, on réalise un insertion-suppression dans la structure $O(\log k)$. Ce schéma devient bien plus soutenable qu’un parcours en $O(n)$ ou $O(n^2)$.

E. Considérations pour la Qualité de l’Auto-Organisation

Le fait de ne maintenir que k liens par entité introduit une forme de **compétition** pour les connexions. Cela peut encourager une meilleure différenciation des clusters et limiter la prolifération de liaisons moyennement fortes. Néanmoins, il faut veiller à ce que k ne soit pas trop restreint, sous peine de **fragmenter** exagérément le réseau ou d’empêcher des liens potentiellement utiles de croître. Dans certains SCN, on commence par un k généreux et on le réduit progressivement pour forcer l’apparition de liens *vraiment* robustes, traduisant une phase de “cristallisation” des clusters.

Conclusion

L’**organisation** en “voisinages restreints” (garder, pour chaque entité, les k plus forts liens sortants) constitue un **outil** puissant de *sparsification*. Sur le plan **mathématique**, cela revient à insérer un opérateur *TopK* dans la boucle de mise à jour, imposant un maximum de k connexions par entité. Sur le plan **ingénierie**, cela se traduit par un stockage de type liste/heap limitant la taille à k . Les **avantages** sont d’alléger massivement la mémoire et d’accélérer la mise à jour quand $k \ll n$. Les **inconvénients** tiennent à la charge de maintenir ce top- k et au risque de compromettre l’exploration de certains liens qui commencent faibles mais pourraient devenir forts. Malgré tout, dans la pratique, cette stratégie offre un excellent compromis, permettant d’évoluer vers un réseau *sparse* plus aisément gérable, et favorisant l’émergence de clusters plus nets au sein du SCN.

5.3.3. Synchronisation en Cas de Distribution

Dans certains **scénarios** de grande envergure ou de localisation multiple, on ne peut plus stocker la **matrice** ω en un seul endroit : on la **répartit** alors sur plusieurs **nœuds** (machines, serveurs, robots, etc.) (cf. 5.2.3.3). Cela soulève la question de la **synchronisation** : comment veiller à ce que les mises à jour $\omega_{i,j}(t+1)$ effectuées sur un nœud SCN_p soient **compatibles** avec celles effectuées sur un autre SCN_q , notamment pour les *liens* ou les *entités* se situant à la frontière entre les deux sous-SCN ?

5.3.3.1. Si la Matrice ω est Répartie sur Plusieurs Nœuds : Protocole de Communication (Verrous, Versioning)

Lorsqu'un **Synergistic Connection Network** (SCN) doit gérer un très grand nombre d'entités, il arrive fréquemment que la taille totale de la matrice ω , potentiellement de l'ordre de $O(n^2)$, dépasse les capacités d'une machine unique. Une approche naturelle consiste alors à **distribuer** les données et le calcul sur plusieurs nœuds ou serveurs, chacun prenant en charge un **sous-ensemble** des entités $\{\mathcal{E}_i\}$. Il s'agit cependant de veiller à maintenir une **cohérence** satisfaisante du processus de mise à jour, en particulier pour les liaisons *inter-blocs* reliant deux entités gérées par des nœuds distincts. C'est à ce niveau que se pose la question d'un **protocole** de communication et de synchronisation incluant verrous ou versioning, afin d'éviter les lectures/écritures incohérentes et de reproduire au mieux la vision "synchrone" du SCN.

Répartition de la Matrice ω en Blocs

Le schéma usuel consiste à découper l'ensemble des entités $\{1, \dots, n\}$ en plusieurs blocs $\mathcal{V}_1, \dots, \mathcal{V}_m$, de sorte que chaque sous-SCN, noté SCN_p , manipule localement les pondérations $\omega_{i,j}$ pour $i, j \in \mathcal{V}_p$. Lorsque le réseau est réellement distribué, on confie à chaque nœud du cluster la responsabilité de stocker et mettre à jour la portion $\omega^{(p)}$ qui correspond à son bloc. Se pose alors la question des liens sortants *inter-blocs* : une pondération $\omega_{i,j}$ avec $i \in \mathcal{V}_p$ et $j \in \mathcal{V}_q$ doit être physiquement gérée par un nœud donné, ou éventuellement répartie. Une solution commune est de décider que les liaisons (i, j) se rangent avec l'entité i : si $i \in \mathcal{V}_p$, alors SCN_p stocke toutes les $\omega_{i,j}$, même si j dépend d'un autre nœud. Cette règle de "détenteur unique" évite les doublons, mais contraint le protocole à synchroniser la lecture des données sur les $\omega_{k,j}$ impliquées pour, par exemple, l'inhibition ou d'autres couplages.

Mécanismes de Coordination et de Cohérence

L'enjeu principal est de veiller à ce que la **dynamique** $\omega_{i,j}(t+1) = \dots$ demeure mathématiquement correcte, même si plusieurs nœuds mettent à jour simultanément des portions distinctes de la matrice. Dans un SCN unifié, on postule en général un *pas synchrone* :

$$\omega(t+1) = F(\omega(t)),$$

et on souhaite que l'itération $\omega(t) \mapsto \omega(t+1)$ corresponde à un état global cohérent. Dans un réseau distribué, chaque nœud SCN_p ne détient qu'une partie de $\omega(t)$. Pour cette mise à jour synchrone, on peut imposer un mécanisme de **barrière** : l'état $\omega(t)$ est lu de façon globale et stabilisée, chaque nœud calcule sa portion $\omega^{(p)}(t+1)$ en s'appuyant sur les données inter-blocs, puis un échange de *diffs* ou de trames met à jour les liaisons (i, j) traversant les blocs, enfin tous les nœuds valident l'étape et on obtient un état $\omega(t+1)$ complet. Ce fonctionnement est courant en **MPI** (Message Passing Interface), où le cluster exécute une étape de communication collective après le calcul local. D'un point de vue algorithmique, ce mode "synchrone" assure une cohérence stricte : la mise à jour $\omega(t+1)$ se base intégralement sur $\omega(t)$, et la "vision globale" du temps demeure claire.

Dans certaines infrastructures, un mode *asynchrone* est recherché pour gagner en parallélisme et réduire les coûts de synchronisation. Chaque nœud SCN_p avance alors sa portion $\omega^{(p)}(t+1)$ dès que possible, en récupérant, au fil de l'eau, les versions mises à jour (ou pas) par les autres nœuds. On risque toutefois des phénomènes de lecture/écriture obsolètes, qui altèrent la dynamique. C'est ici qu'intervient la notion de **versioning** : on peut associer un identifiant de "round" ou "version" à l'état $\omega(t)$. Chaque nœud publie sa version ver_p quand il a terminé l'itération t , et ne traite les communications inter-blocs que si elles sont marquées d'une version reconnue (ou plus récente). Cela permet de garder

trace de la progression dans le temps et de savoir si on lit $\omega_{i,j}(t)$ ou $\omega_{i,j}(t-1)$, etc. D'un point de vue mathématique, cette asynchronie fait perdre une partie de la garantie de convergence synchrone, bien que de nombreux systèmes multi-agents adoptent un tel schéma. Les conditions de stabilité deviennent plus délicates à analyser : on se rapproche d'un paradigme "Gossip-based" où chaque nœud a une vision potentiellement décalée des liaisons $\omega_{k,\ell}$.

Verrous et Protocoles de Verrouillage

Lorsqu'on est en présence d'une mise à jour *au fil de l'eau*, une option est d'utiliser un *verrou par bloc* : le nœud SCN_p peut verrouiller la plage mémoire correspondant à $\omega^{(p)}$ avant d'écrire, s'assurer que les autres nœuds ont terminé leurs lectures sur le même round, puis relâcher. Cela évite des écritures concurrentes sur la même portion, au prix de possibles contentions si l'on manipule un grand nombre de blocs. Dans le cadre synchrone plus strict, on recourt à une barrière globale : tous les nœuds terminent le calcul local, communiquent leurs modifications, valident l'étape, et ensuite seulement on passe à l'itération $t+1$. Cela reproduit exactement la formule $\omega(t+1) = F(\omega(t))$, tout en répartissant le calcul.

Exemple Mathématique de Mise à Jour Synchrone Distribuée

On peut imaginer un SCN divisé en m blocs $\mathcal{V}_1, \dots, \mathcal{V}_m$. Chaque sous-SCN SCN_p calcule, pour $(i, j) \in \mathcal{V}_p \times \mathcal{V}_p$, la mise à jour :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)] - \gamma \sum_{k \neq j} \omega_{i,k}(t),$$

puis envoie, pour $(i \in \mathcal{V}_p, j \in \mathcal{V}_q)$ avec $p \neq q$, la nouvelle valeur $\omega_{i,j}(t+1)$ à SCN_q qui mettra à jour la partie inter-blocs si nécessaire. Après ces échanges, tous attendent une barrière. D'un point de vue global, la mise à jour $\omega(t) \mapsto \omega(t+1)$ a bien été calculée selon une étape unique $O(n^2)$ logiquement, mais physiquement $O(1)$ par blocs SCN_p .

Conclusion

Le fait de **répartir** la matrice ω sur plusieurs nœuds oblige à définir un protocole de **communication** et de **cohérence**. La **version synchrone** met en place une barrière à chaque itération, puis échange les liens *inter-blocs* avant de valider $\omega(t+1)$. Ce modèle reproduit la sémantique $\omega(t+1) = F(\omega(t))$ de manière exacte, au prix d'échanges potentiellement importants si la densité de liaisons inter-blocs est forte. L'**asynchronisme**, en revanche, nécessite un système de versioning ou un mécanisme similaire, afin que chaque nœud sache à quel "round" appartient la pondération qu'il lit. Les verrous ou lock-free partiels interviennent alors pour gérer l'écriture concurrente sur des portions de ω . Sur le plan mathématique, on conserve l'idée d'une évolution couplée $\omega^{(p)} \leftrightarrow \omega^{(q)}$, mais on doit se contenter de garanties de convergence moins strictes si on relâche la synchronie totale. Cette problématique, qui touche simultanément l'**ingénierie** (protocole distribué) et l'**analyse** (stabilité asynchrone), illustre l'importance du choix d'un protocole de mise à jour dans la mise en œuvre d'un SCN à très grande échelle.

5.3.3.2. Approche Asynchrone vs. Synchrone pour la Mise à Jour des Liens Inter-sous-SCN

Dans le cadre d'un **SCN** (Synergistic Connection Network) réparti sur plusieurs nœuds ou machines, il apparaît essentiel de gérer la mise à jour des pondérations $\omega_{i,j}$ qui relient des entités appartenant à différents sous-ensembles, notés $\mathcal{V}_1, \dots, \mathcal{V}_m$. Chaque nœud, désigné par SCN_p , est responsable d'une portion locale de la matrice des pondérations, que nous appellerons $\omega^{(p)}$, englobant les liens intra-blocs, c'est-à-dire pour $(i, j) \in \mathcal{V}_p \times \mathcal{V}_p$. Cependant, la gestion des liens inter-blocs, reliant des entités se trouvant sur des nœuds distincts, nécessite une coordination particulière. Deux approches se dessinent : l'une imposant un **synchronisme strict** (méthode « round-based ») et l'autre permettant une **mise à jour asynchrone**, chaque nœud évoluant à son propre rythme.

Dans la méthode **synchrone**, à chaque itération t , tous les nœuds du SCN disposent d'une vision commune et figée de l'état global des pondérations $\omega(t)$. Concrètement, chaque nœud SCN_p lit l'état local $\omega^{(p)}(t)$ et reçoit par communication les valeurs des liaisons inter-blocs, c'est-à-dire celles pour lesquelles $i \in \mathcal{V}_p$ et j appartient à un autre

sous-ensemble \mathcal{V}_q avec $q \neq p$. À partir de ces informations, chaque nœud calcule simultanément sa mise à jour locale conformément à la règle générale de mise à jour du SCN, par exemple sous la forme

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)],$$

et émet les modifications pertinentes vers les autres nœuds pour mettre à jour les liaisons inter-blocs correspondantes. Une barrière de synchronisation (souvent implémentée via des opérations collectives, par exemple *MPI_Barrier* dans les environnements MPI) garantit que l'ensemble du cluster termine sa mise à jour avant de passer à l'itération suivante. Cette approche, qui reproduit rigoureusement le schéma itératif $\omega(t+1) = F(\omega(t))$ défini en section 2.4.1.1, permet d'appliquer les outils théoriques classiques d'analyse de convergence et de stabilité, puisque l'état global est évalué de manière cohérente à chaque round. Néanmoins, le coût de cette synchronisation peut être élevé, en particulier dans des systèmes distribués géographiquement ou hétérogènes, où la latence de communication impose un ralentissement global si un nœud se trouve en retard.

À l'opposé, dans la **mise à jour asynchrone**, chaque nœud évolue de manière autonome sans attendre la fin des calculs des autres. Dans ce cadre, chaque nœud SCN_p maintient son propre compteur d'itérations et met à jour localement les pondérations $\omega_{i,j}$ pour $i \in \mathcal{V}_p$ en utilisant les informations les plus récentes disponibles concernant les liaisons inter-blocs, qui peuvent provenir de communications précédentes et donc être légèrement décalées dans le temps. Formellement, une mise à jour asynchrone peut être modélisée par

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j; \tilde{\omega}(t-\delta)) - \tau \omega_{i,j}(t)],$$

où $\tilde{\omega}(t-\delta)$ représente la version retardée des pondérations provenant d'un autre nœud, et δ est le délai de communication. D'un point de vue mathématique, cette approche se rapproche d'une mise à jour de type Gauss-Seidel ou Gossip, et la convergence de la dynamique peut être garantie sous certaines conditions de contraction et de bornage des retards, comme discuté dans la littérature sur les systèmes distribués asynchrones. Les avantages de cette approche résident dans sa flexibilité, puisqu'aucun nœud n'est contraint d'attendre que tous les autres se synchronisent, ce qui améliore le débit global du système et le rend plus tolérant aux pannes ou aux différences de performances entre nœuds. Cependant, la complexité d'analyse de la convergence augmente, car l'état global ω n'est plus instantanément cohérent et peut évoluer de manière légèrement désynchronisée, ce qui requiert l'introduction de techniques supplémentaires pour contrôler les oscillations ou le décalage temporel.

La **comparaison** entre ces deux approches révèle ainsi un compromis fondamental. Le mode **synchrone** assure une cohérence globale et simplifie l'analyse théorique de la dynamique d'auto-organisation, en restant fidèle à l'équation itérative centralisée $\omega(t+1) = F(\omega(t))$ présentée en section 2.4.1.1. En revanche, le mode **asynchrone** offre une meilleure tolérance aux latences et une plus grande flexibilité de calcul dans des environnements distribués, mais au prix d'une complexité supplémentaire dans l'analyse de la convergence et la gestion des retards.

Conclusion (5.3.3.2)

En résumé, dans les environnements distribués, le choix entre une mise à jour **synchrone** et **asynchrone** des liens inter-sous-SCN dépend du compromis entre la nécessité de conserver une cohérence d'état global et les contraintes pratiques liées à la latence de communication et aux performances hétérogènes des nœuds. La mise à jour synchrone, en imposant des barrières globales, garantit une transition claire et une analyse mathématique rigoureuse de la convergence, mais peut ralentir le système si un nœud est en retard. À l'inverse, la mise à jour asynchrone permet à chaque nœud d'avancer à son rythme, améliorant ainsi le débit global, tout en imposant des défis supplémentaires pour assurer la stabilité et la cohérence du SCN. Cette dualité, abordée ici, sera approfondie dans les **chapitres 5.3.3.3** et ultérieurs, où des approches hybrides, combinant les avantages des deux modèles, pourront être étudiées afin d'optimiser la dynamique d'auto-organisation dans des architectures distribuées.

5.4.1. Séparation du Calcul de Synergie

La **fonction** $S(i, j)$ n'est pas toujours "une simple distance euclidienne" : elle peut être un **score** de similarité cosinus pour des embeddings, une **compatibilité** logique pour des entités symboliques, ou un **degré** de réussite commune en robotique... Mieux vaut donc **isoler** cette logique dans un **module dédié** plutôt que la noyer dans le code de mise à jour de ω .

5.4.1.1. Raisons : la Fonction $S(i, j)$ Dépend du Type d'Entités (Symbolique, Sub-Symbolique)

Le **calcul** de la fonction de **synergie** $S(i, j)$ dans un **Synergistic Connection Network (SCN)** se trouve au cœur de la dynamique d'auto-organisation (chap. 4). En pratique, les entités \mathcal{E}_i peuvent revêtir des formes très différentes : certaines seront *sub-symboliques*, d'autres *symboliques*, et le réseau peut même cohabiter avec des entités hybrides ou d'autres types plus spécialisés. Cette diversité amène à séparer le **calcul** de $S(i, j)$ dans un **module dédié**, plutôt que de l'implémenter de manière monolithique dans le cœur du SCN.

Il est fréquent qu'un SCN doive manipuler des entités sub-symboliques, comme des vecteurs d'embeddings pour décrire des images ou des documents, et simultanément des entités symboliques, ancrées dans un ensemble de règles logiques ou de concepts ontologiques. Dans la sphère sub-symbolique, la *synergie* entre deux entités $\mathcal{E}_i, \mathcal{E}_j$ prend couramment la forme d'une **distance** (euclidienne, Minkowski, etc.) ou d'une **similarité** (cosinus, RBF kernel, etc.), éventuellement transformée pour la ramener à un score positif ou dans $[0,1]$. Dans la sphère symbolique, la synergie peut provenir de la **cohérence** (quantité de conflits logiques, alignement conceptuel, correspondances sémantiques), conduisant à un autre type de calcul. Situer les deux extrêmes (sub-symbolique vs. symbolique) dans une même fonction S réclame parfois une combinaison (pondérée ou non) de différentes mesures. Cette hétérogénéité se formalise plus aisément si un *module de synergie* distinct prend en charge la logique de "comment calculer $S(i, j)$ selon la nature des entités $\mathcal{E}_i, \mathcal{E}_j$ ".

Sur le plan de l'**ingénierie logicielle**, une architecture monolithique exigerait d'inscrire tous les calculs de S dans le *noyau* du SCN. Cela devient peu extensible : dès que l'on introduit un nouveau type d'entité (par exemple un nouveau format sub-symbolique, ou une nouvelle sémantique symbolique), on serait contraint de modifier des parties centrales du code. En revanche, en **séparant** le *Module de Synergie*, on fournit un point d'entrée unique pour tout nouveau schéma de calcul S . Chaque type d'entité peut offrir sa propre méthode de similarité, ou on peut définir un "composite" s'il s'agit d'entités hybrides. Cette modularisation respecte la logique exposée au chapitre 5 : on *encapsule* les dépendances et rend possible l'évolution ou la substitution du module de synergie sans affecter la dynamique $\omega_{i,j}$ elle-même.

Sur le plan mathématique, vouloir assujettir toutes les entités à un unique calcul S peut s'avérer très lourd si l'on mélange sub-symbolique et symbolique dans un même référentiel. En dissociant ce *module de synergie*, on obtient une brique que l'on peut étudier ou tester en isolation. Il est alors plus simple d'exiger certaines propriétés (bornes, symétrie, pseudo-métrie, etc.) pour S , voire de prouver qu'elle satisfait des conditions facilitant la convergence (par exemple un certain caractère contractant ou monotone). L'architecture modulaire valorise cette démarche : la partie centrale du SCN ne se questionne pas sur la "substance" de S , elle suppose seulement qu'à chaque itération, pour toute paire (i, j) , le *module de synergie* fournit un score dans un intervalle voulu (\mathbb{R}^+ , $[0,1]$, etc.).

Conclusion

Le fait de *séparer* la logique de **calcul** de la *synergie* $S(i, j)$ dans un **module** autonome répond à trois motifs fondamentaux. D'abord, la **nature** hétérogène des entités (symboliques, sub-symboliques, ou hybrides) appelle des calculs multiples et rend périlleux l'idée d'une unique formule universelle codée en dur. Ensuite, la nécessité d'**évolutivité** (pouvoir introduire de nouveaux types d'entités ou modifier la manière de calculer S) conduit naturellement à isoler cette portion du code. Enfin, envisager S comme un *module* distinct favorise la **cohérence mathématique** et la démonstration de propriétés (normes, distances, etc.), dans la mesure où le *noyau* du SCN n'a pas à gérer ces spécificités. Cet agencement s'inscrit dans la logique globale du chapitre 5, dédiée à la construction d'une architecture modulaire et pérenne pour le SCN.

5.4.1.2. Maintenabilité et Extensibilité

La **séparation** du calcul de la fonction $S(i, j)$ dans un module dédié ne s’explique pas seulement par la différence des types d’entités (symboliques vs. sub-symboliques) mentionnée en (5.4.1.1). Un second motif repose sur la **maintenabilité** et la **flexibilité** à long terme, tant du point de vue mathématique que du point de vue de l’ingénierie logicielle. Le fait d’extraire la logique de calcul de S du cœur du Synergistic Connection Network (SCN) offre des garanties de facilité d’évolution, de tests et de documentation, assurant que l’ajout ou la modification de la fonction S n’entraîne pas un remaniement lourd de la dynamique globale.

A. Maintenabilité

Une première dimension tient à la probabilité que la fonction de **synergie** $S(i, j)$ doive évoluer au cours de la vie d’un projet ou d’une plate-forme de recherche. Dans de nombreux contextes, la manière de mesurer la cohérence entre deux entités $\mathcal{E}_i, \mathcal{E}_j$ n’est pas figée : on découvre de nouvelles mesures mieux adaptées, on veut combiner plusieurs critères (sub-symbolique, information mutuelle, co-occurrence symbolique), ou encore adapter la pondération en fonction d’un contexte externe. Si le code calculant S est dispersé dans les routines de mise à jour $\omega_{i,j}(t + 1)$, chaque modification risque de se répercuter dans un ensemble de classes ou de fichiers, augmentant fortement la probabilité de régressions ou d’incohérences.

En **séparant** la logique dans un “**module de synergie**”, la mise à jour $\omega_{i,j}(t + 1)$ (vue au chapitre 4) se borne à invoquer une fonction ou une interface de type $S(i, j)$. Les opérations internes à la dynamique, comme le calcul d’un terme $\eta[S(i, j) - \tau \omega_{i,j}]$, n’ont pas besoin de connaître les détails de la sous-routine qui évalue la similarité ou la compatibilité. Cela facilite la **maintenance** : si S doit être enrichi, remplacé, ou paramétré plus finement, on ajuste ce seul module. Sur le plan des **tests unitaires**, il devient possible de vérifier séparément que le module Synergie délivre les scores attendus selon divers scénarios, sans déployer toute l’infrastructure du SCN (grande matrice ω , inhibition, etc.). Un correctif sur la fonction S n’affectera pas les autres pans du code, préservant ainsi la stabilité du système.

B. Extensibilité

Une seconde dimension touche à l’**extensibilité** : le système doit pouvoir absorber de **nouveaux** types d’entités ou de **nouvelles** formules de synergie. Un SCN peut commencer en manipulant des entités sub-symboliques vectorielles (ex. embeddings d’images) et, plus tard, intégrer des entités purement symboliques, ou hybrides (avec deux volets de représentation). Si la logique de calcul de S reste centralisée, tout ajout de type exigerait de retravailler le code source à de multiples emplacements. En revanche, dans une approche modulaire, on peut imaginer un *registre* ou une *interface polymorphe* : chaque type d’entité, ou chaque couplage de types (sub-symbolique vs. symbolique), peut déclarer ses routines de calcul $S(i, j)$. L’architecture du SCN sollicite ce module de manière unifiée, évitant les modifications lourdes.

Parallèlement, si l’on veut combiner différentes mesures, par exemple

$$S_{\text{hybrid}}(i, j) = \alpha S_{\text{sub}}(i, j) + (1 - \alpha) S_{\text{sym}}(i, j),$$

on peut créer un **module hybride** qui, au moment du calcul, appelle le module “sub” et le module “sym” avant de fusionner les scores. Cet emboîtement de modules apporte une souplesse maximale, permettant de tester ou de raffiner la pondération α sans altérer la dynamique $\omega_{i,j}(t + 1)$.

Cette **extensibilité** correspond à la logique “ouvert/fermé” bien connue en ingénierie logicielle : on est ouvert à l’ajout de nouvelles implémentations de S sans être contraint de modifier le noyau du SCN, qui se contente d’invoquer le module défini par l’utilisateur ou l’équipe de recherche.

C. Vue Globale Math-Implémentation

Sur le plan **mathématique**, envisager la fonction S comme un module isolé clarifie l’analyse de la mise à jour $\omega_{i,j}(t + 1)$. On peut poser comme “**contrat**” que $S(i, j) \in [0, 1]$ ou \mathbb{R}^+ , éventuellement bornée par 1, et la dynamique du SCN suppose $\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \dots$. Les propriétés de convergence, de stabilité, ou de contraction que l’on étudie (chap. 4) dépendent alors d’hypothèses imposées à S (ex. Lipschitz, monotone, etc.), sans exiger de connaître

les détails internes : une entité purement symbolique ou un embedding vectoriel peut satisfaire le même cahier des charges, du moment que le score se conforme à l'API du module.

Côté **implémentation**, cela signifie doter le *Module Synergie* d'une interface explicite (ex. *double synergy(int i, int j)*) ou d'un mécanisme polymorphe selon la nature des entités. Le *Noyau* ne se préoccupe pas de la façon dont on compare deux entités ; il demande juste “donnez-moi un score de synergie”. Le *Module Synergie* se charge de “dispatch” ou de prise de décision si \mathcal{E}_i est sub-symbolique, symbolique, etc. C'est lui également qui gère l'évolutivité : si l'on veut substituer une nouvelle mesure ou en combiner plusieurs, on peut l'implémenter ici.

Conclusion

La **maintenabilité** et l'**extensibilité** constituent des motivations majeures pour placer le **calcul** de la fonction $S(i, j)$ dans un **module** séparé. D'une part, cette approche confère une grande **stabilité** à la codebase : faire évoluer S ou en ajouter de nouvelles versions ne perturbe pas la dynamique $\omega_{i,j}$. D'autre part, cette structuration favorise l'**ouverture** à de multiples types d'entités et de mesures de synergie, essentiel lorsque l'on manipule des représentations hétérogènes (sub-symboliques, symboliques, ou hybrides) ou lorsque l'on explore de nouvelles combinaisons (chap. 3). Au niveau **mathématique**, la modularité met en exergue la brique “ $S(i, j) \mapsto \omega_{i,j}(t + 1)$ ” sous forme d'un *opérateur de mise à jour*, auquel on peut simplement rattacher la fonction S , distincte et interchangeable, rendant la conception plus cohérente et évolutive sur le long terme.

5.4.2. Méthodes d'Implémentation

Au-delà du **principe** de séparer la logique de la synergie S (5.4.1), on doit choisir **comment** l'implémenter concrètement dans le code. Deux grandes approches se distinguent : l'**Approche Polymorphique** (5.4.2.1) et le **Module Central** (5.4.2.2). La première mise en œuvre est fondée sur l'idée que chaque **entité** sait elle-même évaluer la synergie avec une autre, tandis que la seconde recourt à un **orchestrateur** unique identifiant les types et appliquant la bonne fonction S .

5.4.2.1. Approche Polymorphique : chaque entité sait calculer la synergie avec une autre

Lorsqu'un **Synergistic Connection Network** (SCN) doit intégrer des entités de types hétérogènes – par exemple, des entités **sub-symboliques** issues d'embeddings, des entités **symboliques** définies par des règles, ou encore des entités hybrides qui combinent ces deux aspects – il apparaît judicieux d'adopter une approche **polymorphique** dans la conception logicielle. Cette approche consiste à doter chaque entité de la capacité de calculer elle-même sa **synergie** avec une autre entité, c'est-à-dire de déterminer la valeur de $S(i, j)$ en fonction de sa propre représentation et de celle de son interlocuteur.

L'idée fondamentale repose sur la définition d'une **classe de base** ou d'une **interface** commune, que l'on peut nommer par exemple *Entity*. Cette classe déclare une méthode virtuelle (ou abstraite) destinée à être surchargée par chaque sous-classe spécifique. En notation mathématique, on souhaite qu'une entité \mathcal{E}_i dispose d'une fonction

$$S(i, j) = f(\mathcal{E}_i, \mathcal{E}_j),$$

qui exprime la **similarité** ou la **compatibilité** entre \mathcal{E}_i et \mathcal{E}_j . En termes de programmation orientée objet, on peut définir :

```
class Entity { virtual double synergyWith(const Entity& other) const = 0; ... };
```

Chaque type d'entité hérite de cette classe et fournit sa propre implémentation de la méthode *synergyWith*. Par exemple, une **SubEntity** peut être définie pour des données sub-symboliques, stockant un vecteur $\mathbf{x}_i \in \mathbb{R}^d$. Sa méthode de calcul de synergie peut reposer sur la **similarité cosinus**, telle que :

$$S_{\text{sub}}(i, j) = \frac{\mathbf{x}_i \cdot \mathbf{x}_j}{\|\mathbf{x}_i\| \|\mathbf{x}_j\|}.$$

En revanche, une **SymEntity** représentant une entité symbolique, caractérisée par un ensemble de règles \mathcal{R}_i , peut implémenter la méthode *synergyWith* en fonction d'un indice de compatibilité logique, par exemple :

$$S_{\text{sym}}(i, j) = 1 - \frac{\text{nbContradictions}(\mathcal{R}_i, \mathcal{R}_j)}{\text{nbTotal}(\mathcal{R}_i \cup \mathcal{R}_j)}.$$

Dans le cas où une entité sub-symbolique doit être comparée à une entité symbolique, il est alors possible de définir une fonction hybride $S_{\text{hybrid}}(i, j)$ qui combine de manière pondérée les deux critères, par exemple par :

$$S_{\text{hybrid}}(i, j) = \alpha S_{\text{sub}}(i, j) + (1 - \alpha) S_{\text{sym}}(i, j),$$

avec $\alpha \in [0,1]$ déterminant l'importance relative de la composante sub-symbolique par rapport à la composante symbolique.

Pour que la méthode *synergyWith* prenne correctement en compte non seulement le type de l'objet courant (celui qui appelle la méthode) mais également le type de l'objet passé en argument, on peut recourir à des techniques de **double dispatch** ou au *Visitor pattern*. Ainsi, si l'on a deux entités de types différents, par exemple une SubEntity et une SymEntity, l'appel

```
double s = e1->synergyWith(*e2);
```

se résoudra dynamiquement en invoquant la méthode adaptée à la combinaison des types, par exemple en appelant *e2.synergyWithSubEntity(*e1)* ou une méthode équivalente, de façon à ce que le calcul de $S(i, j)$ soit correctement orienté.

Cette approche présente de nombreux atouts sur le plan de la **cohésion** et de l'**extensibilité**. Chaque classe d'entité intègre la connaissance de sa propre représentation et de la méthode la plus pertinente pour mesurer sa synergie avec une autre entité, ce qui garantit une haute **cohérence** entre la structure interne d'une entité et la manière dont elle interagit. De plus, l'**extensibilité** est facilitée puisque l'ajout d'un nouveau type d'entité ne nécessite que la création d'une nouvelle classe dérivée implémentant *synergyWith*. En revanche, il faut parfois recourir à des mécanismes de **double dispatch** pour gérer les interactions entre types hétérogènes, ce qui peut introduire une complexité supplémentaire au niveau du design.

Voici un extrait de code illustratif en C++ :

```
// Interface de base pour une entité
class Entity {
public:
    virtual ~Entity() {}
    virtual double synergyWith(const Entity &other) const = 0;
};

// Sous-classe pour une entité sub-symbolique
class SubEntity : public Entity {
public:
    std::vector<double> x; // Représentation par un vecteur

    SubEntity(const std::vector<double>& vec) : x(vec) {}

    // Méthode pour calculer la similarité cosinus
    virtual double synergyWith(const Entity &other) const override {
        const SubEntity* subOther = dynamic_cast<const SubEntity*>(&other);
        if(subOther) {
            return cosineSimilarity(this->x, subOther->x);
        }
        // Cas hybride ou incompatibilité : retourner une valeur par défaut ou calculer autrement
        return 0.0;
    }
};
```

```

    }
};

// Sous-classe pour une entité symbolique
class SymEntity : public Entity {
public:
    std::set<std::string> rules; // Ensemble de règles ou axiomes

    SymEntity(const std::set<std::string>& r) : rules(r) {}

    // Méthode pour calculer la compatibilité logique
    virtual double synergyWith(const Entity &other) const override {
        const SymEntity* symOther = dynamic_cast<const SymEntity*>(&other);
        if(symOther) {
            return computeLogicalCompatibility(this->rules, symOther->rules);
        }
        return 0.0;
    }
};

```

Dans cet exemple, la fonction *cosineSimilarity* calcule la similarité entre deux vecteurs, tandis que *computeLogicalCompatibility* évalue la compatibilité entre deux ensembles de règles. On peut bien entendu complexifier ces fonctions pour tenir compte de schémas hybrides ou de pondérations adaptatives.

De même, en Python, on peut définir une structure de classes pour adopter une approche polymorphique. Voici un exemple succinct :

```

import numpy as np
from abc import ABC, abstractmethod

class Entity(ABC):
    @abstractmethod
    def synergy_with(self, other) -> float:
        pass

class SubEntity(Entity):
    def __init__(self, x: np.ndarray):
        self.x = x # vecteur d'embedding

    def synergy_with(self, other: Entity) -> float:
        if isinstance(other, SubEntity):
            return np.dot(self.x, other.x) / (np.linalg.norm(self.x) * np.linalg.norm(other.x))
        # Pour le cas hybride, on peut définir un score fixe ou une fonction mixte
        return 0.0

class SymEntity(Entity):
    def __init__(self, rules: set):
        self.rules = rules

    def synergy_with(self, other: Entity) -> float:
        if isinstance(other, SymEntity):
            nb_total = len(self.rules.union(other.rules))
            if nb_total == 0:
                return 1.0
            nb_common = len(self.rules.intersection(other.rules))
            # On définit la compatibilité comme le rapport d'intersection sur union
            return nb_common / nb_total

```

```
return 0.0
```

```
# Exemple d'utilisation
```

```
entity1 = SubEntity(np.array([1.0, 0.0, 0.0]))
entity2 = SubEntity(np.array([0.8, 0.1, 0.0]))
entity3 = SymEntity({"rule1", "rule2", "rule3"})
entity4 = SymEntity({"rule2", "rule4"})

print("Synergie entre SubEntity:", entity1.synergy_with(entity2))
print("Synergie entre SymEntity:", entity3.synergy_with(entity4))
```

Ici, la méthode *synergy_with* est définie de manière polymorphique dans chacune des classes dérivées, et permet de calculer le score de synergie selon la nature de l'entité. Dans un SCN complet, la boucle de mise à jour du réseau se contenterait d'appeler cette méthode pour chaque paire d'entités, sans avoir à connaître les détails internes de leur représentation.

Conclusion

L'**approche polymorphique** permet ainsi de décentraliser le calcul de la synergie en déléguant la responsabilité à chaque entité, qui connaît le mieux sa propre représentation et sait comment la comparer à celle d'un autre objet. Cette méthode, qui repose sur le principe de **résolution dynamique** (dynamic dispatch), facilite l'extension du SCN à des environnements où coexistent des entités de types divers (sub-symboliques, symboliques ou hybrides). Bien que cette approche puisse nécessiter l'emploi de techniques avancées telles que le **double dispatch** pour gérer toutes les combinaisons possibles, elle présente l'avantage de maintenir une **cohérence** interne élevée dans le code et de permettre une **extensibilité** naturelle. Ce paradigme unifie le calcul de la synergie dans un contexte de programmation orientée objet, tout en permettant de conserver une interface uniforme pour la mise à jour des pondérations du SCN.

5.4.2.2. Module Central : un orchestrateur identifie les types et applique la bonne fonction S

Dans un **Synergistic Connection Network (SCN)**, la capacité à quantifier la **synergie** entre deux entités \mathcal{E}_i et \mathcal{E}_j est cruciale pour la mise à jour des pondérations $\omega_{i,j}$. Lorsque les entités proviennent de domaines hétérogènes – qu'il s'agisse de représentations **sub-symboliques**, **symboliques** ou **hybrides** – la logique de calcul de la synergie ne peut plus être implémentée de manière éparse dans chaque classe. Au lieu de cela, une approche centralisée est envisagée afin de regrouper l'ensemble des règles de calcul dans un **module central** qui, tel un orchestrateur, identifie les types d'entités et applique la fonction S appropriée pour chaque paire. Nous développerons ici le principe de cette solution, ses avantages théoriques et pratiques, puis nous illustrerons le concept par un exemple de code en Python.

A. Principe d'un Orchestrateur Central

L'idée fondamentale consiste à dissocier la **logique de calcul de la synergie** des classes d'entités elles-mêmes. Plutôt que de faire appel à un polymorphisme distribué – où chaque entité doit implémenter une méthode de comparaison prenant en compte toutes les combinaisons de types (ce qui conduit souvent à une complexité de *double dispatch*) – le SCN intègre un **Module Central**, ici dénommé *SynergyOrchestrator*. Ce module agit comme une table de correspondance, en associant à chaque couple de types d'entités ($\text{type}_i, \text{type}_j$) une fonction spécifique de calcul de la synergie. Autrement dit, pour des entités \mathcal{E}_i et \mathcal{E}_j , le module central procède ainsi :

$$S(\mathcal{E}_i, \mathcal{E}_j) = f_{(\text{type}_i, \text{type}_j)}(\mathcal{E}_i, \mathcal{E}_j),$$

où $f_{(\text{type}_i, \text{type}_j)}$ désigne la fonction adéquate pour traiter la combinaison des types. Par exemple, si \mathcal{E}_i est une **SubEntity** et \mathcal{E}_j une **SymEntity**, la fonction $f_{(\text{Sub}, \text{Sym})}$ sera appelée pour renvoyer un score, qui pourra être défini, dans un cas simple, comme une combinaison pondérée des scores sub-symboliques et symboliques :

$$S_{\text{hybrid}}(i, j) = \alpha S_{\text{sub}}(i, j) + (1 - \alpha) S_{\text{sym}}(i, j),$$

avec $\alpha \in [0,1]$. Le rôle du *SynergyOrchestrator* est donc de recevoir les identifiants de type fournis par chaque entité, de consulter une table de correspondance interne et de déléguer le calcul à la fonction adéquate.

B. Avantages de la Centralisation de la Logique de Synergie

La centralisation de la logique de calcul présente plusieurs avantages majeurs. Premièrement, la **cohésion** du code est grandement améliorée : toutes les règles de calcul sont concentrées dans un même module, ce qui simplifie la maintenance et la compréhension. En cas d'évolution du modèle ou de la nécessité d'ajouter un nouveau type d'entité, il suffit d'étendre la table de correspondance du module central plutôt que de modifier le code de chacune des classes. Cette approche favorise également une **extensibilité** naturelle puisque, pour ajouter un nouveau type tel que « HybridEntity », il est nécessaire d'implémenter les fonctions spécifiques $f_{(\text{Hybrid}, \cdot)}$ et $f_{(\cdot, \text{Hybrid})}$ dans le module central. De plus, la **séparation** entre les données (contenues dans les entités) et la logique de comparaison (gérée par le module central) permet d'obtenir une architecture plus *data-centric* sur le plan mathématique, dans laquelle la fonction S apparaît comme un opérateur unique paramétré par les attributs des entités.

C. Implémentation en Python

Pour illustrer ce concept dans un environnement de prototypage rapide, considérons un exemple en Python qui met en œuvre ce module central. L'exemple suivant présente une version simplifiée où chaque entité expose un identifiant de type et des données internes (par exemple, un vecteur pour une SubEntity ou un ensemble de règles pour une SymEntity). Le module central effectue un *dispatch* en fonction des types et appelle la fonction de calcul appropriée.

```
import numpy as np
```

```
# Définition de constantes pour les types d'entités
```

```
TYPE_SUB = "SubEntity"
```

```
TYPE_SYM = "SymEntity"
```

```
# Fonction de calcul de la similarité cosinus pour des vecteurs (pour SubEntity)
```

```
def calc_sub_sub(entity1, entity2):
```

```
    x1, x2 = entity1.data, entity2.data
```

```
    norm1, norm2 = np.linalg.norm(x1), np.linalg.norm(x2)
```

```
    if norm1 == 0 or norm2 == 0:
```

```
        return 0.0
```

```
    return np.dot(x1, x2) / (norm1 * norm2)
```

```
# Fonction de calcul de la compatibilité logique pour des ensembles de règles (pour SymEntity)
```

```
def calc_sym_sym(entity1, entity2):
```

```
    rules1, rules2 = entity1.data, entity2.data
```

```
    union_rules = rules1.union(rules2)
```

```
    if not union_rules:
```

```
        return 1.0 # Si aucun règle, considérer la compatibilité maximale
```

```
    common_rules = rules1.intersection(rules2)
```

```
    return len(common_rules) / len(union_rules)
```

```
# Fonction hybride pour le cas SubEntity vs SymEntity (ici, on retourne une valeur mixte)
```

```
def calc_sub_sym(entity1, entity2):
```

```
    # Par exemple, retourner 0.0 pour indiquer qu'on ne rapproche pas sub-symbolique et symbolique
```

```
    return 0.0
```

```
# Module central d'orchestration de la synergie
```

```
class SynergyOrchestrator:
```

```
    def __init__(self, alpha=0.5):
```

```
        self.alpha = alpha
```

```
        # Table de correspondance : (type1, type2) -> fonction de calcul
```

```
        self.dispatch_table = {
```

```
            (TYPE_SUB, TYPE_SUB): calc_sub_sub,
```

```

        (TYPE_SYM, TYPE_SYM): calc_sym_sym,
        (TYPE_SUB, TYPE_SYM): calc_sub_sym,
        (TYPE_SYM, TYPE_SUB): lambda e1, e2: calc_sub_sym(e2, e1)
    }

    def compute_synergy(self, entity1, entity2):
        type1 = entity1.get_type()
        type2 = entity2.get_type()
        key = (type1, type2)
        if key in self.dispatch_table:
            return self.dispatch_table[key](entity1, entity2)
        else:
            # Cas par défaut
            return 0.0

# Classes d'entités utilisant une approche minimaliste
class Entity:
    def __init__(self, entity_type, data):
        self.entity_type = entity_type
        self.data = data # Peut être un vecteur (np.array) ou un ensemble de règles (set)

    def get_type(self):
        return self.entity_type

# Exemple d'entités sub-symboliques et symboliques
sub_entity1 = Entity(TYPE_SUB, np.array([1.0, 0.0, 0.0]))
sub_entity2 = Entity(TYPE_SUB, np.array([0.8, 0.1, 0.0]))
sym_entity1 = Entity(TYPE_SYM, {"rule1", "rule2", "rule3"})
sym_entity2 = Entity(TYPE_SYM, {"rule2", "rule4"})

# Création de l'orchestrateur de synergie
orchestrator = SynergyOrchestrator()

# Calcul des synergies pour différentes paires
synergy_ss = orchestrator.compute_synergy(sub_entity1, sub_entity2)
synergy_qq = orchestrator.compute_synergy(sym_entity1, sym_entity2)
synergy_hybrid = orchestrator.compute_synergy(sub_entity1, sym_entity1)

print("Synergie entre deux SubEntities :", synergy_ss)
print("Synergie entre deux SymEntities :", synergy_qq)
print("Synergie entre une SubEntity et une SymEntity :", synergy_hybrid)

```

Dans cet exemple, le **SynergyOrchestrator** utilise une table de dispatch pour déterminer quelle fonction de calcul appliquer en fonction des types des entités. Les entités elles-mêmes se contentent de stocker leurs données et de fournir leur identifiant de type via la méthode `get_type()`. Ainsi, l'appel à `orchestrator.compute_synergy(e1, e2)` renvoie le score de synergie approprié pour la paire $(\mathcal{E}_1, \mathcal{E}_2)$.

D. Conclusion

L'approche centralisée proposée par le **Module Central** permet d'isoler la logique du calcul de la synergie dans un composant unique, facilitant ainsi la maintenance, l'extension et la compréhension du code. En centralisant l'algorithme de dispatch selon les types d'entités, il devient possible d'assurer que chaque combinaison (sub-sub, sym-sym, sub-sym) est traitée de manière cohérente et transparente. Ce modèle offre un haut degré de modularité, car les entités se limitent à stocker leurs données, tandis que la responsabilité du calcul de $S(i, j)$ est déléguée à un orchestrateur central. Cela simplifie grandement l'extension du SCN à de nouveaux types d'entités et permet de modifier ou d'améliorer les fonctions de synergie de manière localisée, sans avoir à remanier l'ensemble du système.

Le code Python présenté illustre concrètement ce concept et démontre l'efficacité d'une approche orientée objet pour la gestion des interactions hétérogènes dans un SCN.

Cette solution constitue un exemple clair de la manière dont l'**approche polymorphique** et la **centralisation** des fonctions de synergie s'intègrent dans une architecture logicielle solide, favorisant à la fois la **clarté**, l'**extensibilité** et la **maintenabilité** du système.

5.4.3. Gestion du Coût

Dans un SCN (Synergistic Connection Network), le calcul de la **synergie** $S(i, j)$ peut représenter un **gros poste** de dépense algorithmique, surtout lorsqu'on considère un nombre n potentiellement élevé d'entités \mathcal{E}_i . Il est donc impératif d'étudier en détail **comment** évaluer ces synergies, et à **quel rythme** on doit les recalculer. Cette section (5.4.3) s'intéresse à la **gestion du coût** d'un tel calcul, d'abord en constatant qu'un schéma naïf revient à un $O(n^2)$ (5.4.3.1), puis en exposant des **optimisations** (5.4.3.2) visant à réduire ou partiellement approximer ces évaluations.

5.4.3.1. Calcul $O(n^2)$ si on évalue $S(i, j)$ pour toutes les paires

Un moyen direct et souvent intuitif pour obtenir la fonction de synergie $S(i, j)$ dans un Synergistic Connection Network consiste à la calculer de manière exhaustive, c'est-à-dire à parcourir toutes les paires (i, j) avec $i, j \in \{1, \dots, n\}$. Cette approche repose sur la formule

$$S(i, j) = \mathcal{F}(\mathbf{data}(\mathcal{E}_i), \mathbf{data}(\mathcal{E}_j)),$$

où $\mathbf{data}(\mathcal{E}_i)$ représente la représentation interne de l'entité \mathcal{E}_i . Sur le plan algorithmique, on se limite alors à une double boucle :

$$\text{for } i = 1 \dots n \quad \text{for } j = 1 \dots n \quad S[i, j] = \text{calcSynergy}(\mathcal{E}_i, \mathcal{E}_j).$$

Si l'on exécute cette opération à chaque itération du SCN (par exemple pour mettre à jour la matrice $\omega(t+1)$ en fonction de la nouvelle synergie S), le coût total devient $O(T \times n^2)$ pour T itérations. Cette section examine dans le détail les conséquences de ce choix et les raisons qui poussent, en pratique, à le tempérer ou à l'optimiser.

A. Justification et intérêt mathématique

L'exhaustivité du calcul de $S(i, j)$ revêt une certaine élégance théorique : on couvre la totalité des paires d'entités, évitant ainsi de négliger d'éventuelles relations synergiques faibles mais susceptibles de s'amplifier au cours de la dynamique. Si la fonction S est simple à évaluer (par exemple, une similarité cosinus entre vecteurs en dimension modeste), alors le coût unitaire est proche de $O(d)$ et le total en $O(d n^2)$ peut demeurer acceptable tant que n n'est pas trop grand.

Il existe également une grande transparence dans cette méthode : on n'a besoin ni de structures complexes (indexation, arbres k-d, etc.), ni de stratégies pour exclure certaines paires (i, j) . On se contente de remplir ou de rafraîchir une matrice S de dimension $n \times n$. Sur le plan mathématique, la mise à jour $\omega_{i,j}(t+1) = \omega_{i,j}(t) + \dots$ peut ainsi s'appuyer en toute simplicité sur $S(i, j)$ pour n'importe quels i, j , sans hypothèses de sparsité ou de filtrage.

B. Limites en mémoire et en temps

Si n prend des valeurs importantes (par exemple $n \gtrsim 10^5$), la structure $S[i, j]$ devient de taille $O(n^2)$, ce qui peut dépasser les capacités mémoire d'une machine standard. Même pour un n de l'ordre de 10^4 , on atteint une matrice de 100 millions de valeurs, ce qui n'est pas impossible, mais déjà conséquent. À chaque itération du SCN, parcourir la totalité de ces entrées pour recalculer S et en dériver $\omega_{i,j}(t+1)$ s'approche d'un milliard d'opérations, qu'il faut en outre répéter T fois si l'on veut un SCN avec T itérations. Le coût asymptotique $O(T n^2)$ se heurte rapidement à des barrières pratiques lorsque n franchit la dizaine de milliers ou plus.

Au plan numérique, l'hypothèse $O(1)$ pour chaque calcul $S(i, j)$ peut se révéler trompeuse si la dimension des vecteurs est élevée, ou si S implique une logique plus complexe (ex. compatibilité symbolique examinant des structures de règles). On peut alors se retrouver avec $O(T \times n^2 \times \alpha)$, où α traduit le coût unitaire de la fonction S .

C. Dynamique du SCN et réactualisation de S

Une question se pose ensuite : doit-on recalculer $S(i, j)$ à chaque itération, ou est-il suffisant de le faire occasionnellement ? Si les entités \mathcal{E}_i sont stationnaires (leur contenu ne change pas), on peut fixer $S(i, j)$ une fois pour toutes au début et le réutiliser dans la suite des itérations. Le coût $O(n^2)$ se limite alors à l'initialisation. En revanche, dans un cadre plus évolutif (entités modifiées, flux de données), S peut fluctuer dans le temps, imposant une réévaluation fréquente, voire continue.

Cette réévaluation coûteuse se combine alors au schéma de mise à jour $\omega_{i,j}(t+1) = \dots$. Sur un plan purement mathématique, on sait décrire le SCN comme un système d'équations couplées, mais l'implémentation en $O(n^2)$ par itération peut se révéler irréaliste pour de larges échelles. Cela motive une réflexion sur des **optimisations** qui ne traitent pas toutes les paires (i, j) , ou qui recourent à des structures plus sélectives (chap. 5.4.3.2).

D. Architecture logicielle et partition

Si, malgré tout, on choisit le calcul exhaustif S en $O(n^2)$, il est intéressant de se demander comment l'**architecture** logicielle (chap. 5.2.3) peut l'aider à gagner en efficacité. Un partitionnement des entités en sous-blocs, accompagné d'un protocole de communication (synchrone ou asynchrone), peut répartir la charge de calcul S sur plusieurs nœuds. On ne résout cependant pas le facteur $O(n^2)$ total, on ne fait que le distribuer (et éventuellement le paralléliser).

Par ailleurs, la décision de stocker S de façon dense (matrice de taille $n \times n$) s'accompagne d'une problématique de **persistance** (5.3.1) : pour un grand n , la place disque ou mémoire devance souvent les capacités disponibles, à moins de disposer d'une infrastructure de calcul massif. Certains systèmes permettent d'exploiter des GPU ou clusters HPC, mais la multiplication des transferts de données peut introduire des goulots d'étranglement importants.

Conclusion

Le calcul $S(i, j)$ en **approche exhaustive** $O(n^2)$ est la méthode la plus directe pour évaluer la synergie entre toutes les paires, garantissant une couverture intégrale des liens potentiels et une grande **simplicité** au niveau de l'implémentation (double boucle ou matrice dense). Cette solution présente une **transparence** et une **complétude** mathématiques, convenant aux scénarios où n reste **modéré** et où la synergie n'a pas besoin d'être recalculée trop souvent. Cependant, elle devient **prohibitif** à plus grande échelle, tant en temps qu'en mémoire. La section suivante (5.4.3.2) introduit justement les **optimisations** usuelles pour échapper à l'impasse $O(n^2)$: on retiendra notamment l'idée de ne considérer que les plus proches voisins dans l'espace (k-NN, ϵ -radius, index spatial) afin de réduire fortement le nombre de paires (i, j) examinées et calculer S seulement là où elle est susceptible d'être non négligeable.

5.4.3.2. Optimisations : k-NN, ϵ -radius, indexation spatiale

Le calcul exhaustif $S(i, j)$ en $O(n^2)$ (présenté en 5.4.3.1) se révèle rapidement prohibitif pour de grands n . Afin d'y remédier, il est courant de **cibler** le calcul sur un sous-ensemble de paires (i, j) jugées plus pertinentes, ou d'accélérer la recherche de voisinages. Plusieurs stratégies interviennent alors : exploiter la notion de "**k plus proches voisins**" (k-NN), utiliser un " **ϵ -radius**" pour ignorer les entités trop distantes, ou recourir à des **structures d'indexation spatiale** (k-d trees, vantage-point trees, ANN). Ces solutions permettent, d'un point de vue mathématique et algorithmique, de **réduire** le nombre de calculs $S(i, j)$ ou d'en **accélérer** l'identification, et donc de sortir du carcan $O(n^2)$.

A. k-NN (k plus proches voisins)

Les techniques de "**k plus proches voisins**" consistent à limiter la recherche de synergie $S(i, j)$ aux seuls j qui se trouvent dans le **top-k** de i en termes de proximité ou de similarité. Si la représentation \mathbf{x}_i est un vecteur dans \mathbb{R}^d , alors la "**distance**" (euclidienne ou autre) sert de guide : au lieu de comparer \mathbf{x}_i à toutes les \mathbf{x}_j , on se borne aux k entités les plus similaires. En pratique, on détermine

$$\text{kNN}(i) = \{j \mid \mathbf{x}_j \text{ est parmi les } k \text{ plus proches de } \mathbf{x}_i\}.$$

Le calcul effectif de la k-NN dans un espace vectoriel naïf reste $O(n)$ par requête, menant à $O(n^2)$ global pour tous les i . Cependant, on combine généralement cette idée avec des **structures d'index** plus avancées (cf. infra) ou des algorithmes d'**approximate nearest neighbors** (FAISS, Annoy, etc.), qui réduisent la complexité à $O(n \log n)$ (ou un peu plus) en moyenne. On obtient alors un stockage final $O(kn)$ pour la synergie, au lieu de $O(n^2)$.

D'un point de vue mathématique, cette approche revient à nier la pertinence de la synergie au-delà de ces k voisins, hypothèse justifiée par une décroissance rapide de S ou une complémentarité faible en dehors du voisinage local. Dans le contexte d'un SCN (Synergistic Connection Network), cela recoupe la notion de "voisinage restreint" ou "parsimonie", évitant de consacrer des ressources à des comparaisons lointaines et souvent négligeables.

B. ϵ -radius : filtrer par distance

Une seconde famille de méthodes, souvent apparentée, repose sur la définition d'un **rayon** ϵ . On ne s'intéresse qu'aux entités \mathbf{x}_j qui se trouvent à une distance inférieure à ϵ de \mathbf{x}_i (selon un certain metric ou kernel). Cela fait sens si la synergie se base sur un *décroissement* rapide avec la distance :

$$S(i, j) = f(\|\mathbf{x}_i - \mathbf{x}_j\|) \quad \text{avec } f(r) \approx 0 \text{ si } r > \epsilon.$$

Ainsi, pour chaque \mathbf{x}_i , on exécute une **recherche** dans un rayon ϵ . Là encore, en mode naïf, on testerait toutes les \mathbf{x}_j ($O(n)$ par entité, menant à $O(n^2)$ global). On recourt donc à des **structures** spécialisées (k-d tree, vantage-point tree, etc.) ou à des algorithmes d'**indexation** accélérant la "range search". Le résultat est une localité : si \mathbf{x}_j est trop éloigné, on ne calcule même pas S . Ce rayon ϵ est adapté selon le **domaine** et la **géométrie** (en robotique, en vision...). Sur un plan mathématique, on introduit un "**seuil**" d'influence spatiale. Dans un SCN, cela se traduit par :

$$\omega_{i,j}(t+1) = 0 \quad \text{si } \|\mathbf{x}_i - \mathbf{x}_j\| > \epsilon,$$

et donc aucun calcul explicite de S pour ces paires.

C. Indexation spatiale

Afin de rendre efficaces les recherches "k plus proches voisins" ou " ϵ -radius", on utilise des **structures d'indexation** dans l'espace :

- **k-d tree**, vantage-point tree, ball tree, R*-tree, etc. Dans un espace \mathbb{R}^d de dimension modeste ($\leq 20-30$), ces arbres réduisent le parcours moyen pour un range search ou un k-NN search, passant d'un $O(n)$ naïf à $O(n^\alpha)$, $\alpha < 1$.
- **Approximate nearest neighbors (ANN)** si la dimension est plus grande, recourant à des solutions comme Faiss (Facebook AI Similarity Search), Annoy (Spotify), HNSW, etc. Elles ne donnent pas toujours *exactement* le top-k ou l'exact ϵ -voisinage, mais en pratique leur "faible erreur" suffit dans un SCN où l'on ne cherche pas la perfection.

Ces techniques, d'un point de vue mathématique, constituent une pré-organisation de l'espace \mathbf{x}_i , où l'on sait rapidement restreindre la recherche dans une zone d'intérêt ou un "cluster potentiel". Sur le plan ingénierie, on construit l'index (qui peut se mettre à jour si les entités changent), puis on exécute des requêtes k-NN ou range à chaque besoin de calcul S . Cela ramène l'effort global parfois à $O(n \log n)$ ou $O(n^{1+\epsilon})$, bien plus viable que $O(n^2)$.

D. Bénéfices et limites

Ces trois idées (k-NN, ϵ -radius, indexation spatiale) sont souvent **combinées** :

- Sur le **plan mathématique**, on accepte une **localité** : la synergie S s'avère non négligeable que dans un "voisinage" restreint, qu'il soit déterminé par un rayon ϵ ou par une "frontière de k plus proches".

- Cela diminue drastiquement le **nombre** de paires (i, j) à évaluer, passant d'un potentiel $O(n^2)$ à un $O(nk)$ ou $O(n \log n)$ si l'indexation fonctionne bien.
- La limite est que cette hypothèse *peut* omettre certains liens si la synergie n'est pas strictement localisée en distance. Toutefois, pour de nombreux cadres sub-symboliques (embeddings, distances décroissantes), c'est parfaitement cohérent.

Sur un plan pratique, ces méthodes requièrent la **mise à jour** ou la **reconstruction** périodique de l'index (si les entités évoluent ou si de nouvelles arrivent), ce qui ajoute un surcoût, mais généralement bien moindre que le balayage complet $O(n^2)$.

L'efficacité dépend aussi de la **dimension** : un k-d tree peut perdre en performance si d est trop élevé. Dans ce cas, les algorithmes d'**approximate** nearest neighbors prennent le relais, tolérant une petite erreur mais préservant une grande rapidité pour trouver les "meilleurs" voisins.

Conclusion

Les **optimisations** k-NN, ϵ -radius, et **indexation spatiale** forment un **socle** de stratégies pour éviter ou amortir le coût $O(n^2)$ dans le calcul de la synergie $S(i, j)$. Elles consistent à **focaliser** la recherche sur un **voisinage local** jugé le plus pertinent, que ce soit en limitant le nombre (k) de voisins ou en imposant un rayon ϵ . D'un point de vue mathématique, on suppose une **localisation** de S autour de points proches ; sur le plan ingénierie, on s'appuie sur des **structures** (k-d tree, ANN) pour accélérer ce ciblage. L'effet global est de passer d'un régime $O(n^2)$ parfois inabordable, à un schéma plus abordable $O(n \log n)$ ou $O(kn)$, rendant possible le calcul fréquent ou continu de S dans un SCN à plus grande échelle.

5.5. Module de Mise à Jour ω et Inhibition/Contrôle

5.5.1. Rappels des Règles DSL

- 5.5.1.1. Equation additive $\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)]$.
- 5.5.1.2. Variantes (multiplicative, etc.).

5.5.2. Inhibition et Compétition

- 5.5.2.1. Formule $-\gamma \sum_{k \neq j} \omega_{i,k}$ dans la mise à jour.
- 5.5.2.2. Paramétrage γ .
- 5.5.2.3. Application concrète : limiter la somme des liens sortants par entité.

5.5.3. Saturation

- 5.5.3.1. Clipping $\omega_{i,j} \leftarrow \min(\omega_{i,j}, \omega_{\max})$.
- 5.5.3.2. Réglage de ω_{\max} et son effet sur l'émergence de clusters.

5.5.4. Recuit Simulé ou Bruit

- 5.5.4.1. Ajouter un terme stochastique $\xi_{i,j}$ dans la mise à jour.
- 5.5.4.2. Diminution progressive de la “température” pour éviter de rester piégé dans un minimum local.

Dans l'architecture globale du **SCN**, on retrouve un **Module** spécifiquement dédié à la **mise à jour** des pondérations $\omega_{i,j}$. Ce module occupe une place centrale : il assure la transition $\omega(t) \rightarrow \omega(t+1)$ à chaque itération, en se basant sur la **synergie** $S(i,j)$ (chap. 5.4) et d'éventuels **mécanismes** de contrôle (inhibition, saturation, recuit stochastique, etc.). En ce sens, il concrétise la boucle fondamentale du Deep Synergy Learning (DSL) décrite en chapitre 4.

Dans cette section (5.5), nous commençons par rappeler les **règles DSL** les plus courantes pour la mise à jour ω (5.5.1), avant de détailler la manière dont on peut y introduire une **inhibition compétitive** (5.5.2), une **saturation** (5.5.3) et, si besoin, un **recuit simulé** ou un **bruit stochastique** (5.5.4). Sur le plan mathématique, chaque composante (inhibition, saturation, bruit) peut être vue comme un opérateur supplémentaire venant se superposer à la formule de base, modulant la convergence et la formation des clusters.

5.5.1. Rappels des Règles DSL

Les travaux exposés en chapitres 2 et 4 introduisent déjà la dynamique d'un SCN : on y voit comment $\omega_{i,j}$ se met à jour en fonction de $S(i,j)$, d'un taux d'apprentissage η et d'un paramètre de décroissance τ . Ici, nous nous concentrons sur l'**implémentation** et l'**intégration** de cette mise à jour dans le **Module** correspondant, tout en rappelant les éléments mathématiques essentiels.

Voici une version développée, structurée et enrichie de la section **5.5.1.1. Équation additive** $\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)]$, suivie d'une implémentation en Python qui traduit concrètement ce schéma de mise à jour.

5.5.1.1. Équation additive $\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)]$

Les **misés à jour additives** représentent l'une des formulations les plus intuitives et classiques dans l'analyse d'un **Synergistic Connection Network (SCN)**. Le principe fondamental est de faire évoluer chaque **pondération** $\omega_{i,j}$ en fonction d'un terme correctif qui combine deux contributions opposées : d'une part, l'**attraction** induite par la **synergie** $S(i,j)$ qui tend à augmenter $\omega_{i,j}$ et, d'autre part, une **décroissance linéaire** proportionnelle à $\omega_{i,j}$ elle-même, contrôlée par le paramètre τ . Cette dualité assure que la mise à jour guide $\omega_{i,j}$ vers un **équilibre** ou **point fixe** donné par

$$\omega_{i,j}^* = \frac{S(i,j)}{\tau}.$$

A. Forme canonique et point fixe

La **formule** additive est donnée par

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)].$$

Ici, le **taux d'apprentissage** $\eta > 0$ détermine l'amplitude des corrections à chaque itération, tandis que $\tau > 0$ module la **force de décroissance** appliquée à $\omega_{i,j}(t)$. Au **point fixe** de cette dynamique, lorsque $\omega_{i,j}(t+1) = \omega_{i,j}(t) = \omega_{i,j}^*$, l'équation se simplifie en

$$\omega_{i,j}^* = \omega_{i,j}^* + \eta [S(i,j) - \tau \omega_{i,j}^*] \Rightarrow S(i,j) = \tau \omega_{i,j}^*,$$

ce qui implique immédiatement

$$\omega_{i,j}^* = \frac{S(i,j)}{\tau}.$$

Cette solution montre que, dans des conditions idéales et en l'absence d'effets additionnels (tels que l'inhibition ou le bruit), la **pondération** tend vers un équilibre directement proportionnel à la **synergie** et inversement proportionnel à τ .

B. Interprétation mathématique et convergence

En réécrivant l'équation de mise à jour, on peut mettre en évidence un schéma linéaire :

$$\omega_{i,j}(t+1) = (1 - \eta \tau) \omega_{i,j}(t) + \eta S(i,j).$$

Ce schéma se présente comme une combinaison linéaire entre l'**état** antérieur et un **terme forçant** la valeur vers $\eta S(i,j)$. Pour que le système converge de manière stable vers $\omega_{i,j}^* = \frac{S(i,j)}{\tau}$, il est impératif que la contraction imposée par le facteur $(1 - \eta \tau)$ soit suffisante, ce qui se traduit par la condition

$$\eta \tau < 2.$$

Sous cette hypothèse, l'écart $\left| \omega_{i,j}(t) - \frac{S(i,j)}{\tau} \right|$ décroît de façon exponentielle au fil des itérations, garantissant une **convergence stable**.

De plus, on peut associer à cette mise à jour une vision en termes de **pseudo-énergie**. En définissant une fonction potentielle

$$\mathcal{J}(\omega) = - \sum_{i,j} \omega_{i,j} S(i,j) + \frac{\tau}{2} \sum_{i,j} \omega_{i,j}^2,$$

la condition de premier ordre pour minimiser \mathcal{J} (c'est-à-dire $\frac{\partial \mathcal{J}}{\partial \omega_{i,j}} = 0$) conduit exactement à

$$-S(i,j) + \tau \omega_{i,j} = 0 \Rightarrow \omega_{i,j} = \frac{S(i,j)}{\tau}.$$

Ainsi, le système peut être interprété comme effectuant une **descente de gradient** sur \mathcal{J} , orientant les mises à jour vers un minimum de cette fonction potentielle.

C. Implémentation dans un SCN

Du point de vue **algorithmique**, l'implémentation de la règle additive est souvent réalisée via une boucle itérative qui applique, pour chaque paire (i, j) , la formule de mise à jour. Une approche recommandée consiste à utiliser un **double-buffer** afin d'éviter la modification de $\omega(t)$ en temps réel lors du calcul des mises à jour, garantissant ainsi la cohérence des valeurs lues pour l'ensemble des paires. L'architecture logicielle typique comporte un module de **mise à jour** chargé de parcourir la matrice de pondérations, de récupérer les valeurs de la **synergie** $S(i, j)$ à partir d'un module dédié, et d'appliquer la formule suivante :

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta[S(i, j) - \tau \omega_{i,j}(t)].$$

Ce module peut également intégrer des mécanismes complémentaires tels que l'**inhibition** (par exemple, un terme supplémentaire $-\gamma \sum_{k \neq j} \omega_{i,k}(t)$) ou des techniques de **clipping** pour maintenir les valeurs dans un intervalle souhaité.

D. Exemple de Comportement et Implémentation en Python

Pour illustrer concrètement la mise à jour additive, considérons un exemple simple en Python. Supposons que la synergie $S(i, j)$ soit donnée et stationnaire, et que l'on souhaite observer la convergence de la pondération $\omega_{i,j}(t)$ vers la valeur $\frac{S(i,j)}{\tau}$.

Voici une implémentation en Python :

```
import numpy as np
import matplotlib.pyplot as plt

# Paramètres de la mise à jour
eta = 0.05      # Taux d'apprentissage
tau = 1.0       # Facteur de décroissance
num_iterations = 50 # Nombre d'itérations

# Supposons que S(i,j) est donné et constant
# Pour cet exemple, nous considérons une paire d'entités pour laquelle S(i,j) = 0.8
S_ij = 0.8

# Initialisation de la pondération omega avec un bruit faible (près de 0)
omega = 0.0 + np.random.uniform(-0.01, 0.01)

# Liste pour stocker l'évolution de omega
omega_history = [omega]

# Boucle de mise à jour selon la règle additive
for t in range(num_iterations):
    # Calcul du delta selon la règle additive
    delta = eta * (S_ij - tau * omega)
    # Mise à jour de omega
    omega = omega + delta
    # Enregistrement de l'état courant
    omega_history.append(omega)

# Calcul de la valeur théorique de convergence (point fixe)
omega_fixed = S_ij / tau

# Affichage graphique de l'évolution de omega
plt.figure(figsize=(8, 5))
```

```
plt.plot(omega_history, marker='o', linestyle='-', color='b', label=r'$\omega_{i,j}(t)$')
plt.axhline(y=omega_fixed, color='r', linestyle='--', label=r'$S(i,j)/\tau$')
plt.title("Convergence de la pondération $\omega_{i,j}(t)$ selon la règle additive")
plt.xlabel("Itérations $t$")
plt.ylabel(r"$\omega_{i,j}(t)$")
plt.legend()
plt.grid(True)
plt.show()
```

Dans cet exemple, nous initialisons la pondération $\omega_{i,j}(0)$ avec une valeur proche de 0 et nous appliquons la mise à jour additive sur 50 itérations. Le graphique montre que $\omega_{i,j}(t)$ converge progressivement vers la valeur théorique $\omega_{i,j}^* = \frac{S(i,j)}{\tau} = 0.8$. Le schéma de mise à jour est exprimé mathématiquement par :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + 0.05 \left(0.8 - 1.0 \omega_{i,j}(t) \right).$$

La condition de stabilité, $\eta \tau < 2$, est vérifiée puisque $0.05 \times 1.0 = 0.05 < 2$, assurant ainsi une **convergence stable** sans oscillations.

E. Conclusion

La règle additive

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)]$$

représente une approche simple et efficace pour la mise à jour des pondérations dans un SCN. Elle conduit chaque $\omega_{i,j}$ à converger vers le point fixe $\frac{S(i,j)}{\tau}$ lorsque la synergie est stationnaire et que les paramètres η et τ sont choisis de manière appropriée. L'implémentation en Python illustre comment, à l'aide d'un **double-buffer** conceptuel et d'une boucle itérative, la dynamique mathématique se traduit en une routine algorithmique concrète. Ainsi, cette approche constitue le cœur du mécanisme d'auto-organisation dans un SCN, permettant de fusionner la théorie (descente de gradient et pseudo-énergie) avec la pratique (mise à jour itérative des pondérations) dans une infrastructure logicielle robuste.

Ce développement fournit à la fois une compréhension approfondie de la mise à jour additive dans le cadre du DSL et un exemple d'implémentation en Python qui permet de visualiser la convergence des pondérations vers leur équilibre théorique.

Voici une présentation détaillée de la section **5.5.1.2. Variantes (multiplicative, etc.)** rédigée dans un style académique, intégrant de nombreuses formules mathématiques et une implémentation complète en Python avec visualisation graphique.

5.5.1.2. Variantes (multiplicative, etc.)

Dans le cadre du **Synergistic Connection Network (SCN)**, la règle additive présentée en section 5.5.1.1 constitue l'approche la plus simple pour mettre à jour la matrice des pondérations $\omega_{i,j}$. Cependant, d'autres formulations permettent de modifier la nature de la correction appliquée à ces pondérations. L'une des alternatives notoires est l'**approche multiplicative**, qui, au lieu d'ajouter un delta fixe à $\omega_{i,j}$, applique un facteur multiplicatif qui dépend de la valeur courante de $\omega_{i,j}$. Ce type de mise à jour est particulièrement pertinent dans les systèmes où l'on souhaite que les liens déjà forts se renforcent plus rapidement, tout en diminuant proportionnellement les liens faibles.

A. Formulation Multiplicative

La règle multiplicative de base se définit par l'équation :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) [1 + \eta (S(i,j) - \tau \omega_{i,j}(t))].$$

Dans cette formulation, le terme $\eta (S(i,j) - \tau \omega_{i,j}(t))$ représente la correction relative appliquée à $\omega_{i,j}(t)$. Si ce terme est positif, le facteur multiplicatif dépasse 1 et le lien se renforce de façon proportionnelle à sa valeur actuelle, favorisant ainsi une dynamique « effet boule de neige ». À l'inverse, lorsque le terme est négatif, le facteur est inférieur à 1 et $\omega_{i,j}(t)$ diminue.

L'intérêt de cette approche réside dans sa capacité à **amplifier** les liens déjà établis. En effet, un lien dont $\omega_{i,j}(t)$ est élevé bénéficiera d'une multiplication plus forte (à condition que $S(i,j) - \tau \omega_{i,j}(t)$ reste positif), tandis qu'un lien faible aura du mal à sortir de l'état marginal, renforçant ainsi la polarisation des connexions au sein du SCN.

B. Comparaison avec l'Approche Additive

Dans la mise à jour **additive**, la modification est donnée par un incrément absolu :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

ce qui signifie que la correction appliquée ne dépend pas de la valeur courante de $\omega_{i,j}(t)$. Ce schéma induit un mouvement linéaire vers le point fixe $\omega_{i,j}^* = \frac{S(i,j)}{\tau}$.

Dans le cas multiplicatif, la correction est proportionnelle à $\omega_{i,j}(t)$:

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) [1 + \eta \Delta(t)] \quad \text{avec} \quad \Delta(t) = S(i,j) - \tau \omega_{i,j}(t).$$

Ainsi, si $\omega_{i,j}(t)$ est déjà élevé, la mise à jour entraîne une augmentation plus importante, tandis qu'un lien faible sera mis à jour de manière moins significative. Cette **dynamique exponentielle** peut favoriser la formation de clusters très marqués, mais elle nécessite également une gestion rigoureuse des paramètres pour éviter une divergence (risque d'explosion) ou des oscillations.

C. Autres Variantes et Approches Hybrides

Outre l'approche purement multiplicative, il est envisageable de combiner les aspects additifs et multiplicatifs pour obtenir un schéma **semi-additif** ou **hybride**. Par exemple, une formulation hybride peut être exprimée par :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \alpha \omega_{i,j}(t) [S(i,j) - \tau \omega_{i,j}(t)] + \beta [S(i,j) - \tau \omega_{i,j}(t)],$$

où α et β sont des coefficients permettant d'ajuster l'influence relative des termes multiplicatif et additif. Un tel schéma offre une flexibilité accrue, permettant de moduler finement la dynamique des mises à jour en fonction des besoins spécifiques du système.

D. Implémentation Python avec Visualisation

Pour illustrer la règle multiplicative, nous proposons ci-après une implémentation en Python qui simule l'évolution de la pondération $\omega_{i,j}$ pour une paire d'entités, puis affiche la courbe de convergence à l'aide de **matplotlib**.

```
import numpy as np
import matplotlib.pyplot as plt

# Paramètres de mise à jour
eta = 0.05 # Taux d'apprentissage
tau = 1.0 # Facteur de décroissance
num_iterations = 50 # Nombre d'itérations
S_ij = 0.8 # Synergie S(i,j) pour la paire considérée
```

```

# Initialisation de la pondération omega avec un bruit faible autour de 0
omega = 0.0 + np.random.uniform(-0.01, 0.01)

# Liste pour stocker l'évolution de omega
omega_history = [omega]

# Boucle de mise à jour multiplicative
for t in range(num_iterations):
    # Calcul du delta multiplicatif
    delta = eta * (S_ij - tau * omega)
    # Mise à jour multiplicative : on multiplie la valeur courante par un facteur
    omega = omega * (1 + delta)
    # Optionnel : on peut appliquer un clipping pour éviter les valeurs négatives
    if omega < 0:
        omega = 0
    omega_history.append(omega)

# Calcul de la valeur théorique de convergence (point fixe) pour comparaison
omega_fixed = S_ij / tau

# Affichage graphique de l'évolution de omega
plt.figure(figsize=(8, 5))
plt.plot(omega_history, marker='o', linestyle='-', color='blue', label=r'$\omega_{i,j}(t)$')
plt.axhline(y=omega_fixed, color='red', linestyle='--', label=r'$S(i,j)/\tau$')
plt.title("Convergence de la pondération $\omega_{i,j}(t)$ avec la règle multiplicative")
plt.xlabel("Itérations $t$")
plt.ylabel(r"$\omega_{i,j}(t)$")
plt.legend()
plt.grid(True)
plt.show()

```

Dans cet exemple, nous initialisons la pondération $\omega_{i,j}(0)$ avec une valeur proche de 0 et appliquons la mise à jour multiplicative pour 50 itérations. La formule utilisée est :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) \left[1 + \eta \left(S(i,j) - \tau \omega_{i,j}(t) \right) \right],$$

ce qui signifie que la valeur de $\omega_{i,j}(t)$ est multipliée par le facteur $\left[1 + \eta \left(S(i,j) - \tau \omega_{i,j}(t) \right) \right]$. La figure générée montre la convergence de $\omega_{i,j}(t)$ vers la valeur théorique $\frac{S(i,j)}{\tau} = 0.8$ de manière exponentielle, tout en illustrant le comportement non linéaire inhérent à la formulation multiplicative.

E. Conclusion

La variante multiplicative

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) \left[1 + \eta \left(S(i,j) - \tau \omega_{i,j}(t) \right) \right]$$

offre une alternative intéressante à la règle additive en amplifiant proportionnellement la correction selon la valeur courante de $\omega_{i,j}(t)$. Cette approche peut accélérer la formation de liens forts et favoriser des dynamiques de polarisation, tout en nécessitant un contrôle rigoureux des paramètres pour éviter une croissance explosive. L'implémentation en Python présentée ci-dessus illustre la convergence vers le point fixe et offre un moyen visuel d'appréhender le comportement de la mise à jour multiplicative dans un SCN. Ce type de schéma, éventuellement combiné avec des approches hybrides ou semi-additives, constitue une composante essentielle dans l'ensemble des stratégies d'auto-organisation que l'on retrouve dans les SCN, et sert de base pour des développements ultérieurs dans le cadre de systèmes plus complexes et adaptatifs.

5.5.2.1. Formule $-\gamma \sum_{k \neq j} \omega_{i,k}$ dans la mise à jour

Dans un **SCN**, il est souvent souhaitable d'instaurer une **compétition** entre les liens issus d'une même entité. En d'autres termes, lorsqu'une entité \mathcal{E}_i renforce la connexion avec l'une de ses cibles \mathcal{E}_j via la pondération $\omega_{i,j}$, il est pertinent de freiner simultanément la croissance des autres connexions $\omega_{i,k}$ pour $k \neq j$. Cette idée, qui s'inspire notamment des mécanismes d'**inhibition latérale** observés dans les réseaux neuronaux biologiques, se formalise mathématiquement par l'ajout d'un terme négatif dans la règle de mise à jour.

A. Idée Générale de l'Inhibition Compétitive

L'objectif est de limiter la capacité d'une entité à « investir » simultanément dans de nombreux liens forts. On introduit ainsi un **budget** de pondération pour chaque entité \mathcal{E}_i . En d'autres termes, si \mathcal{E}_i a déjà alloué une part importante de son « capital » de connexion à certains partenaires, il lui reste moins de ressources pour renforcer d'autres liens. Ce mécanisme permet d'obtenir une **spécialisation** des connexions et d'éviter une croissance simultanée excessive qui pourrait conduire à un réseau peu discriminant.

B. Sens et Rôle du Terme d'Inhibition

Le terme $-\gamma \sum_{k \neq j} \omega_{i,k}(t)$ introduit une pression négative sur la mise à jour du lien $\omega_{i,j}(t)$. Plus précisément, si l'ensemble des connexions issues de l'entité \mathcal{E}_i est déjà élevé, la somme $\sum_{k \neq j} \omega_{i,k}(t)$ sera grande et la correction négative sera plus importante. Cela se traduit par une limitation de l'augmentation de $\omega_{i,j}(t)$, ce qui force l'entité à concentrer ses ressources sur un nombre restreint de liens – idéalement ceux présentant une synergie forte. En d'autres termes, ce mécanisme de **compétition latérale** veille à ce que l'augmentation d'un lien ne se fasse qu'au détriment des autres, créant ainsi un environnement où seuls quelques liens majeurs émergent.

C. Modélisation Mathématique

Considérons la règle de mise à jour additive classique pour la pondération d'un lien, qui s'exprime sans inhibition par :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)].$$

Pour intégrer la notion de compétition entre les connexions issues d'une même entité, on ajoute un terme d'inhibition :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] - \gamma \sum_{k \neq j} \omega_{i,k}(t).$$

Ici, η est le taux d'apprentissage qui détermine l'amplitude de la correction, τ régule la décroissance linéaire de $\omega_{i,j}$, et γ contrôle l'intensité de l'inhibition compétitive. Cette équation montre que, quelle que soit la synergie $S(i,j)$, le renforcement du lien $\omega_{i,j}$ sera toujours freiné par la somme des autres liens issus de \mathcal{E}_i .

D. Effets sur la Dynamique du Réseau

L'introduction du terme d'inhibition a plusieurs conséquences notables sur la dynamique du SCN :

- **Spécialisation des Connexions** : Seuls les liens pour lesquels la synergie $S(i,j)$ est suffisamment élevée pourront dépasser l'effet inhibiteur, tandis que les autres resteront faibles. Cela favorise l'émergence de clusters où chaque entité concentre ses connexions sur un nombre limité de partenaires.
- **Prévention de l'Emballlement Global** : Même si plusieurs paires présentent une synergie élevée, l'inhibition globale empêche que toutes les connexions soient simultanément fortes, ce qui pourrait diluer l'information sur la structure réelle du réseau.

- **Risque d'Oscillations** : Si le paramètre γ est trop élevé, le frein imposé peut entraîner des oscillations dans la dynamique des mises à jour, ce qui nécessitera alors des ajustements ou l'introduction de mécanismes de saturation pour stabiliser le comportement.

E. Implémentation Python avec Visualisation

Pour illustrer la dynamique induite par la mise à jour avec inhibition compétitive, nous proposons ci-après une implémentation en Python. Le script simule l'évolution de la pondération $\omega_{i,j}(t)$ pour une entité \mathcal{E}_i connectée à plusieurs cibles \mathcal{E}_j et affiche l'évolution de ces poids au cours des itérations.

```
import numpy as np
import matplotlib.pyplot as plt

# Paramètres de la mise à jour
eta = 0.05    # Taux d'apprentissage
tau = 1.0     # Facteur de décroissance
gamma = 0.02  # Coefficient d'inhibition
num_iterations = 100 # Nombre d'itérations
n_links = 5   # Nombre de liens sortants pour l'entité E_i

# Pour cet exemple, fixons une synergie S(i,j) pour chaque lien j
# On peut imaginer que l'entité a une forte synergie avec certains liens et faible avec d'autres
# Par exemple : S = [0.8, 0.7, 0.3, 0.2, 0.1]
S = np.array([0.8, 0.7, 0.3, 0.2, 0.1])

# Initialisation de la matrice des poids pour l'entité E_i (une seule ligne)
# On initialise chaque poids à une petite valeur aléatoire proche de 0
np.random.seed(42)
omega = np.random.uniform(0, 0.05, size=n_links)

# Stockage de l'évolution des poids pour visualisation
omega_history = np.zeros((num_iterations + 1, n_links))
omega_history[0, :] = omega.copy()

# Boucle de mise à jour
for t in range(num_iterations):
    # Calcul de la somme des poids pour l'entité E_i
    total_weight = np.sum(omega)

    # Mise à jour pour chaque lien j
    for j in range(n_links):
        # Calcul du terme d'update de base (sans inhibition)
        delta_update = eta * (S[j] - tau * omega[j])
        # Calcul du terme d'inhibition : somme des poids des autres liens (k != j)
        inhibition = gamma * (total_weight - omega[j])
        # Mise à jour additive avec inhibition
        omega[j] = omega[j] + delta_update - inhibition
        # Assurer que le poids reste positif (clipping à 0)
        if omega[j] < 0:
            omega[j] = 0
    # Enregistrer les poids de l'itération actuelle
    omega_history[t + 1, :] = omega.copy()

# Calcul théorique du point fixe pour chaque lien sans inhibition (pour comparaison)
omega_fixed = S / tau
```

```

# Affichage graphique de l'évolution des poids
plt.figure(figsize=(10, 6))
for j in range(n_links):
    plt.plot(omega_history[:, j], marker='o', label=f'$\\omega_{\\{i,\\{j+1\\}\\}}(t)$')
plt.axhline(y=omega_fixed[0], color='C0', linestyle='--', label=r'$S(1)^{\\tau}$')
plt.axhline(y=omega_fixed[1], color='C1', linestyle='--', label=r'$S(2)^{\\tau}$')
plt.axhline(y=omega_fixed[2], color='C2', linestyle='--', label=r'$S(3)^{\\tau}$')
plt.axhline(y=omega_fixed[3], color='C3', linestyle='--', label=r'$S(4)^{\\tau}$')
plt.axhline(y=omega_fixed[4], color='C4', linestyle='--', label=r'$S(5)^{\\tau}$')
plt.title("Évolution des pondérations $\\omega_{i,j}(t)$ avec inhibition compétitive")
plt.xlabel("Itérations $t$")
plt.ylabel("$\\omega_{i,j}(t)$")
plt.legend()
plt.grid(True)
plt.show()

```

F. Explications de l'Implémentation

Dans ce script Python :

- **Initialisation**

On fixe les paramètres $\eta = 0.05$, $\tau = 1.0$ et $\gamma = 0.02$. Le nombre de liens (représentant les connexions sortantes d'une entité \mathcal{E}_i) est fixé à 5, avec des valeurs de synergie $S(i, j)$ définies par un vecteur $S = [0.8, 0.7, 0.3, 0.2, 0.1]$. Les poids ω sont initialisés à des valeurs aléatoires proches de 0.

- **Mise à jour**

La mise à jour pour chaque poids suit la formule :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta \left(S(j) - \tau \omega_{i,j}(t) \right) - \gamma \left(\sum_{k=1}^{n_{\text{links}}} \omega_{i,k}(t) - \omega_{i,j}(t) \right).$$

Ici, la somme des poids de toutes les connexions (sauf le lien considéré) est utilisée pour appliquer l'inhibition compétitive, qui réduit la mise à jour si d'autres liens sont déjà forts. Un mécanisme de **clipping** garantit que les poids ne deviennent pas négatifs.

- **Visualisation**

L'évolution des poids est enregistrée dans la matrice *omega_history*, et à la fin, un graphique montre la trajectoire de chaque $\omega_{i,j}(t)$ au fil des itérations. Les lignes horizontales en pointillés indiquent les valeurs théoriques $\frac{S(i,j)}{\tau}$ (sans inhibition) pour référence.

G. Conclusion

Le terme $-\gamma \sum_{k \neq j} \omega_{i,k}(t)$ introduit une **inhibition compétitive** qui force l'entité à concentrer ses ressources sur un nombre limité de liens forts. Ce mécanisme prévient une croissance excessive et favorise la spécialisation des connexions, contribuant ainsi à la formation de clusters plus définis dans le SCN. La mise à jour globale, qui combine un terme d'attraction (provenant de $S(i, j)$) et un terme d'inhibition, se traduit par la formule :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta \left[S(i, j) - \tau \omega_{i,j}(t) \right] - \gamma \sum_{k \neq j} \omega_{i,k}(t).$$

L'implémentation Python présentée ci-dessus permet de simuler cette dynamique et d'en observer la convergence ou l'émergence de comportements particuliers, tels que la concentration des liens pour certaines connexions au détriment d'autres, phénomène essentiel dans la formation de clusters dans un SCN.

Ce développement, combinant explications mathématiques et implémentation pratique, illustre la manière dont un terme d'inhibition peut être intégré dans la mise à jour des pondérations pour favoriser une **auto-organisation** efficace dans un Deep Synergy Learning.

5.5.2.2. Paramétrage γ

Dans un **Synergistic Connection Network** (SCN), le coefficient γ apparaît dans le terme d'inhibition compétitive et joue un rôle crucial dans la manière dont une entité \mathcal{E}_i répartit son « potentiel de connexion » entre ses divers liens $\omega_{i,k}$. Pour comprendre ce paramétrage, rappelons d'abord la règle de mise à jour additive de base, à laquelle on ajoute ensuite un terme d'inhibition.

A. Rôle conceptuel et formulation mathématique

Sans inhibition, la mise à jour des pondérations dans un SCN s'exprime par la formule additive classique :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

où :

- $\eta > 0$ est le **taux d'apprentissage**,
- $S(i,j)$ représente la **synergie** (ou similarité) entre les entités \mathcal{E}_i et \mathcal{E}_j ,
- $\tau > 0$ est le paramètre régulant la décroissance linéaire.

Afin d'incorporer une **compétition** entre les différents liens issus de la même entité \mathcal{E}_i , on ajoute un terme d'inhibition qui est proportionnel à la somme des pondérations des autres connexions :

$$-\gamma \sum_{k \neq j} \omega_{i,k}(t),$$

ce qui conduit à la formule complète :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] - \gamma \sum_{k \neq j} \omega_{i,k}(t).$$

Ici, le coefficient γ contrôle l'intensité de cette inhibition compétitive. Un γ élevé signifie que l'entité \mathcal{E}_i est fortement contrainte à limiter simultanément plusieurs connexions fortes. Ainsi, si un lien $\omega_{i,j}$ tend à se renforcer en raison d'une synergie élevée, la somme des autres liens $\sum_{k \neq j} \omega_{i,k}(t)$ va exercer une pression négative qui réduira l'incrément pour $\omega_{i,j}$.

B. Impact sur la dynamique du réseau

Le paramétrage de γ détermine de manière critique la façon dont les ressources de connexion d'une entité se répartissent :

- **Sélectivité accrue** : Un γ suffisamment grand impose une forte compétition, de sorte que l'entité ne pourra renforcer que quelques liens dominants, tandis que les autres resteront faibles. Ce mécanisme favorise la formation de clusters bien définis, dans lesquels chaque entité investit principalement dans un sous-ensemble de connexions.
- **Prévention de la croissance excessive** : Même lorsque plusieurs synergies $S(i,j)$ sont élevées, l'inhibition empêche une croissance simultanée de tous les liens, limitant ainsi le risque d'un emballement global des pondérations.

- **Risques d'oscillations** : Toutefois, un réglage trop agressif de γ peut entraîner des oscillations ou un comportement chaotique, car la compétition latérale devient trop forte. Dans ce cas, les mises à jour peuvent s'alternancer de manière excessive, empêchant une convergence stable vers des valeurs fixes.

En résumé, γ agit comme un **facteur de contrainte** qui module la spécialisation des liens d'une entité et qui, en modulant la pression compétitive, influence directement la formation des clusters dans le réseau.

C. Ajustement empirique et stratégie de régulation

Le choix de γ doit être ajusté en fonction des exigences du problème et du comportement souhaité du SCN. En pratique, plusieurs approches sont envisageables :

- **Essais par paliers** : Il est courant de tester des valeurs telles que $\gamma = 0.01, 0.05, 0.1, 0.2$, etc., et d'observer la dynamique des mises à jour.
- **Adaptation dynamique** : On peut également concevoir une règle d'auto-ajustement où γ évolue en fonction de la somme des pondérations ou d'un indicateur de congestion. Par exemple, on pourrait définir une mise à jour de γ par :

$$\gamma(t+1) = \gamma(t) + \alpha \left[\sum_j \omega_{i,j}(t) - \beta \right],$$

où α et β sont des paramètres ajustables. Ce mécanisme permet de maintenir la somme des connexions autour d'un niveau prédéfini.

- **Normalisation** : Dans certaines applications, il est souhaitable de normaliser la somme des pondérations pour qu'elle reste inférieure à une valeur maximale (par exemple, 1). Ceci peut être réalisé en combinant γ avec d'autres techniques de **clipping** ou de normalisation des poids.

D. Implémentation Python avec Visualisation

Nous proposons ci-dessous une implémentation complète en Python qui simule la dynamique d'inhibition compétitive avec paramétrage de γ . Ce script calcule l'évolution des pondérations $\omega_{i,j}(t)$ pour une entité donnée disposant de plusieurs liens et affiche la trajectoire de chaque poids au cours des itérations.

```
import numpy as np
import matplotlib.pyplot as plt

# Paramètres de la mise à jour
eta = 0.05      # Taux d'apprentissage
tau = 1.0       # Facteur de décroissance
gamma = 0.05    # Coefficient d'inhibition (à ajuster pour observer différents comportements)
num_iterations = 150 # Nombre d'itérations
n_links = 5     # Nombre de liens sortants pour une entité E_i

# Définition des synergies S(i,j) pour chaque lien j
# Pour cet exemple, nous supposons que S(i,j) varie pour simuler différents niveaux de complémentarité
# Exemple : l'entité a une forte synergie avec les deux premiers liens, puis des synergies décroissantes.
S = np.array([0.8, 0.75, 0.4, 0.3, 0.2])

# Initialisation des pondérations omega pour l'entité E_i (un vecteur de taille n_links)
np.random.seed(42)
omega = np.random.uniform(0, 0.05, size=n_links)

# Stockage de l'évolution des pondérations pour la visualisation
omega_history = np.zeros((num_iterations + 1, n_links))
omega_history[0, :] = omega.copy()
```

```

# Simulation de la dynamique de mise à jour avec inhibition compétitive
for t in range(num_itérations):
    # Calcul de la somme totale des poids de l'entité E_i
    total_weight = np.sum(omega)

    # Mise à jour de chaque lien pour l'entité E_i
    for j in range(n_links):
        # Terme d'update additif classique
        delta_update = eta * (S[j] - tau * omega[j])
        # Terme d'inhibition compétitive: on soustrait la somme des poids des autres liens
        inhibition = gamma * (total_weight - omega[j])
        # Mise à jour totale de omega[j]
        omega[j] = omega[j] + delta_update - inhibition
        # Application d'un clipping pour éviter des valeurs négatives
        if omega[j] < 0:
            omega[j] = 0
    # Enregistrement des poids de l'itération courante
    omega_history[t + 1, :] = omega.copy()

# Calcul théorique du point fixe sans inhibition (pour comparaison) : omega* = S / tau
omega_fixed = S / tau

# Affichage graphique de l'évolution des pondérations
plt.figure(figsize=(10, 6))
for j in range(n_links):
    plt.plot(omega_history[:, j], marker='o', linestyle='-', label=f'$\omega_{\{j+1\}}(t)$')
for j in range(n_links):
    plt.axhline(y=omega_fixed[j], linestyle='--', color=f'C{j}', label=f'$S_{\{j+1\}}/\tau$')

plt.title("Évolution des pondérations $\omega_{i,j}(t)$ avec inhibition compétitive ($\gamma = {:.2f}$)".format(gamma))
plt.xlabel("Itérations $t$")
plt.ylabel("$\omega_{i,j}(t)$")
plt.legend(loc="upper right")
plt.grid(True)
plt.show()

```

E. Explications de l'Implémentation

Dans ce script :

- **Initialisation et Paramètres :**

Nous fixons les paramètres $\eta = 0.05$, $\tau = 1.0$ et $\gamma = 0.05$. Le nombre de liens ($n_{\text{links}} = 5$) correspond aux connexions sortantes d'une entité \mathcal{E}_i . Le vecteur de synergies S est défini pour simuler différents niveaux de complémentarité entre l'entité et ses cibles (par exemple, 0.8, 0.75, 0.4, 0.3, 0.2). Les pondérations ω sont initialisées avec de petites valeurs aléatoires proches de zéro.

- **Mise à jour des Pondérations :**

Pour chaque itération, nous calculons d'abord la somme totale des poids pour l'entité. Pour chaque lien j , nous appliquons la mise à jour additive standard $\Delta_{\text{update}} = \eta(S(j) - \tau \omega[j])$ puis soustrayons le terme d'inhibition γ multiplié par la somme des autres poids. Ce terme d'inhibition permet de réduire la capacité de l'entité à renforcer simultanément plusieurs liens. Enfin, nous appliquons un mécanisme de clipping pour garantir que les poids restent positifs.

- **Visualisation :**

La trajectoire de chaque pondération $\omega_j(t)$ est enregistrée et affichée sur un graphique. Des lignes horizontales en pointillés indiquent les valeurs théoriques $\frac{S(j)}{\tau}$ pour référence, ce qui permet de comparer la convergence effective avec le point fixe attendu en l'absence d'inhibition.

F. Conclusion

L'ajout du terme d'inhibition $-\gamma \sum_{k \neq j} \omega_{i,k}(t)$ permet d'instaurer une **compétition** entre les liens d'une même entité, ce qui force cette dernière à concentrer ses ressources sur un nombre limité de connexions fortes. Ce mécanisme aide à la formation de **clusters** plus distincts et empêche la croissance simultanée de tous les liens. La formule de mise à jour complète :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] - \gamma \sum_{k \neq j} \omega_{i,k}(t)$$

est ainsi mise en œuvre de manière simple en Python, et la simulation graphique permet d'observer l'effet de l'inhibition compétitive sur la dynamique des pondérations. Ce développement illustre l'importance du paramètre γ et montre comment, en ajustant sa valeur, on peut influencer la spécialisation et la stabilité des connexions dans un SCN.

5.5.2.3. Application concrète : limiter la somme des liens sortants par entité

Dans de nombreux scénarios d'application d'un **Synergistic Connection Network** (SCN), il est essentiel que chaque entité \mathcal{E}_i ne se lie pas de manière excessive à l'ensemble des autres entités. En effet, dans des systèmes inspirés par le fonctionnement des réseaux neuronaux ou par des modèles de ressources limitées en systèmes cognitifs, il est souvent souhaitable de borner la somme des pondérations associées aux liens sortants de \mathcal{E}_i afin de forcer cette entité à "choisir" les partenaires les plus pertinents. Le mécanisme d'inhibition compétitive, introduit sous la forme du terme

$$-\gamma \sum_{k \neq j} \omega_{i,k}(t),$$

permet justement de réaliser cette contrainte. En ajoutant ce terme dans la règle de mise à jour, la dynamique de $\omega_{i,j}$ se voit modifiée de sorte que la croissance d'un lien est freinée par la présence d'autres liens déjà établis.

A. Principe et justification

La règle de mise à jour additive classique sans inhibition s'exprime par

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

où η est le taux d'apprentissage et τ le paramètre de décroissance. L'ajout d'un terme d'inhibition introduit une compétition entre les liens sortants d'une même entité. La formule devient alors

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] - \gamma \sum_{k \neq j} \omega_{i,k}(t).$$

Le terme $-\gamma \sum_{k \neq j} \omega_{i,k}(t)$ agit comme un prélèvement ou une pénalité sur chaque lien $\omega_{i,j}$ en fonction de la somme des autres liens sortants de \mathcal{E}_i . On peut ainsi interpréter γ comme le **coefficient d'inhibition**, qui définit le budget de connexion de l'entité. Lorsque la somme des autres liens est importante, l'inhibition est plus forte, ce qui freine davantage l'augmentation de $\omega_{i,j}$.

Ce mécanisme de limitation permet d'éviter que l'entité répartisse ses ressources de manière uniforme sur l'ensemble des connexions, favorisant ainsi une spécialisation où seules quelques connexions significatives émergent, tandis que les autres restent faibles. D'un point de vue neuro-inspiré, cela correspond à l'idée d'**inhibition latérale**, dans laquelle un neurone, une fois fortement activé, limite l'activation simultanée de ses voisins. Cette analogie trouve également

un écho dans les modèles économiques où une taxe ou un coût additionnel est appliqué lorsque les ressources sont réparties sur trop de partenariats.

B. Étude mathématique

Afin d'analyser l'effet de ce mécanisme sur la somme totale des liens sortants, nous notons

$$\Sigma_i(t) = \sum_j \omega_{i,j}(t).$$

En sommant la règle de mise à jour sur tous les j pour une entité \mathcal{E}_i , on obtient :

$$\Sigma_i(t+1) = \sum_j \omega_{i,j}(t) + \eta \sum_j [S(i,j) - \tau \omega_{i,j}(t)] - \gamma \sum_j \sum_{k \neq j} \omega_{i,k}(t).$$

Le double-somme $\sum_j \sum_{k \neq j} \omega_{i,k}(t)$ se simplifie en $(n_i - 1) \Sigma_i(t)$, où n_i est le nombre de liens sortants de l'entité \mathcal{E}_i . Ainsi, l'inhibition introduit un terme proportionnel à $\Sigma_i(t)$ lui-même, ce qui tend à stabiliser la somme totale des liens, empêchant ainsi une croissance incontrôlée.

C. Implémentation Python avec Visualisation

Nous proposons ci-dessous une implémentation en Python qui simule la dynamique de mise à jour avec inhibition compétitive pour une entité \mathcal{E}_i disposant de plusieurs liens. L'exemple permet de visualiser comment la somme des liens se régule et comment les différents liens évoluent au fil des itérations.

```
import numpy as np
import matplotlib.pyplot as plt

# Paramètres de la simulation
eta = 0.05      # Taux d'apprentissage
tau = 1.0       # Facteur de décroissance
gamma = 0.1     # Coefficient d'inhibition (à ajuster pour voir l'effet sur la somme)
num_iterations = 150 # Nombre total d'itérations
n_links = 10    # Nombre de liens sortants pour une entité  $\mathcal{E}_i$ 

# Définir les synergies  $S(i,j)$  pour chaque lien  $j$ .
# Ici,  $S$  est un vecteur de valeurs simulant différentes "forces" de synergie.
# On suppose que certains liens ont une synergie élevée et d'autres faible.
S = np.array([0.8, 0.75, 0.6, 0.55, 0.5, 0.4, 0.35, 0.3, 0.25, 0.2])

# Initialisation des pondérations omega pour l'entité  $\mathcal{E}_i$  avec de faibles valeurs aléatoires
np.random.seed(42)
omega = np.random.uniform(0, 0.05, size=n_links)

# Stocker l'évolution des pondérations pour visualisation
omega_history = np.zeros((num_iterations + 1, n_links))
sum_history = np.zeros(num_iterations + 1)
omega_history[0, :] = omega.copy()
sum_history[0] = np.sum(omega)

# Simulation de la mise à jour avec inhibition compétitive
for t in range(num_iterations):
    # Calcul de la somme des pondérations de l'entité  $\mathcal{E}_i$  à l'itération  $t$ 
    total_weight = np.sum(omega)

    # Mise à jour de chaque lien pour l'entité  $\mathcal{E}_i$ 
    for j in range(n_links):
```

```

# Terme d'update additif classique
delta_update = eta * (S[j] - tau * omega[j])
# Terme d'inhibition compétitive : somme des autres liens
inhibition = gamma * (total_weight - omega[j])
# Mise à jour totale
omega[j] = omega[j] + delta_update - inhibition
# Clipping : s'assurer que les pondérations restent positives
if omega[j] < 0:
    omega[j] = 0

# Enregistrement des pondérations et de la somme totale pour la visualisation
omega_history[t + 1, :] = omega.copy()
sum_history[t + 1] = np.sum(omega)

# Calcul théorique du point fixe pour chaque lien sans inhibition (pour comparaison) : omega* = S / tau
omega_fixed = S / tau

# Affichage graphique de l'évolution de chaque pondération
plt.figure(figsize=(12, 6))
for j in range(n_links):
    plt.plot(omega_history[:, j], marker='o', linestyle='-', label=f'$\\omega_{{{j+1}}}(t)$')
for j in range(n_links):
    plt.axhline(y=omega_fixed[j], linestyle='--', color=f'C{j}', label=f'$S({j+1})/\\tau$')

plt.title("Évolution des pondérations $\\omega_{i,j}(t)$ avec inhibition compétitive ($\\gamma = {:.2f}$)".format(gamma))
plt.xlabel("Itérations $t$")
plt.ylabel("$\\omega_{i,j}(t)$")
plt.legend(loc="upper right", fontsize=8)
plt.grid(True)
plt.show()

# Affichage graphique de l'évolution de la somme des pondérations par entité
plt.figure(figsize=(10, 5))
plt.plot(sum_history, marker='s', linestyle='-', color='purple', label='$\\Sigma_i(t)$')
plt.title("Évolution de la somme des liens sortants $\\Sigma_i(t)$")
plt.xlabel("Itérations $t$")
plt.ylabel("$\\Sigma_i(t)$")
plt.legend()
plt.grid(True)
plt.show()

```

D. Explications détaillées de l'implémentation

Dans cet exemple, nous modélisons une entité \mathcal{E}_i dotée de $n_{\text{links}} = 10$ connexions sortantes, chacune caractérisée par une **synergie** $S(i, j)$ fixée. Les pondérations $\omega_{i,j}(t)$ sont initialisées avec de faibles valeurs aléatoires pour simuler un départ quasi nul.

La mise à jour des pondérations s'effectue en deux parties. D'une part, le terme additive classique, $\eta [S(i, j) - \tau \omega_{i,j}(t)]$, tend à faire converger chaque lien vers la valeur $\frac{S(i,j)}{\tau}$ en l'absence d'inhibition. D'autre part, le terme d'inhibition, $-\gamma \sum_{k \neq j} \omega_{i,k}(t)$, représente le prélèvement effectué sur le lien $\omega_{i,j}$ en fonction de la somme des autres pondérations. Ce mécanisme force l'entité à limiter la somme totale de ses connexions, conduisant à une répartition sélective des ressources.

Le script enregistre, à chaque itération, l'évolution de chaque pondération ainsi que la somme totale $\Sigma_i(t) = \sum_j \omega_{i,j}(t)$. Ces données sont ensuite visualisées par deux graphiques : l'un montre l'évolution individuelle de chaque $\omega_{i,j}$

(avec des lignes horizontales indiquant les valeurs théoriques sans inhibition), et l'autre trace l'évolution de la somme totale des liens de l'entité. Ainsi, on peut observer comment le mécanisme d'inhibition stabilise la somme des pondérations et favorise la spécialisation.

E. Conclusion

L'introduction du terme $-\gamma \sum_{k \neq j} \omega_{i,k}(t)$ dans la mise à jour permet d'instaurer une compétition interne entre les liens sortants d'une entité. Ce mécanisme de **budget** impose que l'entité ne puisse renforcer simultanément tous ses liens, favorisant ainsi la formation de clusters denses et la spécialisation des connexions. L'implémentation Python présentée illustre concrètement cette dynamique en montrant, à travers des graphiques, la convergence des pondérations et la régulation de leur somme. Le réglage approprié de γ est crucial pour obtenir un équilibre entre l'amplification des liens forts et le freinage des liens faibles, garantissant ainsi une auto-organisation efficace et cohérente dans le SCN.

Ce développement, associant explications mathématiques et implémentation pratique, fournit une base solide pour comprendre et exploiter le mécanisme d'inhibition compétitive dans un cadre de Deep Synergy Learning.

5.5.3. Saturation

La **saturation** constitue un autre mécanisme majeur pour contrôler la croissance des pondérations $\omega_{i,j}$ dans un SCN. Même si l'on dispose d'une dynamique de base (section 5.5.1) et d'un module d'inhibition/contrôle (section 5.5.2), il subsiste fréquemment un risque d'emballement lorsque plusieurs liens $\omega_{i,j}$ se renforcent simultanément ou lorsqu'une seule connexion prend une ampleur disproportionnée. La **saturation** permet alors de **clorre** la valeur d'un lien au-delà d'un certain plafond ω_{\max} .

D'un point de vue **mathématique**, la saturation introduit une **barrière** supérieure : même si l'itération calcule une mise à jour positive qui pousserait $\omega_{i,j}$ à dépasser ω_{\max} , on la "coupe" (clipping) pour éviter qu'elle ne franchisse ce seuil. Dans cette section (5.5.3), nous décrirons le **clipping** (5.5.3.1) et son impact, puis nous analyserons comment choisir ω_{\max} et en quoi cela influe sur la structure de clusters (5.5.3.2).

5.5.3.1. Clipping : $\omega_{i,j} \leftarrow \min(\omega_{i,j}, \omega_{\max})$

Dans un **Synergistic Connection Network** (SCN), les mécanismes d'inhibition compétitive (voir notamment la section 5.5.2.2) permettent de réguler la croissance collective des liens sortants d'une entité \mathcal{E}_i . Néanmoins, il reste possible qu'un ou plusieurs liens, par leur dynamique interne ou en raison de paramètres mal ajustés (par exemple, un taux d'apprentissage η trop élevé ou une décroissance τ trop faible), atteignent des valeurs excessives. Pour éviter cette situation, on introduit le **clipping**, qui consiste à borner chaque pondération individuellement en imposant que :

$$\omega_{i,j}(t+1) \leftarrow \min(\omega_{i,j}(t+1), \omega_{\max}).$$

A. Principe du Clipping

Le **clipping** agit comme un opérateur de **saturation** appliqué en post-traitement à la mise à jour des poids. La formule de base d'une mise à jour additive dans un SCN s'exprime généralement par :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)].$$

Lorsque cette règle aboutit à une valeur de $\omega_{i,j}(t+1)$ supérieure à un seuil prédéfini ω_{\max} , le clipping intervient en forçant :

$$\omega_{i,j}(t+1) \leftarrow \min(\omega_{i,j}(t+1), \omega_{\max}).$$

Ce procédé garantit que, quelle que soit la synergie $S(i,j)$ ou les conditions locales de mise à jour, aucune connexion ne pourra dépasser la valeur ω_{\max} . Cette opération est particulièrement pertinente pour éviter une croissance

incontrôlée des poids, phénomène pouvant être observé lorsque des paramètres de mise à jour conduisent à une dynamique exponentielle ou à des oscillations.

B. Rôle Mathématique et Effets sur la Convergence

D'un point de vue **mathématique**, le clipping agit comme une contrainte non linéaire dans le système. Sans clipping, le point fixe théorique pour chaque lien, en l'absence d'inhibition supplémentaire, serait donné par

$$\omega_{i,j}^* = \frac{S(i,j)}{\tau}.$$

Cependant, lorsque $\frac{S(i,j)}{\tau}$ dépasse la valeur ω_{\max} , le clipping force le lien à se stabiliser à ω_{\max} plutôt que de continuer à croître. En d'autres termes, le système évolue vers un "point fixe saturé". On obtient ainsi une dynamique par morceaux où la mise à jour se déroule selon la règle linéaire jusqu'à ce que $\omega_{i,j}$ atteigne ω_{\max} , puis la valeur est fixée.

Cette approche présente l'avantage de prévenir l'emballement de certaines pondérations, tout en permettant aux autres liens de converger vers leurs valeurs théoriques lorsque celles-ci sont inférieures à ω_{\max} . En conséquence, le SCN est amené à concentrer ses ressources sur un nombre limité de liens forts, favorisant la formation de clusters plus distincts et économiquement répartis, sans pour autant perdre l'information sur la synergie.

C. Implémentation en Python avec Graphiques

Nous présentons ci-dessous un code Python complet illustrant la dynamique d'un SCN avec mise à jour additive, incluant l'inhibition compétitive et l'opération de clipping. Ce script simule la mise à jour de la pondération d'un lien et trace son évolution au fil des itérations, ainsi que l'évolution de la somme des liens sortants pour une entité donnée.

```
import numpy as np
import matplotlib.pyplot as plt

# Paramètres de la simulation
eta = 0.05      # Taux d'apprentissage
tau = 1.0       # Facteur de décroissance
gamma = 0.1     # Coefficient d'inhibition compétitive
omega_max = 0.7 # Valeur de clipping maximale pour chaque lien
num_iterations = 150 # Nombre d'itérations
n_links = 10    # Nombre de liens sortants pour une entité E_i

# Définition des synergies S(i,j) pour l'entité E_i (vecteur de synergies pour chaque lien)
# Supposons que les synergies varient pour illustrer des cas où certains liens devraient théoriquement dépasser omega_max.
S = np.array([0.8, 0.75, 0.6, 0.55, 0.5, 0.4, 0.35, 0.3, 0.25, 0.2])

# Initialisation des pondérations omega pour l'entité E_i avec de faibles valeurs aléatoires
np.random.seed(42)
omega = np.random.uniform(0, 0.05, size=n_links)

# Stockage de l'évolution des pondérations pour visualisation
omega_history = np.zeros((num_iterations + 1, n_links))
sum_history = np.zeros(num_iterations + 1)
omega_history[0, :] = omega.copy()
sum_history[0] = np.sum(omega)

# Simulation de la mise à jour avec inhibition compétitive et clipping
for t in range(num_iterations):
    total_weight = np.sum(omega) # Somme des pondérations pour l'entité E_i à l'itération t
    for j in range(n_links):
        # Calcul du terme de mise à jour additive
```

```

delta_update = eta * (S[j] - tau * omega[j])
# Calcul du terme d'inhibition compétitive : somme des autres liens (pour k != j)
inhibition = gamma * (total_weight - omega[j])
# Mise à jour additive avec inhibition
new_omega = omega[j] + delta_update - inhibition
# Application du clipping pour s'assurer que new_omega ne dépasse pas omega_max
new_omega = min(new_omega, omega_max)
# Garantie de non-négativité
omega[j] = max(new_omega, 0)
# Enregistrement de l'évolution des pondérations et de la somme totale
omega_history[t + 1, :] = omega.copy()
sum_history[t + 1] = np.sum(omega)

# Calcul théorique du point fixe sans clipping pour comparaison (omega* = S / tau)
omega_fixed = S / tau

# Affichage graphique de l'évolution de chaque pondération
plt.figure(figsize=(12, 6))
for j in range(n_links):
    plt.plot(omega_history[:, j], marker='o', linestyle='-', label=f'$\\omega_{{{j+1}}}(t)$')
    # Ajout d'une ligne horizontale pour la valeur théorique sans clipping, seulement si elle est inférieure à omega_max
    ax
    if omega_fixed[j] <= omega_max:
        plt.axhline(y=omega_fixed[j], linestyle='--', color=f'C{j}', label=f'$S({j+1})/\\tau$')

plt.title("Évolution des pondérations $\\omega_{{{i,j}}}(t)$ avec inhibition compétitive et clipping")
plt.xlabel("Itérations $t$")
plt.ylabel("$\\omega_{{{i,j}}}(t)$")
plt.legend(loc="upper right", fontsize=8)
plt.grid(True)
plt.show()

# Affichage graphique de l'évolution de la somme des pondérations pour l'entité E_i
plt.figure(figsize=(10, 5))
plt.plot(sum_history, marker='s', linestyle='-', color='purple', label='$\\Sigma_i(t)$')
plt.title("Évolution de la somme des liens sortants $\\Sigma_i(t)$")
plt.xlabel("Itérations $t$")
plt.ylabel("$\\Sigma_i(t)$")
plt.legend()
plt.grid(True)
plt.show()

```

D. Explications Complètes

Dans cet exemple, nous simulons la dynamique d'un lien sortant d'une entité \mathcal{E}_i possédant $n_{\text{links}} = 10$ connexions. Chaque lien $\omega_{i,j}(t)$ évolue selon une règle additive classique à laquelle on ajoute un terme d'inhibition compétitive et, enfin, une opération de clipping. Plus précisément :

- **Mise à jour additive :**

Chaque pondération est mise à jour en ajoutant le terme $\eta [S(j) - \tau \omega_{i,j}(t)]$. Ce terme tend à rapprocher $\omega_{i,j}$ de la valeur théorique $S(j)/\tau$.

- **Inhibition compétitive :**

La somme $\sum_{k \neq j} \omega_{i,k}(t)$ est calculée pour l'entité, et le terme $-\gamma \times (\text{total des autres liens})$ est soustrait à la mise à jour de chaque lien. Ainsi, si d'autres liens sont déjà élevés, l'augmentation de $\omega_{i,j}$ sera freinée.

- **Clipping :**

Après avoir appliqué la mise à jour, nous utilisons l'opérateur $\min(\cdot, \omega_{\max})$ pour garantir que la valeur de $\omega_{i,j}(t+1)$ ne dépasse jamais le seuil fixé ω_{\max} . Ce mécanisme empêche toute croissance excessive, même si la synergie $S(j)$ et les autres paramètres favoriseraient une augmentation trop importante.

- **Visualisation :**

Nous traçons deux graphiques : le premier montre l'évolution individuelle de chaque lien $\omega_{i,j}(t)$ au fil des itérations, avec en point de référence les valeurs théoriques $S(j)/\tau$ lorsque celles-ci sont inférieures à ω_{\max} . Le second graphique trace l'évolution de la somme totale des liens sortants $\Sigma_i(t)$ pour l'entité, permettant de vérifier l'effet du mécanisme de "budget" sur la répartition globale des ressources.

E. Conclusion

L'opération de **clipping** constitue un mécanisme essentiel pour stabiliser la dynamique des pondérations dans un SCN, en imposant une borne supérieure ω_{\max} à chaque lien. Ce procédé, combiné avec l'inhibition compétitive, permet de contrôler la répartition des ressources de connexion d'une entité, empêchant ainsi l'emballement des poids et favorisant la formation de clusters bien définis. L'exemple en Python présenté ici, avec ses graphiques, illustre concrètement comment la mise à jour de $\omega_{i,j}$ converge vers des valeurs régulées et comment la somme totale des liens se stabilise sous l'effet de ces mécanismes.

Ce développement, intégrant explications mathématiques et implémentation pratique, fournit une base solide pour comprendre et exploiter le mécanisme de clipping dans le cadre du Deep Synergy Learning.

5.5.3.2. Réglage de ω_{\max} et son effet sur l'émergence de clusters

Lorsqu'on introduit un **clipping** dans la mise à jour des pondérations, nous imposons la contrainte suivante :

$$\omega_{i,j}(t+1) \leftarrow \min(\omega_{i,j}(t+1), \omega_{\max}).$$

Ce procédé, bien qu'il puisse paraître comme un simple ajustement, a un impact majeur sur la **dynamique** du SCN et la qualité des **clusters** qui émergent. En effet, le choix de la valeur ω_{\max} détermine dans quelle mesure chaque lien $\omega_{i,j}$ pourra se développer, et par conséquent comment les entités \mathcal{E}_i vont concentrer leurs ressources de connexion.

A. Importance Fondamentale du Choix de ω_{\max}

Le mécanisme de clipping agit localement sur chaque connexion en contraignant la valeur maximale atteinte par $\omega_{i,j}$. Formellement, dans la mise à jour additive classique, nous avons :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)].$$

Lorsque le résultat de cette mise à jour dépasse ω_{\max} , l'opérateur de clipping force :

$$\omega_{i,j}(t+1) \leftarrow \min(\omega_{i,j}(t+1), \omega_{\max}).$$

Ainsi, le point fixe théorique sans clipping, qui serait $\omega_{i,j}^* = \frac{S(i,j)}{\tau}$, est remplacé par :

$$\omega_{i,j}^* = \min\left(\frac{S(i,j)}{\tau}, \omega_{\max}\right).$$

Ce réglage permet de moduler la **sélectivité** du réseau : un ω_{\max} faible restreint la force maximale de chaque lien, favorisant une répartition plus uniforme, tandis qu'un plafond élevé permet à certains liens de devenir très forts et de se démarquer, ce qui peut conduire à des clusters hiérarchisés.

B. Impact sur la Structure de Clusters

Du point de vue de la formation de clusters, le choix de ω_{\max} influence directement la topologie du réseau :

- Si ω_{\max} est trop bas, même les liens qui devraient naturellement devenir forts seront limités. Cela peut conduire à une homogénéisation des connexions où la différenciation entre les liens forts et faibles est atténuée. Dans ce cas, le SCN aura tendance à former des groupes moins contrastés, car toutes les connexions atteignent presque le même niveau maximal.
- Inversement, un ω_{\max} élevé permet à certains liens de se développer pleinement selon leur synergie intrinsèque $S(i, j)$. Si plusieurs liens sont en compétition, il est alors possible qu'un ou quelques liens dominent l'ensemble des connexions sortantes d'une entité, menant à une formation de clusters très marqués, dans lesquels certains liens sont beaucoup plus forts que d'autres.

Ainsi, la valeur de ω_{\max} se révèle être un paramètre de **régulation** crucial. En le combinant avec d'autres mécanismes tels que l'inhibition (qui limite la somme totale des connexions d'un nœud), on obtient un contrôle fin sur la manière dont les entités sélectionnent leurs partenaires privilégiés dans le réseau.

C. Ajustement Empirique et Adaptatif

Dans la pratique, le choix de ω_{\max} peut être réalisé de façon empirique. Par exemple, si l'on connaît l'ordre de grandeur des valeurs de $S(i, j)/\tau$, il est judicieux de fixer ω_{\max} en fonction de ces valeurs. Une approche consiste à définir

$$\omega_{\max} = \alpha \cdot \max_{i,j} \left\{ \frac{S(i, j)}{\tau} \right\},$$

où $\alpha \geq 1$ est un facteur multiplicatif qui permet de ne pas trop contraindre la dynamique lorsque certaines synergies sont très élevées. Dans certains scénarios, il est également envisageable d'adapter dynamiquement ω_{\max} au fil des itérations (similaire à un recuit simulé), afin de permettre une phase exploratoire initiale suivie d'une stabilisation plus stricte.

D. Implémentation Python Complète avec Graphiques

Nous présentons ci-dessous un exemple complet en Python qui illustre l'effet du paramétrage de ω_{\max} sur la dynamique d'un SCN. Ce script simule la mise à jour d'une matrice de pondérations pour une entité possédant plusieurs liens, intègre le clipping, et affiche l'évolution des pondérations au fil des itérations ainsi que la répartition finale sous forme de heatmap.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Paramètres de la simulation
```

```
eta = 0.05      # Taux d'apprentissage
tau = 1.0       # Facteur de décroissance
gamma = 0.1     # Coefficient d'inhibition compétitive
num_iterations = 150 # Nombre d'itérations
n_links = 10    # Nombre de liens sortants pour une entité
```

```
# Paramètre de clipping : on va comparer différents réglages
```

```
omega_max_values = [0.5, 1.0, 2.0]
```

```
# Définition des synergies S(i,j) pour l'entité E_i
```

```
# Pour illustrer, nous fixons des valeurs différentes pour chaque lien.
```

```

S = np.linspace(0.8, 0.2, n_links) # Synergies décroissantes de 0.8 à 0.2

# Fonction de mise à jour avec inhibition et clipping
def simulate_update(omega_max, eta, tau, gamma, num_iterations, S):
    # Initialisation des pondérations avec de faibles valeurs aléatoires
    np.random.seed(42)
    omega = np.random.uniform(0, 0.05, size=n_links)
    omega_history = np.zeros((num_iterations + 1, n_links))
    omega_history[0, :] = omega.copy()

    # Simulation de la mise à jour sur num_iterations itérations
    for t in range(num_iterations):
        total_weight = np.sum(omega) # Somme des pondérations pour l'entité E_i à l'itération t
        for j in range(n_links):
            # Calcul de la mise à jour additive de base
            delta_update = eta * (S[j] - tau * omega[j])
            # Calcul du terme d'inhibition : somme des autres liens pour l'entité E_i
            inhibition = gamma * (total_weight - omega[j])
            # Nouvelle pondération avant clipping
            new_omega = omega[j] + delta_update - inhibition
            # Application du clipping : la nouvelle valeur ne doit pas dépasser omega_max
            new_omega = min(new_omega, omega_max)
            # On garantit que la pondération reste positive
            omega[j] = max(new_omega, 0)
            omega_history[t + 1, :] = omega.copy()

    return omega_history

# Simulation pour chaque valeur de omega_max
histories = {}
for omega_max in omega_max_values:
    histories[omega_max] = simulate_update(omega_max, eta, tau, gamma, num_iterations, S)

# Affichage des trajectoires pour chaque réglage de omega_max
plt.figure(figsize=(12, 8))
for omega_max, history in histories.items():
    for j in range(n_links):
        plt.plot(history[:, j], label=f'Lien {j+1},  $\omega_{\max}$ ={omega_max}', alpha=0.7)
    plt.title(f'Évolution des pondérations pour  $\omega_{\max}$  = {omega_max}')
    plt.xlabel("Itérations")
    plt.ylabel("Pondération  $\omega$ ")
    plt.legend(fontsize=8, loc="upper right")
    plt.grid(True)
    plt.show()

# Affichage d'une heatmap finale pour comparer les distributions de pondérations
plt.figure(figsize=(14, 4))
for idx, omega_max in enumerate(omega_max_values):
    final_weights = histories[omega_max][-1, :].reshape(1, -1)
    plt.subplot(1, len(omega_max_values), idx+1)
    sns.heatmap(final_weights, annot=True, cmap="viridis", cbar=True, vmin=0, vmax=max(omega_max_values))
    plt.title(f' $\omega_{\max}$  = {omega_max}')
    plt.xlabel("Lien index")
    plt.ylabel("Entité E_i")
plt.tight_layout()
plt.show()

```

F. Explications Détaillées

Dans cet exemple, nous considérons une entité \mathcal{E}_i qui possède $n_{\text{links}} = 10$ connexions, chacune ayant une synergie fixée $S(j)$ variant linéairement de 0.8 à 0.2. La mise à jour de chaque lien suit l'équation :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(j) - \tau \omega_{i,j}(t)] - \gamma \sum_{k \neq j} \omega_{i,k}(t).$$

Après avoir calculé la mise à jour, nous appliquons le **clipping** :

$$\omega_{i,j}(t+1) \leftarrow \min(\omega_{i,j}(t+1), \omega_{\max}).$$

Nous comparons trois valeurs de ω_{\max} : 0.5, 1.0 et 2.0. La fonction *simulate_update* effectue la mise à jour sur un nombre fixé d'itérations et retourne l'historique des pondérations pour chaque lien. Ensuite, nous traçons, pour chaque réglage, l'évolution des valeurs de $\omega_{i,j}(t)$ ainsi qu'une heatmap finale montrant la distribution des pondérations après la convergence.

L'analyse graphique permet de constater que :

- Pour un ω_{\max} faible (par exemple 0.5), les pondérations sont strictement limitées, ce qui peut conduire à une uniformisation des liens et à une perte de différenciation dans la formation de clusters.
- Pour un ω_{\max} plus élevé (par exemple 1.0 ou 2.0), les liens qui bénéficient d'une synergie forte peuvent atteindre des valeurs proches de leur point fixe théorique (sous réserve de l'effet inhibiteur), permettant ainsi la mise en évidence de clusters plus marqués et hiérarchisés.

G. Conclusion

Le paramétrage de ω_{\max} est crucial pour contrôler la dynamique des pondérations dans un SCN. Le clipping, en imposant une saturation locale sur chaque lien, agit comme une contrainte de ressources permettant de limiter l'emballement de certains poids et de favoriser une spécialisation des connexions. En ajustant ω_{\max} et en combinant cet effet avec l'inhibition compétitive, on module la formation des clusters et on contrôle la répartition des ressources de connexion. L'implémentation Python présentée ici, avec ses visualisations, offre un exemple concret de l'impact du réglage de ω_{\max} sur l'émergence des structures dans un SCN, fournissant ainsi un outil précieux pour l'expérimentation et l'analyse en Deep Synergy Learning.

Cette approche intégrée, alliant explications mathématiques détaillées et implémentation pratique, permet d'illustrer la puissance du clipping dans la stabilisation de la dynamique et dans la formation de clusters au sein d'un SCN.

5.5.4. Recuit Simulé ou Bruit

En plus des mécanismes d'inhibition (section 5.5.2) et de saturation (section 5.5.3), un autre procédé couramment employé dans la mise à jour des pondérations $\omega_{i,j}$ est l'**injection de bruit** ou l'utilisation d'une **méthode de recuit simulé**. Ces approches visent à **perturber** périodiquement le système pour **éviter** qu'il ne se fige trop tôt dans un minimum local, ou pour mieux explorer l'espace des configurations possibles. Sur le plan mathématique, elles introduisent un **terme stochastique** ou un **paramètre de "température"** qui se modifie au fil des itérations.

5.5.3.2. Réglage de ω_{\max} et son effet sur l'émergence de clusters

Dans un **Synergistic Connection Network** (SCN), la mise à jour des pondérations $\omega_{i,j}$ se fait typiquement selon une règle additive telle que

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

où $\eta > 0$ est le taux d'apprentissage et $\tau > 0$ représente le coefficient de décroissance. Afin d'empêcher qu'un ou plusieurs liens ne deviennent excessivement forts — situation qui pourrait fausser la dynamique du réseau et masquer la spécialisation souhaitée —, on introduit un **mécanisme de clipping**. Ce procédé consiste à contraindre chaque valeur $\omega_{i,j}(t + 1)$ par la relation

$$\omega_{i,j}(t + 1) \leftarrow \min(\omega_{i,j}(t + 1), \omega_{\max}),$$

ce qui force chaque connexion à ne pas dépasser un plafond fixé, ω_{\max} . Sur le plan mathématique, en l'absence de clipping le point fixe théorique pour la mise à jour additive serait

$$\omega_{i,j}^* = \frac{S(i,j)}{\tau}.$$

Or, dans le cas où $\frac{S(i,j)}{\tau} > \omega_{\max}$, la dynamique est « coupée » et le lien se stabilise à ω_{\max} . Cette contrainte a plusieurs conséquences sur la formation des clusters au sein du SCN :

- **Uniformisation versus différenciation** : Si ω_{\max} est trop faible, même les liens qui devraient être forts seront plafonnés à une valeur inférieure, aboutissant à une homogénéisation des connexions et à des clusters moins différenciés.
- **Hiérarchisation des liens** : Un ω_{\max} élevé permet à certains liens de se développer pleinement (selon leur synergie intrinsèque $S(i,j)$) tandis que d'autres restent faibles. On obtient alors une structure de clusters plus contrastée, avec des liens dominants entre certaines entités qui se distinguent nettement des connexions moins significatives.
- **Interaction avec l'inhibition** : Le clipping agit sur chaque lien individuellement, alors que l'inhibition (par exemple, un terme du type $-\gamma \sum_{k \neq j} \omega_{i,k}$) régule la somme des connexions sortantes d'une entité. Ensemble, ces deux mécanismes permettent à l'entité de concentrer son « budget de connexion » sur quelques partenaires privilégiés.

La détermination de ω_{\max} est donc cruciale : elle doit être choisie en fonction de l'échelle des synergies $S(i,j)$ et du coefficient τ pour permettre une discrimination effective entre les connexions. Dans certains cas, il est même envisageable de faire évoluer ω_{\max} au fil des itérations (similaire à un recuit simulé) afin de favoriser une phase d'exploration initiale suivie d'une consolidation des clusters.

Implémentation Python avec Visualisations

Nous proposons ci-dessous un exemple complet en Python. Ce script simule un SCN où n entités sont organisées en clusters grâce à une matrice de synergie **S** dotée d'une structure en blocs. La mise à jour des pondérations suit la règle additive avec clipping :

$$\omega_{i,j}(t + 1) = \min(\omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)], \omega_{\max}).$$

Nous étudierons l'impact de différents réglages de ω_{\max} sur la formation des clusters.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# -----
# Paramètres de la simulation
# -----
np.random.seed(42)

n = 50                # nombre total d'entités
```



```

n_clusters = 3          # nombre de clusters souhaités
eta = 0.05              # taux d'apprentissage
tau = 1.0               # coefficient de décroissance
num_iterations = 200    # nombre d'itérations
# On va tester différents plafonds
omega_max_values = [0.3, 0.7, 1.5]

# -----
# Création d'une matrice de synergie S
# -----
# On suppose que les entités sont réparties en n_clusters,
# avec une forte synergie intra-cluster et une faible synergie inter-cluster.

# Attribution aléatoire de clusters
cluster_labels = np.random.choice(n_clusters, n)

# Initialisation de la matrice S
S = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        if cluster_labels[i] == cluster_labels[j]:
            # Synergie élevée pour les entités dans le même cluster
            S[i, j] = 0.8
        else:
            # Synergie faible pour les entités dans des clusters différents
            S[i, j] = 0.2
# On s'assure que la diagonale est nulle (pas de self-connexion)
np.fill_diagonal(S, 0)

# -----
# Fonction de simulation avec clipping
# -----
def simulate_scn(omega_max, eta, tau, num_iterations, S):
    n = S.shape[0]
    # Initialisation des pondérations avec un bruit faible
    omega = np.random.uniform(0, 0.05, (n, n))
    np.fill_diagonal(omega, 0)
    omega_history = np.zeros((num_iterations+1, n, n))
    omega_history[0] = omega.copy()

    # Mise à jour itérative
    for t in range(num_iterations):
        # Calcul de la nouvelle matrice selon la règle additive
        omega_new = omega + eta * (S - tau * omega)
        # Application du clipping
        omega_new = np.minimum(omega_new, omega_max)
        # S'assurer que les self-connections restent nulles
        np.fill_diagonal(omega_new, 0)
        # Mettre à jour pour la prochaine itération
        omega = omega_new.copy()
        omega_history[t+1] = omega.copy()

    return omega_history

# -----
# Exécution de la simulation pour différents omega_max

```

```

# -----
results = { }
for omega_max in omega_max_values:
    results[omega_max] = simulate_scn(omega_max, eta, tau, num_iterations, S)

# -----
# Visualisation des trajectoires moyennes des pondérations
# -----
plt.figure(figsize=(12, 8))
for omega_max in omega_max_values:
    # Calcul de la moyenne des pondérations hors diagonale pour chaque itération
    mean_values = [np.mean(omega_history[np.triu_indices_from(omega_history, k=1)])
                    for omega_history in results[omega_max]]
    plt.plot(mean_values, label=f'ω_max = {omega_max} ')
plt.xlabel("Itérations")
plt.ylabel("Pondération moyenne (hors diagonale)")
plt.title("Évolution de la pondération moyenne selon ω_max")
plt.legend()
plt.grid(True)
plt.show()

# -----
# Visualisation finale : Heatmaps des matrices ω
# -----
plt.figure(figsize=(18, 5))
for idx, omega_max in enumerate(omega_max_values):
    final_omega = results[omega_max][-1]
    plt.subplot(1, len(omega_max_values), idx+1)
    sns.heatmap(final_omega, cmap="viridis", vmin=0, vmax=max(omega_max_values), annot=False)
    plt.title(f'Matrice ω finale (ω_max = {omega_max} )')
    plt.xlabel("Index j")
    plt.ylabel("Index i")
plt.tight_layout()
plt.show()

```

Explications

A. Principe du Clipping et Réglage de ω_{\max}

La formule de mise à jour initiale est donnée par

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)].$$

Pour éviter que certains liens ne deviennent trop forts, nous appliquons ensuite le **clipping** :

$$\omega_{i,j}(t+1) \leftarrow \min(\omega_{i,j}(t+1), \omega_{\max}).$$

Cette contrainte force chaque connexion à être bornée, ce qui, en fonction de la valeur choisie pour ω_{\max} , influencera la capacité des liens à atteindre leur point fixe théorique $\frac{S(i,j)}{\tau}$. Si $\frac{S(i,j)}{\tau}$ excède ω_{\max} , alors la croissance est limitée par cette borne.

B. Impact sur l'Émergence de Clusters

Dans notre simulation, la matrice de synergie S est construite de manière à favoriser une forte cohésion au sein d'un même cluster (valeur de 0.8) et une faible cohésion entre des entités de clusters différents (valeur de 0.2). Le clipping intervient ensuite pour influencer la distribution finale des poids $\omega_{i,j}$.

- Un ω_{\max} faible force toutes les pondérations à rester en dessous d'un seuil strict, ce qui peut engendrer des clusters moins différenciés.
- Un ω_{\max} élevé permet aux liens d'exprimer pleinement leurs différences, conduisant à une hiérarchisation plus marquée des connexions.

C. Implémentation et Visualisation

Le script Python présenté ci-dessus simule la dynamique de mise à jour du SCN sur un nombre donné d'itérations et pour plusieurs valeurs de ω_{\max} . La fonction *simulate_scn* réalise la mise à jour des poids selon la règle additive, applique le clipping, et stocke l'historique de la matrice de pondérations. Les graphiques générés montrent l'évolution moyenne des pondérations ainsi qu'une heatmap de la matrice finale, permettant d'observer l'impact du réglage de ω_{\max} sur la structure émergente.

Conclusion

L'ajustement de ω_{\max} joue un rôle clé dans la régulation des connexions au sein d'un SCN. Par l'intermédiaire du **clipping**, on impose une borne sur chaque pondération, empêchant ainsi un emballement individuel et favorisant une répartition plus sélective des ressources de connexion. L'implémentation en Python ci-dessus, accompagnée de visualisations graphiques, permet d'illustrer concrètement comment la dynamique évolue pour différentes valeurs de ω_{\max} et comment ce paramètre influence l'émergence des clusters dans le réseau. Cette approche offre ainsi un outil précieux pour expérimenter et affiner la structure auto-organisée d'un SCN dans des contextes variés d'apprentissage profond et de systèmes multi-agents.

5.5.4.1. Ajout d'un Terme Stochastique dans la Mise à Jour

A. Principes Généraux et Analogie avec le Recuit Simulé

Dans un SCN, la mise à jour classique des pondérations se réalise selon une règle additive déterministe de la forme

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

où $\eta > 0$ est le taux d'apprentissage et $\tau > 0$ représente le coefficient de décroissance. Cette mise à jour tend à conduire chaque $\omega_{i,j}$ vers un point fixe théorique $\omega_{i,j}^* = \frac{S(i,j)}{\tau}$. Toutefois, dans un environnement purement déterministe, le réseau peut se figer dans des configurations localement stables qui ne sont pas optimales, car certains liens potentiellement pertinents pourraient être négligés en raison du verrouillage dans un minimum local.

Pour pallier ce problème, un **terme stochastique** est ajouté à la règle de mise à jour. Ce terme, généralement de la forme $\alpha \xi_{i,j}(t)$, apporte une perturbation aléatoire contrôlée :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] + \alpha \xi_{i,j}(t).$$

Ici, $\xi_{i,j}(t)$ est une variable aléatoire tirée d'une distribution centrée, par exemple $\xi_{i,j}(t) \sim \mathcal{N}(0, \sigma^2)$ pour une distribution normale ou $\mathcal{U}(-\delta, \delta)$ pour une distribution uniforme, et $\alpha > 0$ ajuste l'amplitude relative de cette perturbation par rapport au terme déterministe.

Ce procédé s'inspire du **recuit simulé**, une méthode d'optimisation stochastique dans laquelle l'injection de bruit permet d'éviter un blocage prématuré dans un minimum local, tout en favorisant l'exploration de l'espace de recherche. Le terme stochastique permet ainsi à la dynamique du SCN de « rebondir » hors d'un point fixe potentiellement sous-optimal et d'explorer d'autres configurations qui pourraient conduire à une meilleure organisation globale du réseau.

B. Modélisation Mathématique et Impact sur la Dynamique

La dynamique complète, avec l'ajout du bruit, s'exprime par :

$$\Delta \omega_{i,j}(t) = \omega_{i,j}(t+1) - \omega_{i,j}(t) = \eta [S(i,j) - \tau \omega_{i,j}(t)] + \alpha \xi_{i,j}(t).$$

En l'absence de bruit ($\alpha = 0$), la suite $\{\omega_{i,j}(t)\}$ converge vers $\omega_{i,j}^* \approx \frac{S(i,j)}{\tau}$ (ou s'éteint si $S(i,j)$ est trop faible). L'ajout du terme $\alpha \xi_{i,j}(t)$ modifie cette convergence : plutôt que de converger exactement vers un point fixe, la dynamique converge vers une **distribution** autour du point fixe déterministe. La variance de cette distribution est fonction de l'amplitude α et de la variance σ^2 de la variable aléatoire.

En outre, la possibilité de faire varier α au fil du temps (par exemple en appliquant une décroissance exponentielle du type

$$\alpha(t+1) = \delta \alpha(t) \quad \text{avec} \quad 0 < \delta < 1,$$

) permet d'initier la dynamique dans un régime d'**exploration** (fort bruit initial) et de la laisser se stabiliser en réduisant progressivement le niveau de perturbation, à l'instar d'un processus de recuit.

C. Potentiel d'Exploration et Risques d'Instabilité

L'introduction d'un terme stochastique offre plusieurs avantages en termes d'exploration de l'espace de configuration. Parfois, même si la dynamique déterministe se rapproche d'un attracteur, de légères fluctuations permettent au système de s'extraire de minima locaux peu profonds pour atteindre des configurations plus globalement optimales. Cependant, un niveau de bruit excessif (un α trop grand ou une variance trop élevée pour $\xi_{i,j}(t)$) peut empêcher la convergence, générant des oscillations persistantes ou même un comportement chaotique.

Le défi consiste donc à **équilibrer** le niveau d'exploration induit par le terme stochastique avec la nécessité de convergence pour que le SCN puisse former des **clusters** stables et significatifs.

D. Implémentation Python Complète avec Graphiques

Nous présentons ci-dessous une implémentation en Python qui simule la mise à jour des pondérations dans un SCN avec ajout d'un terme stochastique. Le code inclut la visualisation graphique de la dynamique de quelques liens pour observer l'effet du bruit et de la décroissance de α .

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Fixation d'une graine pour la reproductibilité
np.random.seed(42)

# -----
# Paramètres de la simulation
# -----
n = 50 # Nombre d'entités (pour la matrice, on considère une simulation dense)
eta = 0.05 # Taux d'apprentissage
tau = 1.0 # Coefficient de décroissance
initial_alpha = 0.1 # Amplitude initiale du terme stochastique
delta = 0.99 # Facteur de décroissance de alpha (baisse de température)
sigma = 0.1 # Écart-type pour la distribution normale du bruit
num_iterations = 300 # Nombre total d'itérations

# -----
# Création d'une matrice de synergie S
# -----
# Pour simplifier, nous supposons que S(i,j) est constant et uniforme
```

```

# sur le réseau, par exemple  $S(i,j)=0.8$  pour  $i \neq j$ .
S = np.full((n, n), 0.8)
np.fill_diagonal(S, 0) # Pas de self-synergie

# -----
# Initialisation des pondérations  $\omega$ 
# -----
omega = np.random.uniform(0, 0.05, (n, n))
np.fill_diagonal(omega, 0)

# Stockage de l'historique pour la visualisation
omega_history = np.zeros((num_iterations + 1, n, n))
omega_history[0] = omega.copy()

# Initialisation de alpha
alpha = initial_alpha
alpha_history = [alpha]

# -----
# Simulation de la mise à jour avec terme stochastique
# -----
for t in range(num_iterations):
    # Mise à jour alpha (recuit simulé)
    alpha *= delta
    alpha_history.append(alpha)

    # Mise à jour de  $\omega$  pour chaque paire (i,j)
    omega_next = omega.copy()
    for i in range(n):
        for j in range(n):
            if i != j:
                # Calcul déterministe de la mise à jour additive
                deterministic_update = eta * (S[i, j] - tau * omega[i, j])
                # Terme stochastique: bruit tiré de  $N(0, \sigma^2)$ 
                stochastic_term = alpha * np.random.normal(0, sigma)
                # Mise à jour totale
                omega_next[i, j] = omega[i, j] + deterministic_update + stochastic_term
                # Option de clipping pour éviter des valeurs négatives
                if omega_next[i, j] < 0:
                    omega_next[i, j] = 0
    # Mise à jour de la matrice pour la prochaine itération
    omega = omega_next.copy()
    omega_history[t+1] = omega.copy()

# -----
# Visualisation des résultats
# -----
# Exemple de visualisation : moyenne des pondérations au fil du temps
mean_omega = [np.mean(omega_history[t][np.triu_indices(n, k=1)]) for t in range(num_iterations+1)]
plt.figure(figsize=(12, 6))
plt.plot(mean_omega, label="Pondération moyenne")
plt.xlabel("Itérations")
plt.ylabel("Valeur moyenne de  $\omega$  (hors diagonale)")
plt.title("Évolution de la pondération moyenne avec terme stochastique")
plt.legend()
plt.grid(True)

```

```
plt.show()

# Visualisation d'une heatmap de la matrice finale  $\omega$ 
plt.figure(figsize=(8, 6))
sns.heatmap(omega_history[-1], cmap="viridis", annot=False)
plt.title("Heatmap de la matrice  $\omega$  finale")
plt.xlabel("Index j")
plt.ylabel("Index i")
plt.show()

# Visualisation de l'évolution de alpha
plt.figure(figsize=(12, 4))
plt.plot(alpha_history, label="α (Amplitude du bruit)")
plt.xlabel("Itérations")
plt.ylabel("α")
plt.title("Diminution du paramètre α au fil des itérations")
plt.legend()
plt.grid(True)
plt.show()
```

Explications Complètes

Principe et Formulation

La mise à jour déterministe initiale est donnée par

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)].$$

Pour introduire un **terme stochastique**, nous ajoutons une perturbation $\alpha \xi_{i,j}(t)$ où $\xi_{i,j}(t)$ est une variable aléatoire suivant une distribution normale centrée $\mathcal{N}(0, \sigma^2)$ (vous pouvez également utiliser une distribution uniforme). Le paramètre α contrôle l'amplitude du bruit. Ainsi, la mise à jour devient :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] + \alpha \xi_{i,j}(t).$$

En outre, pour favoriser une phase d'exploration initiale, on peut permettre à α de décroître au fil des itérations selon :

$$\alpha(t+1) = \delta \alpha(t) \quad \text{avec} \quad 0 < \delta < 1.$$

Impact sur la Dynamique

L'ajout du terme stochastique permet au réseau de s'extraire des minima locaux en introduisant une variabilité qui, dans certains cas, peut mener à un réaménagement des clusters. La dynamique ainsi obtenue est alors celle d'un **processus stochastique** qui converge non pas vers un unique point fixe, mais vers une distribution autour de ce point, dont la variance dépend de α et σ .

Lorsque le bruit est élevé, le système explore un plus grand nombre de configurations avant de se stabiliser. À mesure que α décroît (baisse de « température »), le réseau se stabilise progressivement, permettant la formation de clusters plus robustes et adaptés à la variabilité des données.

Implémentation et Visualisations

Le code Python ci-dessus simule la mise à jour des pondérations $\omega_{i,j}$ dans un SCN avec injection de bruit. Nous utilisons des bibliothèques standards telles que **NumPy** pour le calcul matriciel et **Matplotlib** ainsi que **Seaborn** pour la visualisation.

- La **fonction de simulation** initialise une matrice ω avec de faibles valeurs aléatoires, puis met à jour cette matrice selon la règle stochastique.
- À chaque itération, le paramètre α est multiplié par δ pour simuler le recuit simulé.
- La mise à jour est effectuée pour chaque paire (i, j) (hors diagonale) en ajoutant le terme déterministe et le terme stochastique.
- Finalement, plusieurs visualisations sont proposées : l'évolution de la pondération moyenne, la heatmap de la matrice finale, et la décroissance de α .

Ces visualisations permettent d'observer comment le bruit influence la convergence des liens et la formation de clusters, offrant ainsi une compréhension concrète de l'impact de l'**ajout d'un terme stochastique** dans la dynamique d'un SCN.

Conclusion

L'ajout d'un **terme stochastique** dans la mise à jour des pondérations, via la formule

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] + \alpha \xi_{i,j}(t),$$

permet de surmonter le risque de figement dans des minima locaux et favorise l'exploration de nouvelles configurations. Le paramètre α (avec une décroissance programmée) joue un rôle crucial en équilibrant l'exploration initiale et la stabilisation ultérieure. L'implémentation en Python ci-dessus, accompagnée de graphiques explicatifs, illustre concrètement comment cette dynamique peut être réalisée et visualisée dans un SCN. Ce procédé constitue une stratégie efficace pour améliorer la **flexibilité** et la **robustesse** d'un réseau auto-organisé dans des applications variées telles que la robotique multi-agent ou l'apprentissage non supervisé.

5.5.4.2. Diminution Progressive de la "Température" pour Éviter de Rester Piégé dans un Minimum Local

Dans le cadre d'un **Synergistic Connection Network (SCN)**, la mise à jour des pondérations $\omega_{i,j}$ repose sur une dynamique qui, dans sa version déterministe, tend à figer les valeurs autour d'un point fixe. Or, cette stabilité peut être trompeuse puisqu'elle risque de conduire le système à se retrouver bloqué dans un **minimum local** qui ne reflète pas l'optimum global de la synergie. Pour pallier ce problème, il est pertinent d'introduire un **terme stochastique** dans la mise à jour, de manière à favoriser l'**exploration** des configurations alternatives. Afin de contrôler cette exploration, on rend l'amplitude du bruit variable dans le temps en introduisant le concept de "**température**" qui décroît progressivement.

A. Principes du Recuit Simulé et Rôle de la Température

Le principe du **recuit simulé** est inspiré du processus de refroidissement en métallurgie, où un matériau est chauffé à haute température pour permettre aux atomes de se réarranger librement, puis refroidi lentement pour stabiliser une structure ordonnée. Dans le contexte d'un SCN, la mise à jour stochastique des pondérations s'exprime par :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] + \alpha(t) \xi_{i,j}(t),$$

où η est le taux d'apprentissage, τ le coefficient de décroissance, et $\alpha(t)$ représente la **température** à l'itération t . La variable aléatoire $\xi_{i,j}(t)$ est tirée d'une distribution centrée (par exemple, $\xi_{i,j}(t) \sim \mathcal{N}(0, \sigma^2)$). La fonction de température $\alpha(t)$ est conçue pour décroître au fil du temps afin de favoriser une phase initiale d'exploration, suivie d'une phase de stabilisation. Une loi exponentielle de décroissance peut être utilisée, par exemple :

$$\alpha(t+1) = \delta \alpha(t) \quad \text{avec} \quad 0 < \delta < 1.$$

Ainsi, au début, lorsque $\alpha(t)$ est élevé, le bruit introduit des fluctuations importantes qui permettent au SCN d'explorer un vaste espace de solutions et d'échapper à des minima locaux peu profonds. Au fur et à mesure que $\alpha(t)$ décroît, les fluctuations diminuent, ce qui permet au réseau de se stabiliser autour d'une configuration optimale.

B. Impact sur la Dynamique du Système

La dynamique globale devient alors celle d'un **processus stochastique non autonome** où la mise à jour des pondérations est influencée par un bruit dont l'amplitude évolue avec t . En l'absence de bruit ($\alpha = 0$), le système converge généralement vers le point fixe déterministe :

$$\omega_{i,j}^* \approx \frac{S(i,j)}{\tau}.$$

Avec le terme stochastique, la suite $\{\omega_{i,j}(t)\}$ ne converge pas vers une valeur unique, mais plutôt vers une **distribution** centrée autour de ce point fixe. La variance de cette distribution dépend de la magnitude de $\alpha(t)$ et de σ . Le processus de décroissance de $\alpha(t)$ assure qu'après une phase d'exploration, le système « refroidit » et se stabilise progressivement, permettant la formation de **clusters** robustes et la convergence vers une configuration qui maximise la synergie globale.

C. Implémentation Python Complète avec Visualisation

Le code Python suivant simule cette dynamique dans un SCN en utilisant la mise à jour stochastique avec recuit simulé. Nous générons également plusieurs graphiques pour illustrer l'évolution des pondérations, la décroissance de la température, et une heatmap finale de la matrice ω .

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Fixer une graine pour la reproductibilité
np.random.seed(42)

# Paramètres de la simulation
n = 30 # nombre d'entités (pour la simulation dense)
eta = 0.05 # taux d'apprentissage
tau = 1.0 # coefficient de décroissance
alpha0 = 0.2 # température initiale (amplitude du bruit)
delta = 0.98 # facteur de décroissance de la température (0 < delta < 1)
sigma = 0.1 # écart-type du bruit (pour N(0, sigma^2))
iterations = 300 # nombre d'itérations

# Création d'une matrice de synergie S (par exemple, S(i,j)=0.8 pour i != j)
S = np.full((n, n), 0.8)
np.fill_diagonal(S, 0) # pas de synergie sur la diagonale

# Initialisation de la matrice des pondérations omega
omega = np.random.uniform(0, 0.05, (n, n))
np.fill_diagonal(omega, 0)

# Stocker l'évolution de omega et de la température
omega_history = np.zeros((iterations + 1, n, n))
omega_history[0] = omega.copy()
alpha_history = [alpha0]

# Température initiale
alpha = alpha0
```



```

# Simulation de la dynamique avec recuit simulé
for t in range(iterations):
    # Mise à jour de la température (recuit simulé)
    alpha *= delta
    alpha_history.append(alpha)

    # Initialiser une nouvelle matrice pour  $\omega(t+1)$  (double-buffer)
    omega_next = omega.copy()
    for i in range(n):
        for j in range(n):
            if i != j:
                # Calcul déterministe de la mise à jour additive
                delta_det = eta * (S[i, j] - tau * omega[i, j])
                # Génération du bruit aléatoire ( $N(0, \sigma^2)$ )
                noise = alpha * np.random.normal(0, sigma)
                # Mise à jour complète
                omega_next[i, j] = omega[i, j] + delta_det + noise
                # Option de clipping pour éviter des valeurs négatives
                if omega_next[i, j] < 0:
                    omega_next[i, j] = 0
    # Passage à l'itération suivante
    omega = omega_next.copy()
    omega_history[t+1] = omega.copy()

# -----
# Visualisations
# -----

# 1. Évolution de la température alpha au fil des itérations
plt.figure(figsize=(10, 4))
plt.plot(alpha_history, color='blue', lw=2)
plt.xlabel("Itérations")
plt.ylabel("Température ( $\alpha$ )")
plt.title("Diminution Progressive de la Température")
plt.grid(True)
plt.show()

# 2. Évolution de la pondération moyenne (hors diagonale) au fil des itérations
mean_omega = [np.mean(omega_history[t][np.triu_indices(n, k=1)]) for t in range(iterations+1)]
plt.figure(figsize=(10, 4))
plt.plot(mean_omega, color='green', lw=2)
plt.xlabel("Itérations")
plt.ylabel("Pondération moyenne ( $\omega$ )")
plt.title("Évolution de la Pondération Moyenne dans le SCN")
plt.grid(True)
plt.show()

# 3. Heatmap de la matrice  $\omega$  à la fin de la simulation
plt.figure(figsize=(8, 6))
sns.heatmap(omega_history[-1], cmap="viridis", annot=False)
plt.title("Heatmap Finale de la Matrice  $\omega$ ")
plt.xlabel("Index j")
plt.ylabel("Index i")
plt.show()

# 4. Visualisation de l'évolution de quelques liens spécifiques

```

```

# Choisissons 5 liens aléatoires pour suivre leur évolution
num_links_to_plot = 5
links = []
while len(links) < num_links_to_plot:
    i, j = np.random.randint(0, n, 2)
    if i != j and (i, j) not in links:
        links.append((i, j))

plt.figure(figsize=(12, 6))
for (i, j) in links:
    link_values = [omega_history[t][i, j] for t in range(iterations+1)]
    plt.plot(link_values, label=f" $\omega(\{i\}, \{j\})$ ")
plt.xlabel("Itérations")
plt.ylabel("Valeur de  $\omega$ ")
plt.title("Évolution des Pondérations de Quelques Liens Sélectionnés")
plt.legend()
plt.grid(True)
plt.show()

```

Explications et Analyse

Principe et Formulation

La mise à jour stochastique des pondérations dans un SCN s'exprime par la formule :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] + \alpha(t) \xi_{i,j}(t),$$

où $\xi_{i,j}(t)$ est une variable aléatoire centrée, typiquement distribuée selon $\mathcal{N}(0, \sigma^2)$. La **température** $\alpha(t)$ décroît selon une loi exponentielle définie par

$$\alpha(t+1) = \delta \alpha(t),$$

ce qui signifie que le bruit est fort en début de simulation (favorisant l'exploration) et diminue progressivement pour stabiliser la configuration finale.

Impact sur la Dynamique

Grâce à ce mécanisme de recuit simulé, le système peut sortir des minima locaux en autorisant temporairement des fluctuations importantes dans les pondérations. La dynamique converge ensuite vers une distribution autour de la valeur théorique $\frac{S(i,j)}{\tau}$, avec une variance qui diminue au fil du temps lorsque $\alpha(t)$ tend vers zéro. Ce procédé permet d'obtenir des **clusters** plus robustes et mieux adaptés aux données.

Implémentation et Visualisations

Le code Python présenté simule la mise à jour itérative de la matrice ω dans un SCN, en intégrant un terme stochastique dont l'amplitude diminue progressivement. Plusieurs graphiques sont générés pour visualiser :

- La décroissance de la température α ,
- L'évolution de la pondération moyenne (hors diagonale),
- Une heatmap de la matrice ω à la fin de la simulation,
- L'évolution temporelle de quelques liens spécifiques.

Ces visualisations permettent d'observer comment le **recuit simulé** aide le système à explorer l'espace des configurations dans un premier temps, puis à se stabiliser pour former des clusters cohérents.

Conclusion

L'introduction d'un **terme stochastique** avec une **température décroissante** (recuit simulé) dans la mise à jour des pondérations permet de surmonter le risque de piégeage dans des minima locaux, en favorisant l'exploration initiale avant de stabiliser la dynamique du SCN. La formule :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)] + \alpha(t) \xi_{i,j}(t)$$

associe ainsi la correction déterministe à une perturbation contrôlée, dont l'amplitude $\alpha(t)$ décroît progressivement. L'implémentation Python fournie illustre de manière complète ce mécanisme, et les visualisations permettent de suivre la diminution progressive de la température, l'évolution de la moyenne des pondérations, et la formation d'une structure finale dans le SCN. Ce procédé offre une stratégie efficace pour améliorer la **flexibilité** et la **robustesse** des réseaux auto-organisés dans divers domaines d'application.

5.6. Interfaces Entrée-Sortie et Gestion en Temps Réel

5.6.1. Ajouter / Retirer une Entité

- 5.6.1.1. $\text{addEntity}(\mathcal{E}_i)$: initialisation des liens $\omega_{i,\dots}$.
- 5.6.1.2. $\text{removeEntity}(\mathcal{E}_i)$: suppression de ses liens, mise à jour de la structure.

5.6.2. Mise à Jour Périodique vs. Événementielle

- 5.6.2.1. Stratégie batch : attend un certain nombre d'événements avant un recalcul massif.
- 5.6.2.2. Stratégie streaming : mise à jour locale itérative.

5.6.3. Extraction de Clusters / Sous-Réseaux

- 5.6.3.1. $\text{getTopLinks}(i, k)$, $\text{getAllClusters}(\theta)$.
- 5.6.3.2. Usage externe (visualisation, classification), design des formats de sortie (JSON, CSV, etc.).

5.6. Interfaces Entrée-Sortie et Gestion en Temps Réel

Dans l’environnement d’un **SCN** (Synergistic Connection Network), la **gestion** des entités en temps réel revêt une importance cruciale : il n’est pas rare qu’au cours de la simulation ou d’une application concrète, on doive **ajouter** de nouvelles entités (apparition de nouveaux objets, agents, données) ou **supprimer** des entités devenues obsolètes ou inutiles. Dans la même veine, il peut s’avérer nécessaire d’opérer des mises à jour non pas selon un calendrier fixe (batch), mais en mode “flux” (streaming), de sorte à s’adapter rapidement à des changements fréquents. La section (5.6) traite ainsi des **interfaces** d’entrée-sortie qui permettent de gérer le SCN “en direct”, et explique comment ajouter ou retirer des entités (5.6.1), quelles stratégies de mise à jour choisir (périodique ou événementielle, 5.6.2), et enfin comment extraire ou visualiser des **clusters** à la volée (5.6.3).

5.6.1. Ajouter / Retirer une Entité

L’une des fonctionnalités clés d’une interface temps réel est la possibilité de **modifier** dynamiquement l’ensemble $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ que le SCN prend en charge. Nous décrivons ici les opérations fondamentales `addEntity` et `removeEntity`, indispensables pour maintenir la cohérence de la structure ω .

5.6.1.1. `addEntity(\mathcal{E}_i)` : Initialisation des Liens $\omega_{i,\dots}$

L’opération `addEntity(\mathcal{E}_i)` permet d’incorporer une *nouvelle entité* \mathcal{E}_i dans un SCN (Synergistic Connection Network) en cours de fonctionnement. Son importance se justifie par la nécessité, dans de nombreux contextes pratiques, d’intégrer des données apparaissant dynamiquement : un flux multimédia de grande ampleur, l’arrivée d’utilisateurs dans un réseau social ou la mise en service d’agents supplémentaires dans un système robotique. Sur le plan **mathématique**, l’ajout d’une entité modifie la dimension de la matrice ω qui passe de $(n \times n)$ à $((n + 1) \times (n + 1))$, et requiert une procédure d’initialisation des **liaisons** $\omega_{i,j}$ associées au nouvel index i .

Dans la plupart des applications, on choisit de **réserver** un *identifiant* supplémentaire — souvent $i = n + 1$ — et de créer les lignes et colonnes nécessaires. Les connexions $\omega_{i,j}$ et $\omega_{j,i}$ sont alors insérées avec des valeurs nulles ou très faibles, comme

$$\omega_{i,j}(0) = 0, \quad \omega_{j,i}(0) = 0, \quad \forall j \in \{1, \dots, n\}.$$

Cette option garantit que le nouvel élément \mathcal{E}_i ne perturbe pas immédiatement la structure du réseau et autorise la **dynamique** du DSL à faire croître ou décroître ses liaisons de manière autonome selon la règle

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

ou selon la version intégrant des mécanismes d’**inhibition** ou de **bruit**. Dans certains schémas plus rapides, il est concevable de fixer

$$\omega_{i,j}(0) \propto S(i,j) \quad \text{et} \quad \omega_{j,i}(0) \propto S(j,i),$$

afin d’accorder une forme de “démarrage accéléré” à \mathcal{E}_i en exploitant l’information de synergie dès son arrivée.

Du point de vue **ingénierie**, cette démarche s’accompagne souvent de l’allocation nécessaire en mémoire pour stocker les poids $\omega_{i,\dots}$. Dans le cas d’un **SCN** dense, cela implique de redimensionner la matrice, tandis que dans une implémentation **sparse**, il suffit d’ajouter un *champ* ou une *structure* associée au nouvel identifiant i . Il est aussi crucial de gérer les index pour que chaque module du DSL (mise à jour, synergie, parsimonie) reconnaisse la nouvelle entité. L’insertion d’une *API* claire pour la fonction `addEntity(e)` assure la cohérence entre la phase d’ajout et la poursuite de l’apprentissage local.

Sur le plan **mathématique**, cette insertion d’entités à la volée modifie le caractère du système, passant d’une évolution dans un espace de dimension n^2 à un espace de dimension $(n + 1)^2$. La topologie du **SCN** s’en trouve enrichie : dans les itérations ultérieures, le nouveau nœud \mathcal{E}_i noue des liens selon

$$\Delta \omega_{i,j}(t) = \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

ou dans les variantes multiplicatives ou inhibées, sans avoir besoin de réinitialiser l’ensemble du réseau. Cette propriété est essentielle lorsque l’on traite un **flux continu** : chaque arrivée d’information préserve la structure accumulée jusque-là, tout en offrant à la nouvelle entité la possibilité de s’insérer de façon fluide dans les *clusters* existants ou d’en faire émerger de nouveaux.

Conclusion

La fonction $\text{addEntity}(\mathcal{E}_i)$ illustre la capacité d’un SCN à *croître* de manière dynamique, facteur déterminant pour les applications où l’on ne peut se contenter d’une architecture fixe. En initialisant les liens à des valeurs nulles ou faibles, on mise sur la **règle de mise à jour** locale pour ajuster progressivement $\omega_{i,j}$ en accord avec la fonction de **synergie**. L’implémentation doit nécessairement gérer l’allocation et la synchronisation des données, laissant à la **dynamique** du DSL le soin de décider des connexions à consolider ou à supprimer. Une telle approche rend le système “vivant” et adaptable, au sens où l’insertion d’une nouvelle entité ne remet pas en question les clusters déjà formés, mais offre plutôt au réseau la possibilité d’acquérir de nouveaux liens ou de se réorganiser pour incorporer la *vitalité* apportée par \mathcal{E}_i .

5.6.1.2. $\text{removeEntity}(\mathcal{E}_i)$: Suppression de ses Liens et Mise à Jour de la Structure

Dans un **SCN** (Synergistic Connection Network) dynamique, la possibilité de retirer une entité déjà présente est aussi cruciale que l’ajout abordé en **Section 5.6.1.1**. Les raisons de cette suppression sont multiples : données devenues caduques, agent quittant un système multi-agent, ou simple volonté d’alléger la structure pour des motifs de performance. Le point essentiel est de gérer la disparition de \mathcal{E}_i de façon cohérente, afin d’éviter toute corruption ou incohérence dans les indices et dans la mise à jour des liaisons ω .

A. Motivations et Cadre Mathématique

La suppression d’une entité \mathcal{E}_i dans un SCN implique de **réduire** la dimension de l’espace des connexions. Lorsque la matrice ω était de taille $(n \times n)$, le retrait d’une entité transforme cet espace en $(n - 1) \times (n - 1)$, à moins de conserver les emplacements vides pour ne pas réindexer. Du point de vue des règles d’**auto-organisation** (voir **Section 5.5** sur la mise à jour), toute entité disparue ne doit plus influencer sur la synergie ou la dynamique de **clusters**. Concrètement, les liaisons $\omega_{i,\cdot}$ et $\omega_{\cdot,i}$ doivent cesser d’exister, et la matrice ω n’a plus à entretenir la structure associée à l’index i .

Les circonstances menant à une telle suppression sont variées. Dans un réseau temps réel, une **donnée** peut n’être plus valide après un délai, ou bien se révéler obsolète au sens où sa synergie $\{S(i,j)\}$ n’est plus calculée. Dans un réseau social ou un système multi-agent, un **utilisateur** ou un **robot** peut quitter la scène et libérer les ressources y afférentes. Sans suppression, des lignes et colonnes fantômes peuvent se maintenir, produisant un bruit inutile et encombrant la topologie du SCN.

B. Opération de Suppression et Conséquences sur la Structure

La commande $\text{removeEntity}(\mathcal{E}_i)$ doit éliminer toutes les références à l’entité \mathcal{E}_i dans l’ensemble des liaisons. Cela équivaut à annuler ou à effacer les *lignes* $\omega_{i,j}$ pour tout j et les *colonnes* $\omega_{j,i}$ pour tout j . L’implémentation diffère selon que la matrice ω est en format **dense** ou **sparse**. Dans un format dense, on peut choisir de mettre à zéro tous les $\omega_{i,j}$ et $\omega_{j,i}$, quitte à laisser une ligne morte, ou alors reconstruire une matrice $(n - 1) \times (n - 1)$ en réindexant toutes les entités. Dans un format sparse (hashmaps ou listes d’adjacence), il suffit de balayer les entrées associées à l’index i et de les supprimer.

Cette manipulation a un impact direct sur la **dynamique** du SCN : toute entité \mathcal{E}_k qui présentait des liens $\omega_{k,i}$ non négligeables perd désormais ce canal de synergie et peut se réorganiser dans les itérations ultérieures. Du point de vue mathématique, on retire un ensemble de degrés de liberté, et la dynamique locale

$$\omega_{k,j}(t+1) = \omega_{k,j}(t) + \eta[S(k,j) - \tau \omega_{k,j}(t)]$$

se poursuit en l'absence de l'index i . Dans le cas d'une **inhibition compétitive**, la somme $\sum_m \omega_{k,m}$ peut également s'en trouver modifiée, libérant ainsi de la "place" pour d'autres connexions.

C. Cohérence Logique et Implantation dans l'API

Une **API** clairement définie, par exemple *removeEntity(e)*, assure la cohérence de l'opération. Cette fonction :

4. Identifie l'entité \mathcal{E}_i (via son ID).
5. Supprime ou met à zéro l'ensemble $\omega_{i,j}$ et $\omega_{j,i}$.
6. Informe éventuellement les autres modules (mise à jour, synergie, monitoring) que \mathcal{E}_i n'est plus active.

Si le SCN fonctionne en mode **temps réel**, il peut être préférable de gérer cette suppression en fin d'itération, afin de ne pas altérer les boucles de mise à jour en cours. Dans un **cadre distribué** (voir **Section 5.2.3.3**), il est également important d'annoncer la disparition aux autres nœuds afin qu'ils ne continuent pas de calculer $\omega_{i,j}$ pour des valeurs qui ne sont plus censées exister.

Les considérations de **performance** et de **stabilité** justifient l'implémentation d'une telle opération. Sur le plan de la **complexité**, maintenir des entités obsolètes génère un surcoût systématique, tant en calcul de synergie $\{S(i,j)\}$ qu'en stockage de ω . Sur le plan de la **stabilité**, la persistance d'entités inactives peut orienter les mises à jour vers de fausses configurations ou influencer négativement les clusters.

D. Conséquences Dynamiques et Réorganisation de Clusters

La disparition d'une entité \mathcal{E}_i peut entraîner une **réorganisation** notable si \mathcal{E}_i occupait un rôle clé (par exemple, un "hub" au sein d'un cluster). Les entités \mathcal{E}_k qui appuyaient leur synergie sur la présence de \mathcal{E}_i vont connaître un nouvel équilibre. Sur le plan purement mathématique, les règles d'**update** (cf. **Section 5.5**) s'appliquent désormais sur l'espace $(n-1)$. Les pondérations associées à d'autres liens peuvent se renforcer ou diminuer afin de compenser la perte de contribution de $\omega_{k,i}$.

Si l'on considère un SCN avec règles d'**inhibition** (voir **Section 5.5.2**) qui contraint $\sum_j \omega_{k,j} \leq \Omega_{\max}$, la suppression de \mathcal{E}_i libère un potentiel de croissance pour d'autres liaisons de \mathcal{E}_k , modifiant parfois la morphologie des clusters existants.

Conclusion

La fonction *removeEntity(\mathcal{E}_i)*, symétrique de *addEntity(\mathcal{E}_i)* présentée en **Section 5.6.1.1**, assure la **souplesse** d'un SCN évolutif. En retirant proprement toutes les liaisons associées à \mathcal{E}_i , on évite toute confusion dans la dynamique locale, on prévient le gaspillage de ressources et on préserve la cohérence mathématique de la règle de mise à jour. La dimension du système se trouve ainsi ajustée de manière flexible, permettant un fonctionnement *continu* même dans un flux de données ou d'entités changeant. L'opération de suppression, si elle est correctement orchestrée, aboutit à un réseau plus **robuste** et mieux adapté à la réalité des applications, qu'il s'agisse de flux de données éphémères ou d'agents intermittents dans un système distribué.

5.6.2. Mise à Jour Périodique vs. Événementielle

Au sein d'un SCN (Synergistic Connection Network) gérant des données ou des agents en **temps réel**, on peut se demander **comment** orchestrer la dynamique de mise à jour des pondérations ω lorsque l'environnement évolue (ajout/suppression d'entités, arrivée de nouvelles informations, etc.). Deux stratégies s'opposent en général : la **mise à**

jour par batch (périodique) et la **mise à jour événementielle** (streaming). Tandis que la première attend qu'un certain nombre d'événements se soit accumulé avant de relancer un "recalcul" global, la seconde ajuste les pondérations localement à chaque événement ou de manière quasi continue.

5.6.2.1. Stratégie Batch : Attendre un Certain Nombre d'Événements avant un Recalcul Massif

La **stratégie batch**, également qualifiée de stratégie **périodique**, consiste à ne pas actualiser en continu le *Synergistic Connection Network* (SCN) au gré de chaque arrivée ou départ d'entité, ni dès la moindre fluctuation de flux. Il s'agit plutôt de définir des intervalles, exprimés en nombre d'événements ou en durée temporelle, au terme desquels on procède à une mise à jour globale et plus approfondie de la structure des liens ω . Cette approche présente l'intérêt de limiter les recomputations constantes, ce qui s'avère essentiel lorsque le nombre d'entités n peut devenir très grand et que la fonction de **synergie** $\{S(i, j)\}$ nécessite un calcul coûteux.

Dans le principe, la méthode batch repose sur l'idée qu'une suite de petits changements, intervenant de manière rapprochée, ne justifie pas forcément un recalcul complet des liaisons $\omega_{i,j}(t)$. On se contente alors de maintenir les règles d'**update** (cf. **Section 5.5**) sans intégrer immédiatement les nouvelles entités ou les entités supprimées. Les modifications sont simplement enregistrées dans une liste d'attente ou dans un buffer d'événements. Dès lors qu'un certain nombre de changements a été atteint, ou que l'on arrive à un instant prédéfini Δt dans l'horloge système, on exécute le "batch" : on ajoute ou on retire toutes les entités différées, on met à jour la matrice ω ou la structure sparse correspondante, et l'on recalcule si nécessaire le *score* de synergie pour ces nouveaux nœuds.

Sur le plan **mathématique**, on peut formaliser la stratégie batch par un compteur d'événements κ . À chaque ajout ou suppression d'entité, on incrémente κ . Tant que $\kappa < \kappa_{\max}$, on n'altère pas la topologie du réseau ; on laisse la dynamique $\{\omega_{i,j}(t)\}$ se dérouler comme si rien n'avait changé, selon la règle habituelle

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)],$$

ou selon l'une de ses variantes (inhibition compétitive, mécanisme multiplicatif, présence de bruit). Lorsque $\kappa \geq \kappa_{\max}$, on applique en bloc les opérations différées et on remet κ à zéro. De façon similaire, dans un cadre temporel, on peut définir un pas de temps Δt et n'autoriser le recalcul qu'aux instants multiples de Δt .

Ce procédé présente un avantage d'**économie de calcul** : pour un système volumineux ou hautement dynamique, il est souvent plus efficient d'effectuer une *mise à jour massive* de temps à autre plutôt qu'un ajustement permanent à chaque échantillon. Il offre également une forme de **stabilité** temporelle, dans la mesure où la structure du SCN ne se transforme pas incessamment et reste intelligible entre deux temps de recalcul. En revanche, cette approche implique une **perte de réactivité** : si un événement critique se produit, il ne sera intégré qu'au prochain lot, laissant un décalage entre la réalité et l'état du réseau.

Un exemple concret se rencontre dans un système de **recommandation** en ligne, où des utilisateurs et des produits apparaissent constamment. Une intégration *événement par événement* exigerait de réapprendre la topologie $\{\omega_{i,j}\}$ à chaque nouvel utilisateur, ce qui se révélerait onéreux. En adoptant une stratégie batch, on peut décider de réactualiser le SCN toutes les 12 heures ou lorsque 100 nouveaux comptes sont créés. Entre ces deux recalculs, l'infrastructure de recommandation applique la dynamique existante pour affiner ω , sans devoir gérer en permanence l'arrivée de nouvelles entités.

Sur le plan de l'**implémentation** logicielle, on maintient généralement un **registre** des modifications. Chaque fois qu'une entité est ajoutée ou supprimée (cf. **Sections 5.6.1.1** et **5.6.1.2**), on ne touche pas immédiatement à la matrice ω . On incrémente un compteur ou on note la date de la dernière modification. Puis, lorsque le quota est atteint ou que l'horloge a franchi un seuil, on applique de façon *synchronisée* la mise à jour : on relance éventuellement un calcul de synergie $\{S(i, j)\}$ pour les nouveaux nœuds, on retire ceux qui sont devenus obsolètes, et on solde toutes les modifications avant de reprendre le cours normal de la dynamique locale.

En matière de **comparaison** avec la stratégie *événementielle* (voir **Section 5.6.2.2**), le choix dépend largement de la nature du flux de données et du coût de recalcul. La stratégie batch favorise la **robustesse** et la réduction du **bruit** dans le SCN, au détriment de la *fraîcheur* de l'information. Elle prévient aussi de possibles oscillations liées à des micro-événements successifs. La stratégie événementielle, en revanche, intègre chaque changement quasiment en temps réel,

ce qui convient mieux à des applications de haute réactivité, mais peut alourdir la charge de calcul si les insertions et suppressions sont très fréquentes.

En conclusion, la **stratégie batch** offre une solution pragmatique pour les grandes architectures : plutôt que de mobiliser des ressources de calcul constamment, on regroupe les évolutions sur une fenêtre donnée, on les applique simultanément, puis on laisse la *dynamique* du SCN s'ajuster paisiblement jusqu'à la prochaine vague de modifications. Cette gestion par lots est par conséquent courante dans les systèmes qui tolèrent un léger retard entre la détection d'un événement et sa répercussion effective sur la structure du SCN, mais qui souhaitent maintenir un fonctionnement maîtrisé, à faible bruit, et bien échelonné dans le temps.

5.6.2.2. Stratégie Streaming : Mise à Jour Locale Itérative

La stratégie dite “streaming” se propose de mettre à jour le *Synergistic Connection Network* (SCN) de manière continue et événementielle, plutôt que de recourir à des “lots” ou *batches* espacés dans le temps. Cette approche se caractérise par une réactivité immédiate face à chaque événement (arrivée ou départ d'entité, flux de données, etc.), et privilégie un ajustement *incrémental* plutôt qu'un recalcul massif et périodique. Elle se distingue ainsi de la logique présentée en **Section 5.6.2.1**, où les modifications sont accumulées puis appliquées en bloc.

Dans la stratégie streaming, on considère que toute perturbation du système doit être répercutée aussitôt sur la structure de poids $\{\omega_{i,j}\}$. Cela signifie qu'à chaque fois qu'une entité \mathcal{E}_{n+1} est ajoutée, le SCN se dote immédiatement de la ligne et de la colonne correspondantes. Dans une implémentation dense, la matrice ω est agrandie de $(n+1) \times (n+1)$. Dans un format plus économe, on crée la structure sparse nécessaire pour référencer $\omega_{n+1,j}$ et $\omega_{j,n+1}$. À l'inverse, si une entité \mathcal{E}_i disparaît, alors les lignes et colonnes associées sont supprimées (voir **Sections 5.6.1.1** et **5.6.1.2**).

L'un des atouts majeurs de ce mode “événementiel” tient à sa **réactivité**. Dans un contexte où les données évoluent rapidement, comme dans un flux sensoriel temps réel ou un réseau d'agents qui se connectent et se déconnectent sans cesse, une latence trop élevée peut nuire aux performances globales. La mise à jour locale itérative assure que le SCN reflète toujours l'état courant du système, ce qui peut être déterminant pour les applications critiques (surveillance, détection de fraude, coordination robotique). Mathématiquement, au moment où se produit l'ajout de \mathcal{E}_{n+1} , on initialise $\omega_{n+1,j} \approx 0$ (ou à une valeur proportionnelle à $S(n+1, j)$) puis on laisse la règle DSL classique

$$\omega_{n+1,j}(t+1) = \omega_{n+1,j}(t) + \eta [S(n+1, j) - \tau \omega_{n+1,j}(t)]$$

prendre le relais lors des itérations suivantes, sans attendre le déclenchement d'un gros *batch*.

Cependant, cette extrême proximité avec l'évolution du système n'est pas dénuée de difficultés. Dès lors que la fréquence des événements devient très élevée (centaines ou milliers de changements par minute), chaque mise à jour locale représente un travail algorithmique pouvant se cumuler de façon importante. Il faut donc concevoir une mécanique de mise à jour suffisamment efficace pour qu'à chaque arrivée (ou suppression) d'entité, on ne se retrouve pas à parcourir l'entièreté du réseau $\{\omega_{i,j}\}$. L'idée est généralement de s'en tenir à un *voisinage local*, par exemple les entités présentant une synergie initiale importante, afin d'éviter de manipuler systématiquement les $O(n^2)$ liaisons.

La stratégie streaming exige également un certain soin pour préserver la **stabilité** du SCN dans la durée. Si le flux d'événements est permanent, le système ne bénéficie plus vraiment de phases de repos où les liens $\omega_{i,j}(t)$ peuvent converger. Les règles présentées en **Sections 5.5** (mécanismes additifs, multiplicatifs, inhibition compétitive, etc.) doivent donc être calibrées pour limiter les oscillations. Sur le plan mathématique, on se situe dans un cadre dynamique *non autonome* où la taille et le contenu du réseau changent en continu, ce qui peut rendre la démonstration de convergence plus délicate.

Enfin, la distribution du SCN sur un ensemble de nœuds physiques (voir **Section 5.2.3.3**) accroît la complexité de cette approche, dans la mesure où chaque arrivée d'entité nécessite une propagation asynchrone de l'événement à tous les sous-ensembles où la synergie ou la mise à jour locale doit être prise en compte. Dans un environnement hautement distribué, la charge de synchronisation peut s'avérer très lourde si les événements affluent. Certaines implémentations

optent pour des techniques asynchrones plus avancées, éventuellement conjuguées à des mécanismes de réplication ou de cohérence partielle, de façon à gérer cet afflux sans provoquer un embouteillage sur le réseau de communication.

La stratégie streaming peut être modérée par des approches hybrides, parfois qualifiées de “micro-batch” ou “fenêtre glissante”, qui visent à limiter l’application de la mise à jour à quelques entités à la fois. Par exemple, l’intégration d’une nouvelle entité \mathcal{E}_{n+1} peut être prise en compte immédiatement, mais la réallocation de synergies pour la globalité du SCN peut être différée si l’élément ajouté n’a qu’un impact minime. On se situe alors dans un compromis entre la mise à jour événementielle pure et la mise à jour par lots (batch), laissant la possibilité d’adapter le niveau de réactivité en fonction du *taux de modification*.

En conclusion, la stratégie streaming correspond à un paradigme d’**adaptation continue**, idéal pour les cas où la fraîcheur des informations est prioritaire et où la dynamique doit coller au plus près à un flux évoluant sans relâche. Elle suppose toutefois une implantation soignée, apte à traiter les événements en temps réel sans excéder les capacités de calcul, et un paramétrage approprié (coefficients η , τ , éventuel bruit ou inhibition) pour éviter de perpétuelles oscillations. Elle se révèle ainsi complémentaire à la stratégie batch, présentée en **Section 5.6.2.1**, dans le vaste domaine des SCN dynamiques, où chaque modèle d’actualisation répond à des besoins et des contraintes différents.

5.6.3. Extraction de Clusters / Sous-Réseaux

Même lorsqu’un **SCN** (Synergistic Connection Network) opère en temps réel (sections 5.6.1 et 5.6.2), il demeure essentiel de pouvoir **extraire** ou **observer** en continu la structure émergente : quels liens sont réellement “forts” ? Quels **clusters** ou sous-réseaux se constituent ? Cette section (5.6.3) aborde donc les opérations par lesquelles on récupère des informations sur le SCN pour les exploiter ou les visualiser. Nous nous concentrons d’abord sur deux fonctions couramment rencontrées : $\text{getTopLinks}(i, k)$ et $\text{getAllClusters}(\theta)$ (5.6.3.1), puis sur la manière de mettre ces informations à disposition d’autres systèmes (5.6.3.2).

5.6.3.1. $\text{getTopLinks}(i, k)$, $\text{getAllClusters}(\theta)$

Le **Deep Synergy Learning** amène souvent à gérer un *Synergistic Connection Network* (SCN) où les entités $\{\mathcal{E}_i\}$ sont reliées par une matrice de liens ω . L’**objectif** de ce réseau est la mise en évidence de *clusters* ou sous-réseaux fortement cohérents, toutefois la matrice ω , qu’elle soit dense ou présentée en format sparse, ne fournit pas automatiquement l’information structurée sous forme de communautés. Dans cette optique, les opérations internes $\text{getTopLinks}(i, k)$ et $\text{getAllClusters}(\theta)$ permettent une extraction commode des connexions les plus significatives, qu’il s’agisse d’une analyse locale (pour un nœud donné) ou d’une identification globale de clusters par seuil.

A. Motivation Générale pour l’Extraction de Clusters

L’évolution d’un SCN sous les règles d’auto-organisation (décrites en **Section 5.5**) vise à établir, renforcer ou inhiber des liaisons $\{\omega_{i,j}\}$ en fonction de la **synergie** $S(i, j)$. À l’issue de cette dynamique, on espère discerner des *communautés* ou *composantes* où les liens internes conservent une grande valeur, traduisant un haut degré de similarité ou d’interaction entre les entités. Néanmoins, la matrice ω peut atteindre une taille considérable, et son contenu n’est pas directement lisible si l’on se contente de parcourir l’ensemble $\{\omega_{i,j}\}$. Les fonctions getTopLinks et getAllClusters s’inscrivent donc dans un registre **opérationnel**, conçu pour extraire les liens les plus pertinents et révéler la structure de *clusters* à un instant donné de l’itération.

B. $\text{getTopLinks}(i, k)$: Extraction Locale des Liens Principaux

L’opération $\text{getTopLinks}(i, k)$ fournit, pour une entité \mathcal{E}_i , les k liens $\omega_{i,j}$ de plus forte valeur. L’idée est de rapidement identifier le **voisinage principal** de \mathcal{E}_i , c’est-à-dire les entités \mathcal{E}_j vers lesquelles \mathcal{E}_i entretient la relation la plus intense. Cette notion rappelle la pratique du “k plus proches voisins” (k-NN), à la différence que la mesure de proximité ou d’intérêt est ici fixée par la valeur courante de $\omega_{i,j}$.

Sur le plan **mathématique**, la requête $\text{getTopLinks}(i, k)$ se fonde sur un tri (ou un ordre partiel) des valeurs $\{\omega_{i,1}, \omega_{i,2}, \dots, \omega_{i,n}\}$. Pour $\omega_{i,j}$ susceptible de prendre des valeurs faibles ou nulles, une structure de données adaptée

(section 5.3.2.2 sur les indexations et accès rapides) permet de retrouver les k premières liaisons en un temps réduit, souvent $O(k \log n)$ ou $O(n \log k)$. La liste retournée reflète l'état du SCN à l'itération en cours, et peut aisément changer dans le temps au fur et à mesure que la synergie entre \mathcal{E}_i et les autres entités \mathcal{E}_j évolue.

En **ingénierie** logicielle, $\text{getTopLinks}(i, k)$ est très utile pour la **visualisation locale** : on peut afficher l'entité \mathcal{E}_i et ses k connexions dominantes, et ainsi naviguer dans le SCN de proche en proche. Ce mécanisme convient aussi à des tâches de **recommandation**, où l'on cherche les “voisins” d'un utilisateur \mathcal{E}_i dans un espace de préférences, ou à des algorithmes incrémentaux qui ne souhaitent pas analyser l'intégralité de la matrice ω .

C. $\text{getAllClusters}(\theta)$: Vue Globale par Seuil

L'opération $\text{getAllClusters}(\theta)$ vise à révéler la structure des *clusters* en appliquant un **seuil** θ sur les liaisons $\{\omega_{i,j}\}$. Plus précisément, on considère le sous-graphe dont les arêtes satisfont

$$\omega_{i,j} \geq \theta,$$

tandis que celles situées en dessous de θ sont ignorées. Il en résulte un graphe binaire, où les liens jugés “forts” sont seuls préservés. Sur ce graphe, il devient aisé d'identifier des **composantes connexes** ou des **communautés**, par exemple au moyen d'un parcours DFS/BFS ou d'algorithmes de détection de clusters (type Louvain ou analyse de modularité). Ainsi, l'utilisateur choisit θ pour ajuster la granularité : un seuil proche de 0 rend le graphe très dense, tandis qu'un seuil élevé ne conserve que quelques connexions majeures, formant des *clusters* réduits mais très cohésifs.

Le choix de θ se rattache à la fois à la **distribution** des valeurs $\omega_{i,j}$ et aux intentions d'exploration. On peut également faire varier θ par paliers pour observer les transitions dans la structure de clusterisation (cf. “coupes” successives dans la matrice ω). Sur le plan **mathématique**, $\text{getAllClusters}(\theta)$ décrit un instantané : il se peut que deux appels consécutifs à des itérations différentes t et $t + 10$ n'aboutissent pas au même regroupement, si les liaisons ont significativement changé entre-temps.

D. Intégration dans l'Interface SCN et Avantages

Le fait de **fournir** $\text{getTopLinks}(i, k)$ et $\text{getAllClusters}(\theta)$ de manière native dans un SCN présente plusieurs avantages. Tout d'abord, l'utilisateur ou les modules applicatifs n'ont pas besoin de manipuler la matrice ω dans sa totalité, laquelle peut être massive pour un grand n . Ensuite, si l'implémentation de ω emploie des structures de données **sparse** ou exploite des mécanismes de **parsimonie** (voir Section 5.5.3), la fonction “top- k ” et le filtrage par θ peuvent être réalisés efficacement, sans avoir à traiter toutes les paires (i, j) . Par ailleurs, cette capacité d'extraction **internationale** (en lien direct avec l'état du réseau) permet des mises à jour rapides et cohérentes en fin d'itération, ou même pendant un flux streaming (voir Sections 5.6.2.1 et 5.6.2.2).

Les usages pratiques sont nombreux. Dans des systèmes de **recommandation**, $\text{getTopLinks}(i, k)$ sert à déterminer quelles entités, produits ou contenus sont les plus proches d'un utilisateur \mathcal{E}_i . Dans un **dashboard** de supervision, on peut régulièrement appeler $\text{getAllClusters}(\theta)$ pour dévoiler l'évolution des communautés en temps réel. Dans un cadre purement analytique, ces fonctions constituent la base de l'examen de la structure *multi-échelle* du SCN : si θ décroît, on obtient des regroupements plus vastes ; s'il croît, on isole les noyaux les plus cohésifs.

E. Exemple d'Utilisation : Visualisation Locale et Regroupement Global

Dans la pratique, on peut imaginer un scénario où la commande $\text{getTopLinks}(i, 5)$ est utilisée pour naviguer dans le SCN, à la manière d'un **explorateur** : partant d'une entité \mathcal{E}_i , on obtient ses 5 liens les plus forts, puis on se déplace vers l'un de ces voisins et réitère, formant ainsi un chemin au sein du réseau. Cette visualisation locale évite de charger toutes les relations ω .

Pour la **vue d'ensemble**, on appelle $\text{getAllClusters}(\theta = 0.2)$: on filtre le réseau sur les liens supérieurs ou égaux à 0.2, et l'on détecte ensuite les sous-ensembles fortement connectés (composantes connexes si on considère le graphe non orienté, communautés plus élaborées si on applique un algorithme idoine). On obtient ainsi un partitionnement en *clusters*, qui reflète la structure issue de l'auto-organisation : à bas seuil, les clusters sont gros et moins nets ; à haut seuil, on ne garde que des groupements extrêmement soudés.

Conclusion

Les fonctions $\text{getTopLinks}(i, k)$ et $\text{getAllClusters}(\theta)$ constituent des pièces maîtresses de l’**interface** SCN pour l’extraction de structure. Elles offrent une **passerelle** entre la représentation interne (pondérations $\{\omega_{i,j}\}$) et les besoins concrets de **visualisation**, de **clustering** ou d’**analyse**. Le **k plus forts liens** d’une entité \mathcal{E}_i permet un accès local et ciblé, tandis que la construction d’un graphe seuil θ et l’identification de composantes ou communautés procure une vue globale et hiérarchique des clusters. Par leur intégration directe dans l’API SCN, ces fonctions s’inscrivent dans la logique d’un système de *Deep Synergy Learning* modulable, au service d’usages aussi divers que la recommandation, la classification ou la supervision temps réel.

5.6.3.2. Usage Externe (Visualisation, Classification), Design des Formats de Sortie (JSON, CSV, etc.)

Une fois la structure d’un Synergistic Connection Network (SCN) extraite — par exemple via des outils internes tels que getTopLinks ou getAllClusters présentés en **Section 5.6.3.1** — se pose la question de la mise à disposition de ces informations pour d’autres modules ou d’autres analyses. Dans la plupart des applications, le **réseau** n’existe pas pour lui-même : on souhaite en effet **exploiter** les pondérations ω , les clusters ou la topologie qui en résulte pour des finalités concrètes, comme la **visualisation** en temps réel, la **classification** dans un autre système d’apprentissage machine, ou encore la **détection** de motifs ou d’anomalies. Cette section détaille donc les aspects relatifs au **design** d’interfaces et de formats de sortie, afin de diffuser la structure de clusterisation et les liens du SCN vers l’extérieur.

A. Visualisation et Classification : Motivations et Approches

Dans un grand nombre de cas d’usage, les données structurées par le SCN — qu’il s’agisse de regroupements (clusters) ou de liaisons pondérées en cours d’évolution — doivent être **communiquées** à des composants externes. Il peut s’agir d’une librairie de visualisation, typiquement pour représenter un graphe dynamique, ou d’un pipeline d’apprentissage souhaitant intégrer les *communautés* ou la *mesure de synergie* comme caractéristiques supplémentaires.

7. Visualisation interactive

En ingénierie de **dashboard** ou en contextes de **monitoring**, il est fréquent qu’on veuille afficher la disposition des entités, leurs liens forts et les clusters qu’ils forment. Des bibliothèques comme D3.js, Sigma.js ou Cytoscape nécessitent des **formats** standards (JSON, GraphML, etc.) pour représenter la liste des nœuds et des arêtes. Un composant “SCN” doit donc être en mesure de fournir ces informations, idéalement à la demande (API REST) ou via une mise à jour continue (WebSocket). Cela permet de voir “en direct” quels liens $\omega_{i,j}$ sont hauts et comment les regroupements évoluent à mesure que la dynamique s’applique.

8. Classification et analyse en aval

De plus, le SCN peut servir de **préalable** à la classification d’entités, par exemple en exportant la **structure de cluster** comme une étiquette de “classe” ou de “communauté”. Un autre module de type “machine learning supervisé” récupère alors ces *labels* et les exploite comme feature ou comme partition de référence. De même, des algorithmes de détection d’anomalies peuvent comparer les comportements d’entités supposées être dans le même cluster et repérer d’éventuelles divergences. Les services externes ont donc besoin d’accéder :

- Soit aux **clusters** (chaque entité \mathcal{E}_i étant associée à un cluster \mathcal{C}_i),
- Soit aux **scores** $\omega_{i,j}$ directement, afin de calculer leur propre fonction.

Dans un système évolutif (voir **Section 5.6.2**), il est d’autant plus important d’actualiser ces informations dès que le réseau subit une transformation (arrivées, départs, renforcement de liens). Les modules externes peuvent alors “pull” (requête périodique) ou se faire “push” (notification en temps réel) d’un snapshot mis à jour du SCN.

B. Formats de Sortie : JSON, CSV, GraphML, etc.

L’**export** des données du SCN peut prendre plusieurs formes, en fonction de la taille, de l’usage et des standards adoptés dans l’environnement applicatif. Les formats textuels comme JSON et CSV dominent largement, mais des formats plus spécifiques à la représentation de graphes (GraphML, GEXF) ou des formats binaires peuvent également être requis.

9. JSON

JSON (JavaScript Object Notation) est le plus fréquemment employé, aussi bien pour une exposition **web** (via REST) que pour un échange **machine-to-machine**. On peut par exemple renvoyer un objet JSON structuré comme suit :

```
{
  "nodes": [
    { "id": i, "label": ..., "attributes": {...} },
    ...
  ],
  "links": [
    { "source": i, "target": j, "weight":  $\omega_{i,j}$  },
    ...
  ],
  "clusters": [
    { "clusterId": c_1, "nodes": [i_1, i_2, ...] },
    ...
  ]
}
```

Les bibliothèques de visualisation comme D3.js, Cytoscape.js, etc. comprennent aisément ce type de structure, et il est simple d'y associer des informations graphiques (couleurs, positions, etc.). JSON est aussi facile à générer ou à parser dans la plupart des langages.

10. CSV (Comma-Separated Values)

CSV demeure très populaire pour un usage **off-line** ou pour un import/export rapide vers Excel, pandas ou R. On peut créer :

- Un **fichier d'arêtes** (*edges.csv*) :

```
source,target,weight
i,j, $\omega$ 
i',j', $\omega$ 
...
```

- Un **fichier de clusters** (*clusters.csv*) :

```
node,clusterId
i,c
i',c
...
```

Bien que cette forme tabulaire soit succincte, elle est facilement analysable par une large gamme d'outils. En revanche, elle n'est pas idéale pour coder des structures plus complexes (attributs multiples par lien ou par nœud).

11. GraphML, GEXF

Dans le champ de la **visualisation de graphes** et de la recherche en réseau, des formats comme GraphML ou GEXF (Graph Exchange XML Format) sont couramment utilisés, notamment pour la compatibilité avec des outils comme Gephi, Tulip ou NetworkX. Le SCN peut être converti dans l'un de ces formats afin de bénéficier des algorithmes et interfaces de ces logiciels (layout, calcul de modularité, etc.).

12. Autres possibilités

Pour des **SCN** de très grande envergure (millions de nœuds, millions d'arêtes), le volume de données devient prohibitif dans un format JSON ou CSV. On recourt alors à des exports **binaires** ou à des protocoles de sérialisation plus

compacts (Protobuf, Cap'n Proto). L'essentiel est de réduire la taille des liens enregistrés, éventuellement en exploitant la **parcimonie** et en ne stockant que les arêtes supérieures à un certain seuil.

C. Mécanismes de Production et d'Accès

Pour offrir de tels formats, un composant interne peut réaliser :

13. Une **API** ou un **service** spécialisé dans l'export. Celui-ci reçoit une requête, par exemple `GET /scn/clusters?threshold=0.3`, exécute `getAllClusters(0.3)`, puis sérialise le résultat en JSON ou CSV.
14. Une **fonction** locale, telle que `exportLinks(format="CSV")`, qui génère un fichier ou un flux textuel contenant la liste des liaisons.
15. Des **instantanés** programmés : à chaque itération ou toutes les N itérations, on produit un "snapshot" de l'état actuel du SCN (avec labels de cluster), que l'on stocke dans un répertoire ou que l'on envoie à un pipeline de data-mining.

Le **temps réel** (visualisation en continu) peut être assuré par des technologies comme WebSocket ou SSE (Server-Sent Events), où le SCN pousse automatiquement des mises à jour sous forme de JSON chaque fois qu'un changement notable (évolution de la matrice ω , arrivée/suppression d'entité) se produit.

D. Exemples d'Usage Externe

16. Visualisation d'un Graphe Dynamique

Un tableau de bord (exemple avec D3.js) récupère l'état courant via `getAllClusters(θ)`, puis construit un JSON "nodes/links". L'interface affiche un graphe évolutif où les nœuds sont colorés par *cluster*. Si un nœud \mathcal{E}_i change de cluster suite à un réajustement de ω , le prochain appel reflète cette mutation, donnant un effet d'animation ou de transition dans la vue utilisateur.

17. Réintégration dans une Chaîne de Classification

Un module aval, chargé de classer les entités (ex. détection de fraude dans des transactions), demande régulièrement la partition $\{\mathcal{C}_i\}$ ou la liste des top- k liens de chaque entité. Il intègre ces données comme *features*, afin de compléter l'information sur chaque transaction ou utilisateur. Par exemple, un utilisateur dont le cluster diffère subitement de son historique peut être marqué comme suspect.

18. Diagnostic et Archivage

Dans des systèmes complexes, on peut consigner — à intervalles réguliers — un export CSV ou JSON de la matrice ω scellée. Ces archives constituent une base d'analyse a posteriori pour valider la stabilité de l'auto-organisation, expliquer des événements remarquables (consolidation ou éclatement de clusters), ou confronter le réseau à un "oracle" de vérité terrain.

Conclusion

Les informations sur la structure d'un SCN (top-links, clusters, communautés) ne demeurent pas confinées au module d'auto-organisation. Elles doivent être **exposées** sous des formats standards (JSON, CSV, GraphML...) à d'autres composantes, qu'il s'agisse de bibliothèques de visualisation, d'outils de classification ou de frameworks d'exploration de graphes. Sur le plan **ingénierie**, on mettra ainsi en place :

- Des **endpoints** ou **fonctions** d'export (ex. `exportClusters(theta, format="json")`),
- Un **protocole** de diffusion (API REST, WebSocket, fichiers),
- Une **intégration** dans des bibliothèques ou plateformes de data science.

Ce design veille à équilibrer la **lisibilité** (formats standard), la **performance** (possibilité d'extraire seulement un sous-ensemble ou de filtrer par seuil) et la **dimension** du SCN (certaines structures massives imposant des stratégies d'export parcimonieux). L'essentiel demeure de faire en sorte que le SCN, en tant que *cœur adaptatif*, puisse rayonner ses

informations de clusterisation au **reste** de l'architecture logicielle, facilitant à la fois la **compréhension** humaine et le **traitement** automatique par les algorithmes externes.

5.7. Distribution, Scalabilité et Architecture Modulaire

5.7.1. SCN Distribué sur Plusieurs Nœuds

- 5.7.1.1. Motifs : si n est grand, on peut diviser en sous-SCN (chacun gère un sous-ensemble d'entités).
- 5.7.1.2. Communication et protocole d'échange ω inter-sous-SCN.
- 5.7.1.3. Risque d'incohérence et solutions (synchronisation épisodique, etc.).

5.7.2. Mise en Place d'un "meta-SCN"

- 5.7.2.1. Idée : un SCN global reliant plusieurs sous-SCN, usage d'un "graphon" ou d'un "meta-nœud".
- 5.7.2.2. Périodicité de la mise à jour, agrégation de clusters émergents.
- 5.7.2.3. Cas d'une architecture microservices : chaque service = sous-SCN.

5.7.3. Ingénierie Logicielle

- 5.7.3.1. Patterns de conception (Observer, Strategy, etc.) pour moduler l'implémentation.
- 5.7.3.2. Approches NoSQL, BDD orientée graphe (Neo4j, Titan) pour stocker ω .
- 5.7.3.3. Équilibrage de charge (sharding de la matrice ω).

5.7. Distribution, Scalabilité et Architecture Modulaire

Dans l'optique d'un **SCN** (Synergistic Connection Network) de très grande taille (nombre d'entités n élevé) ou réparti entre différents sites, la question de la **distribution** et de la **scalabilité** se pose inévitablement. Le chapitre 5.7 se concentre sur cette problématique : comment structurer un SCN en plusieurs sous-parties (sous-SCN), chacune gérant une portion du réseau ou un sous-ensemble d'entités, et comment organiser la communication et la synchronisation inter-blocs. Même si cette approche s'éloigne d'une vision purement locale ou centralisée (chapitres précédents), elle s'avère indispensable pour une **architecture** modulaire et apte à traiter des volumes importants de données ou des entités éparpillées sur plusieurs nœuds.

5.7.1. SCN Distribué sur Plusieurs Nœuds

Lorsque la dimension n d'un SCN devient considérable ou que les entités se trouvent déjà distribuées (ex. multiples robots, serveurs distants, etc.), on procède à un **partitionnement** du réseau en "sous-SCN". Cette section (5.7.1) en présente les justifications (5.7.1.1), puis explore les questions de communication (5.7.1.2) et de synchronisation (5.7.1.3).

5.7.1.1. Motifs : si n est grand, on peut diviser en sous-SCN (chacun gère un sous-ensemble d'entités)

Lorsqu'un **Synergistic Connection Network (SCN)** doit gérer un nombre très élevé d'entités $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$, les défis se multiplient tant du point de vue **mathématique** qu'**algorithmique** et **matériel**. En effet, si n atteint des valeurs de l'ordre de 10^5 à plusieurs millions, la gestion centralisée d'une matrice de pondérations Ω de dimensions $n \times n$ devient rapidement problématique. D'un point de vue **mémoire**, le coût de stockage de Ω croît en $\mathcal{O}(n^2)$, ce qui, pour de grandes valeurs de n , peut représenter plusieurs dizaines, voire centaines de gigaoctets, même en utilisant des structures de données optimisées telles que des matrices **sparse**. Du point de vue **algorithmique**, la mise à jour de toutes les pondérations nécessite la réalisation d'un nombre quadratique d'opérations par itération, ce qui rend la boucle de mise à jour extrêmement coûteuse en temps de calcul sur une seule machine. Enfin, sur le plan **matériel**, l'accès à la bande passante mémoire et le parallélisme deviennent des goulots d'étranglement majeurs lorsqu'on traite simultanément des millions de liaisons.

Afin de surmonter ces limitations, il est judicieux de segmenter le SCN en plusieurs **sous-SCN**. Chaque sous-SCN est responsable de la gestion d'un sous-ensemble d'entités, ce qui permet de partitionner la matrice globale Ω en blocs plus petits, notés par $\Omega^{(p)}$ pour $p = 1, \dots, m$. Mathématiquement, si l'ensemble des entités est divisé en m groupes $\mathcal{V}_1, \dots, \mathcal{V}_m$ avec $|\mathcal{V}_p| \approx \frac{n}{m}$, chaque sous-SCN gère une matrice de taille approximative $\left(\frac{n}{m}\right)^2$. Ainsi, le coût de stockage et de mise à jour de chaque bloc est réduit en $\mathcal{O}\left(\frac{n^2}{m^2}\right)$ par rapport au système centralisé, et les interactions inter-blocs peuvent être traitées séparément, voire de manière moins fréquente.

Du point de vue **théorique**, le partitionnement en sous-SCN permet également de mieux capturer des **structures naturelles** ou des **séparations géographiques** dans le réseau. Par exemple, dans un réseau de robots répartis sur plusieurs sites physiques ou dans un système multi-agent où les entités présentent des caractéristiques distinctes (capteurs, actuateurs, utilisateurs, etc.), il est cohérent de regrouper les entités par affinité ou proximité. Ce découpage favorise non seulement une **réduction des coûts** (tant en mémoire qu'en temps de calcul) mais aussi une **meilleure résilience**. En cas de défaillance d'un sous-SCN, les autres continuent de fonctionner de manière autonome, assurant une tolérance aux pannes au niveau du système global.

Du point de vue **ingénierie**, l'implémentation d'un SCN partitionné est facilitée par une architecture logicielle modulaire. Chaque sous-SCN peut être déployé sur un serveur ou un nœud de calcul différent, avec des interfaces standardisées pour la communication inter-blocs. Cette approche modulaire simplifie la gestion de la parallélisation et permet d'exploiter des stratégies de calcul distribué, où la mise à jour locale se fait indépendamment dans chaque sous-SCN, et où les interactions inter-blocs sont traitées par des protocoles d'échange de messages ou des agrégations périodiques.

Implémentation Python avec Graphiques

Le code Python suivant simule un SCN avec un grand nombre d'entités que nous partitionnons en plusieurs sous-SCN. Chaque sous-SCN gère la mise à jour de ses propres pondérations $\omega_{i,j}$ en utilisant la règle additive classique, et nous visualisons ensuite la structure finale des matrices pour observer l'effet de la segmentation.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Paramètres de base
n = 200          # Nombre total d'entités
m = 4           # Nombre de sous-SCN (groupes)
eta = 0.05       # Taux d'apprentissage
tau = 1.0        # Coefficient de décroissance
iterations = 100 # Nombre d'itérations pour chaque sous-SCN

# Création d'une matrice de synergie S globale pour toutes les entités
# Pour simplifier, on fixe  $S(i,j) = 0.8$  pour  $i \neq j$  et 0 sur la diagonale
S_global = np.full((n, n), 0.8)
np.fill_diagonal(S_global, 0)

# Partitionner les entités en m groupes égaux
group_size = n // m
subSCN_indices = [range(i * group_size, (i + 1) * group_size) for i in range(m)]

# Initialisation des matrices de pondérations pour chaque sous-SCN
omega_sub = {}
for idx, indices in enumerate(subSCN_indices):
    size = len(indices)
    # Initialisation aléatoire faible pour éviter l'absence de signal
    omega_sub[idx] = np.random.uniform(0, 0.05, (size, size))
    np.fill_diagonal(omega_sub[idx], 0)

# Fonction de mise à jour déterministe additive pour un sous-SCN
def update_omega(omega, S, eta, tau):
    new_omega = omega.copy()
    size = omega.shape[0]
    for i in range(size):
        for j in range(size):
            if i != j:
                delta = eta * (S[i, j] - tau * omega[i, j])
                new_omega[i, j] = omega[i, j] + delta
                # Éviter les valeurs négatives par clipping
                new_omega[i, j] = max(new_omega[i, j], 0)
    return new_omega

# Stockage de l'évolution pour visualisation
omega_history = {idx: [] for idx in range(m)}

# Simulation de mise à jour pour chaque sous-SCN
for idx, indices in enumerate(subSCN_indices):
    omega = omega_sub[idx]
    # On suppose que la matrice de synergie S pour chaque sous-SCN est extraite de S_global
    S_local = S_global[np.ix_(list(indices), list(indices))]
    # Stockage initial
```

```

omega_history[idx].append(omega.copy())
for t in range(iterations):
    omega = update_omega(omega, S_local, eta, tau)
    omega_history[idx].append(omega.copy())
omega_sub[idx] = omega # Sauvegarde finale dans le dictionnaire

# Visualisation : Heatmap finale de chaque sous-SCN
plt.figure(figsize=(14, 10))
for idx in range(m):
    plt.subplot(2, m // 2, idx + 1)
    sns.heatmap(omega_sub[idx], cmap="viridis", cbar=True)
    plt.title(f"Sous-SCN {idx+1} (taille = {group_size}x{group_size})")
plt.tight_layout()
plt.show()

# Visualisation : Évolution de la moyenne des pondérations dans chaque sous-SCN
plt.figure(figsize=(10, 6))
for idx in range(m):
    mean_values = [np.mean(omega_history[idx][t]) for t in range(iterations + 1)]
    plt.plot(mean_values, label=f"Sous-SCN {idx+1}")
plt.xlabel("Itérations")
plt.ylabel("Pondération moyenne")
plt.title("Évolution de la Pondération Moyenne dans Chaque Sous-SCN")
plt.legend()
plt.grid(True)
plt.show()

```

Analyse et Discussion

Principes et Formulation Mathématique

La mise à jour de chaque pondération dans un SCN est déterminée par la formule additive

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)].$$

Dans le contexte d'un grand nombre d'entités, la division en **sous-SCN** permet de limiter les coûts de stockage et de calcul en traitant séparément des blocs de la matrice Ω . Chaque sous-SCN est associé à une sous-matrice $\omega^{(p)}$ de taille réduite, ce qui rend l'algorithme de mise à jour plus gérable, tant du point de vue algorithmique que matériel.

Implémentation et Visualisation

L'implémentation Python proposée simule le processus de mise à jour pour chaque sous-SCN. Le code partitionne les entités en m groupes égaux, extrait la matrice de synergie locale correspondante, et applique itérativement la règle de mise à jour additive. Les visualisations générées – des heatmaps finales pour chaque sous-SCN et des courbes montrant l'évolution de la pondération moyenne – illustrent clairement l'auto-organisation locale dans chaque partition. Ces graphiques permettent d'observer la convergence progressive des pondérations, tout en montrant que la division en sous-SCN réduit la complexité du problème global.

Avantages du Partitionnement

En divisant le réseau en sous-SCN, on réduit le coût en mémoire de la matrice Ω et on limite le nombre d'opérations par itération. Ce découpage favorise également la parallélisation, car chaque sous-SCN peut être traité indépendamment, voire sur des machines différentes. De plus, dans des scénarios où les entités présentent des caractéristiques géographiques ou typologiques distinctes, le partitionnement permet d'adapter la dynamique locale aux spécificités de chaque sous-ensemble.

Conclusion

La stratégie de partitionnement d'un grand SCN en sous-SCN permet de surmonter les limitations inhérentes à un réseau centralisé de grande dimension. D'un point de vue mathématique, la segmentation réduit la complexité en divisant une matrice Ω de taille $\mathcal{O}(n^2)$ en plusieurs sous-matrices de taille réduite, chacune traitée de manière autonome. Du point de vue algorithmique et matériel, cette approche favorise la scalabilité et la résilience du système, tout en facilitant l'implémentation de mises à jour parallèles et de stratégies de contrôle local. L'implémentation Python présentée, assortie de graphiques, démontre concrètement comment la dynamique d'auto-organisation se déploie dans chaque sous-SCN, offrant ainsi un outil pédagogique et pratique pour l'étude des SCN à grande échelle.

5.7.1.2. Communication et Protocole d'Échange ω Inter-sous-SCN

Dans un **Synergistic Connection Network** (SCN) segmenté en plusieurs sous-réseaux distincts, la gestion des interactions entre entités réparties dans différents sous-SCN devient une question cruciale. Lorsqu'un ensemble d'entités $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ est divisé en plusieurs blocs $\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_m$, chaque sous-SCN gère localement sa propre matrice de pondérations, c'est-à-dire les valeurs $\omega_{i,j}$ pour $i, j \in \mathcal{V}_p$. Cependant, les liaisons inter-blocs, c'est-à-dire celles pour lesquelles $i \in \mathcal{V}_p$ et $j \in \mathcal{V}_q$ avec $p \neq q$, nécessitent une coordination particulière. La présente section expose en détail les principes et enjeux de la **communication** et des **protocoles d'échange** permettant de synchroniser les pondérations inter-sous-SCN.

A. Notion de Liens Inter-Blocs et Rôle du Protocole

Lorsque le SCN est segmenté, il ne s'agit pas de traiter chaque sous-SCN comme un système isolé ; en effet, des **synergies** $S(i, j)$ importantes peuvent exister entre des entités appartenant à des blocs différents. Mathématiquement, pour des entités $\mathcal{E}_i \in \mathcal{V}_p$ et $\mathcal{E}_j \in \mathcal{V}_q$ avec $p \neq q$, la pondération $\omega_{i,j}$ est une variable d'importance qui doit être intégrée dans la dynamique globale du SCN. Dans un tel cadre, le protocole de communication a pour objectif de garantir que ces valeurs inter-blocs soient correctement mises à jour et que les informations échangées entre les sous-SCN reflètent la synergie réelle entre les entités, même si elles résident sur des nœuds ou des serveurs différents.

D'un point de vue mathématique, la dynamique globale peut être vue comme un système couplé, dans lequel chaque sous-SCN contribue à l'évolution de l'ensemble des pondérations. Ainsi, les mises à jour locales dans un sous-SCN, notées par exemple $\omega^{(p)}(t)$ pour le bloc \mathcal{V}_p , doivent être complétées par des échanges d'informations pour les pondérations $\omega_{i,j}$ lorsque $i \in \mathcal{V}_p$ et $j \in \mathcal{V}_q$. Le protocole doit donc assurer la **transmission** et la **synchronisation** de ces données inter-blocs afin de préserver la cohérence globale.

B. Protocoles de Communication et de Synchronisation

Pour assurer un échange efficace et cohérent des pondérations inter-sous-SCN, plusieurs stratégies peuvent être envisagées. Parmi elles, deux approches principales se dégagent :

- **Responsabilité Unique** : Dans ce mode, une sous-SCN est désignée comme responsable de la gestion d'un lien inter-blocs particulier. Par exemple, si $\omega_{i,j}$ relie une entité de \mathcal{V}_p à une entité de \mathcal{V}_q , alors le sous-SCN de \mathcal{V}_p (ou un module central dédié) stocke et met à jour cette pondération. Les autres blocs qui doivent accéder à cette information effectuent des requêtes périodiques pour obtenir la valeur actualisée. Cette méthode centralise la mise à jour d'un lien spécifique, ce qui facilite la garantie d'une cohérence temporelle, mais nécessite une définition claire de la responsabilité de chaque lien.
- **Copie Locale et Synchronisation** : Alternativement, chaque sous-SCN peut maintenir sa propre copie locale des pondérations inter-blocs. Dans ce cas, lorsqu'un sous-SCN modifie une pondération $\omega_{i,j}$ pour $i \in \mathcal{V}_p$ et $j \in \mathcal{V}_q$, il doit diffuser cette modification aux autres sous-SCN concernés. Pour gérer ce processus, un mécanisme de synchronisation est indispensable, souvent via l'attribution de **timestamps** ou de **numéros de version** pour chaque mise à jour. La synchronisation peut se faire de manière asynchrone, avec une stratégie de réconciliation en cas de divergence entre copies, ou par des rounds synchrones qui imposent un "barrier" d'échange à la fin de chaque itération.

Sur le plan algorithmique, ces stratégies se traduisent par des protocoles de type **round-based synchronization** dans lesquels, à chaque cycle d'itération, les sous-SCN effectuent d'abord leurs mises à jour locales, puis entrent dans une phase de communication où ils échangent les valeurs des pondérations inter-blocs. Ce mécanisme garantit que, au début de chaque itération, toutes les parties du système disposent d'une version cohérente de la matrice ω .

C. Gestion des Conflits et Garanties de Cohérence

Dans un système distribué, l'actualisation concurrente des liens inter-blocs peut générer des **conflits** si plusieurs sous-SCN tentent de modifier la même pondération simultanément. Pour éviter ce genre de problèmes, plusieurs méthodes peuvent être employées :

- L'usage de **verrous** (locks) ou de mécanismes de synchronisation fine qui assurent qu'une seule mise à jour d'une pondération donnée se réalise à la fois. Bien que cette approche garantisse une cohérence forte, elle peut ralentir la dynamique si le nombre de conflits est important.
- La technique du **double-buffer**, dans laquelle la matrice $\omega(t)$ est lue en lecture seule pendant que les mises à jour sont écrites dans une matrice distincte ω_{next} . À la fin de l'itération, un **barrier** de synchronisation permet d'effectuer un swap complet entre ω_{next} et $\omega(t + 1)$, assurant ainsi la cohérence sans verrouillage fin.
- Des **algorithmes lock-free** ou de type **Gossip**, qui permettent une mise à jour asynchrone tout en utilisant des numéros de version ou des timestamps pour réconcilier les divergences. Ces algorithmes, bien qu'efficaces en termes de performances, requièrent une analyse rigoureuse pour garantir la convergence du système.

Le choix entre ces approches dépend fortement de la taille du SCN et de la vitesse de communication entre les sous-SCN. Pour des systèmes de grande envergure, où la latence et la bande passante peuvent devenir critiques, une approche hybride combinant rounds synchrones périodiques et mises à jour asynchrones intermédiaires peut s'avérer optimale.

D. Conclusion

La **communication et le protocole d'échange** des pondérations inter-sous-SCN représentent une composante essentielle pour maintenir la cohérence globale dans un SCN distribué. En segmentant le réseau en sous-SCN, il est indispensable de définir un protocole précis pour l'échange des informations entre les blocs afin que les synergies entre entités situées dans des sous-ensembles différents soient prises en compte correctement. Les solutions vont de la responsabilité unique, où un seul bloc est responsable d'un lien donné, à la copie locale synchronisée via des mécanismes de versioning. Le choix du protocole doit être guidé par les exigences en termes de performance, de latence et de robustesse de convergence. Ainsi, cette communication inter-blocs permet à l'ensemble du système de s'auto-organiser de manière cohérente, tout en assurant une évolutivité et une résilience accrues dans des environnements distribués complexes.

Ces mécanismes de synchronisation et d'échange garantissent que, malgré la dispersion physique ou logique des sous-SCN, le SCN global conserve une dynamique intégrée qui favorise l'émergence de clusters pertinents et la formation d'une organisation réseau robuste.

5.7.1.3. Risque d'Incohérence et Solutions (Synchronisation Épisodique, etc.)

Dans le cadre d'un **Synergistic Connection Network** (SCN) divisé en plusieurs sous-réseaux, la répartition des entités en blocs distincts pose inévitablement le problème de la cohérence des mises à jour des liens inter-blocs. En effet, alors que chaque sous-SCN gère localement la dynamique de ses propres pondérations $\omega_{i,j}$ pour i, j appartenant à un même ensemble \mathcal{V}_p , les connexions reliant des entités de blocs différents, c'est-à-dire des pondérations pour lesquelles $i \in \mathcal{V}_p$ et $j \in \mathcal{V}_q$ avec $p \neq q$, ne sont plus mises à jour de manière centralisée. Cette situation engendre plusieurs défis, notamment des retards dans la diffusion des mises à jour et des déphasages pouvant conduire à une incohérence globale du SCN.

A. Nature du Risque d'Incohérence

Dans un SCN centralisé, la mise à jour de la matrice $\omega(t)$ est effectuée de façon homogène à l'aide d'une fonction d'itération unique, ce qui permet d'assurer que chaque composante $\omega_{i,j}$ évolue en fonction des mêmes informations à chaque itération. Cependant, dans un SCN distribué, chaque sous-SCN maintient une partie de la matrice, et les liens inter-blocs $\omega_{i,j}$ sont actualisés localement par différents agents ou processus de calcul. Par conséquent, deux sous-SCN peuvent ne pas disposer simultanément de la même version de $\omega_{i,j}$ lorsque des mises à jour sont effectuées. Par exemple, le sous-SCN SCN_p pourrait mettre à jour $\omega_{i,j}$ et obtenir une valeur de 0.65, tandis que le sous-SCN SCN_q n'a pas encore reçu cette mise à jour et maintient une valeur de 0.50. Ce décalage temporel engendre une incohérence, car des corrections contradictoires peuvent être appliquées ensuite par les deux sous-réseaux, chacun se basant sur une information obsolète. Mathématiquement, on peut considérer que l'ensemble des pondérations inter-blocs obéit à une dynamique couplée asynchrone : si la fonction de mise à jour locale dans le sous-SCN p est donnée par

$$\omega_{i,j}^{(p)}(t+1) = F\left(\omega_{i,j}^{(p)}(t), S_{i,j}\right),$$

alors l'absence de synchronisation précise avec $\omega_{i,j}^{(q)}(t)$ peut faire en sorte que, sur le long terme, les versions locales ne convergent pas vers un unique point fixe commun. Un tel phénomène se traduit par des oscillations ou par une divergence progressive des valeurs d'une même liaison au sein des différents blocs.

B. Solutions pour Limiter l'Incohérence

Afin de pallier ces problèmes, plusieurs stratégies peuvent être mises en œuvre, dont les principales sont les synchronisations épisodiques et l'attribution d'une responsabilité unique pour la mise à jour de chaque lien inter-blocs.

Synchronisation Épisodique :

Une approche efficace consiste à introduire des phases de synchronisation périodique. Pendant une phase d'itération locale, chaque sous-SCN effectue ses mises à jour indépendamment, en utilisant les données disponibles à l'instant t . Puis, après un certain nombre d'itérations, tous les sous-SCN se synchronisent en échangeant leurs versions des pondérations inter-blocs. Cette synchronisation peut se faire par le biais d'une barrière (barrier synchronization) qui bloque temporairement l'évolution jusqu'à ce que chaque sous-SCN ait communiqué sa mise à jour. Mathématiquement, si chaque sous-SCN p possède une version $\omega_{i,j}^{(p)}(t)$, alors à la fin d'un cycle de synchronisation, on peut définir la version globale par une moyenne ou par un mécanisme de sélection, par exemple :

$$\omega_{i,j}^{\text{global}}(t) = \frac{1}{M} \sum_{p=1}^M \omega_{i,j}^{(p)}(t),$$

où M est le nombre de sous-SCN impliqués. Ce procédé permet de réduire les écarts et de réaligner les mises à jour avant de reprendre le cycle d'auto-organisation. Le principal avantage de cette méthode est la garantie d'une cohérence périodique, même si elle introduit une latence dans la propagation des mises à jour.

Responsabilité Unique (Bloc Maître) :

Une autre solution consiste à désigner, pour chaque lien inter-blocs $\omega_{i,j}$, un sous-SCN responsable de sa mise à jour. Par exemple, si $\omega_{i,j}$ relie une entité appartenant à \mathcal{V}_p à une entité dans \mathcal{V}_q , on peut assigner la responsabilité de la mise à jour à SCN_p (ou à un module central dédié). Dans ce cas, tous les sous-SCN concernés se réfèrent à la même source de vérité pour la valeur de $\omega_{i,j}$. Cette méthode élimine la redondance des mises à jour et assure que la dynamique s'appuie sur une valeur unique pour chaque lien inter-blocs. Cependant, elle peut créer un goulet d'étranglement si un nombre important de liens est géré par un seul bloc, et nécessite la mise en place de protocoles de consultation et de notification efficaces.

Approches Hybrides et Paramétrage :

Pour limiter les effets des retards et des divergences, il est également possible de moduler les paramètres de la mise à jour locale, tels que le taux d'apprentissage η , pour atténuer les fluctuations. En combinant des méthodes de

synchronisation épisodique avec un ajustement fin des paramètres (par exemple, en introduisant des coefficients d’inertie qui lissent la transition entre les mises à jour locales et globales), on peut obtenir une dynamique plus robuste. Ces ajustements permettent d’atténuer les oscillations dues aux déphasages entre sous-SCN et de favoriser une convergence harmonieuse vers des configurations cohérentes.

Conclusion

Le risque d’incohérence dans un SCN segmenté découle de la nature distribuée des mises à jour inter-blocs, où des décalages temporels ou des différences dans la réception des informations peuvent engendrer des divergences dans les valeurs de $\omega_{i,j}$. Pour pallier ce problème, plusieurs solutions sont envisageables, telles que la synchronisation épisodique par rounds, la désignation d’un bloc maître pour la mise à jour de chaque lien, ou encore des mécanismes hybrides combinant ajustement paramétrique et synchronisation partielle. Ces approches, en garantissant une cohérence périodique ou continue des mises à jour, permettent au SCN global de maintenir une dynamique intégrée et de favoriser l’émergence de clusters cohérents malgré la dispersion des sous-SCN. D’un point de vue mathématique, ces dispositifs s’intègrent dans un cadre de systèmes dynamiques couplés avec décalages, tandis que d’un point de vue ingénierie, ils constituent le compromis nécessaire pour allier scalabilité, performance et stabilité dans un environnement distribué.

5.7.2. Mise en Place d’un “meta-SCN”

Lorsque l’on segmente un SCN en plusieurs sous-SCN (5.7.1), il peut devenir utile, voire essentiel, de **coordonner** ces blocs par un **réseau de plus haut niveau** : un “meta-SCN”. L’idée générale consiste à créer, au-dessus des sous-SCN, une **couche** ou un **nœud intermédiaire** capable d’agréger l’information inter-blocs, de manière à préserver une forme de cohésion globale. Cette section (5.7.2) présente le concept (5.7.2.1), puis aborde la périodicité de mise à jour (5.7.2.2) et, enfin, l’illustration d’une architecture microservices (5.7.2.3).

5.7.2.1. Idée : un SCN global reliant plusieurs sous-SCN, usage d’un “graphon” ou d’un “meta-nœud”

Dans le cadre d’un **Synergistic Connection Network** (SCN) de grande envergure, où le nombre d’entités n est si élevé qu’il devient inenvisageable de gérer une matrice de pondérations ω de taille $n \times n$ dans un système monolithique, il apparaît judicieux de segmenter le réseau en plusieurs **sous-SCN**. Chaque sous-SCN, noté SCN_p pour $p = 1, \dots, m$, gère un sous-ensemble d’entités $\mathcal{V}_p \subset \{\mathcal{E}_1, \dots, \mathcal{E}_n\}$. Toutefois, pour que le réseau dans son ensemble conserve une cohérence globale, il est indispensable d’introduire une couche d’agrégation supérieure qui relie ces sous-SCN entre elles. Cette couche peut être conceptualisée sous la forme d’un **SCN global** basé sur un **graphon** ou, de façon équivalente, sur un système de **meta-nœuds**.

A. Motivation pour un SCN global

La nécessité de diviser le SCN en sous-ensembles découle principalement de contraintes de **scalabilité** et de **complexité computationnelle**. En effet, lorsque n devient très grand, la complexité mémoire et temporelle d’un traitement centralisé (avec une complexité en $O(n^2)$) devient prohibitive. Par ailleurs, dans des contextes distribués ou hétérogènes (par exemple, des réseaux de robots dispersés géographiquement ou des systèmes multi-agents fonctionnant sur des infrastructures distinctes), il existe des **séparations naturelles** dans les interactions. Ces considérations incitent à organiser le réseau en blocs autonomes. Toutefois, isoler complètement ces blocs conduirait à une perte d’information sur la **synergie** entre des entités appartenant à des sous-ensembles différents. Il est donc souhaitable d’introduire un mécanisme d’agrégation, permettant de condenser l’information relative aux connexions inter-blocs en un ensemble de **pondérations agrégées**. Ce mécanisme est comparable à la notion de **graphon** en théorie des graphes, qui sert d’approximation continue à la matrice d’adjacence d’un grand réseau, ou encore à l’idée de **meta-nœud** dans une approche hiérarchique, où chaque sous-SCN est représenté par un nœud unique dans un graphe de niveau supérieur.

B. Construction d'un Meta-SCN et Modélisation par Graphon

Le **meta-SCN** repose sur l'agrégation des liaisons inter-blocs. Soit $\mathcal{V}_1, \dots, \mathcal{V}_m$ la partition de l'ensemble des entités. Pour chaque paire de blocs (p, q) avec $p \neq q$, on définit une pondération agrégée

$$\hat{\omega}_{p,q} = \Psi(\{\omega_{i,j} : i \in \mathcal{V}_p, j \in \mathcal{V}_q\}),$$

où Ψ représente une **fonction d'agrégation** choisie selon les besoins du système (par exemple, la moyenne, la somme pondérée, ou même le maximum des $\omega_{i,j}$ entre les deux blocs). Cette approche permet de réduire la complexité du SCN global, en passant d'une matrice de dimension $n \times n$ à une matrice $\hat{\omega}$ de dimension $m \times m$, avec $m \ll n$. Dans certains contextes, l'objet continu appelé **graphon** est utilisé pour décrire la limite d'un graphe dense lorsque $n \rightarrow \infty$, fournissant ainsi une représentation continue de la structure d'interaction. Ici, le concept de graphon se traduit par une fonction $W: [0,1]^2 \rightarrow \mathbb{R}^+$ qui, par un processus de rééchantillonnage, permet de représenter les interactions entre blocs de manière continue.

C. Avantages et Enjeux de l'Agrégation Inter-blocs

L'utilisation d'un meta-SCN offre plusieurs avantages d'un point de vue à la fois théorique et pratique. D'un point de vue **mathématique**, l'agrégation permet de traiter la dynamique des interconnexions à un niveau supérieur, ce qui simplifie l'analyse de convergence et de stabilité du système global. En effet, l'étude de la dynamique dans un espace de dimension réduite facilite l'identification des attracteurs et des transitions de phase. Du point de vue **ingénierie**, cette approche permet de réduire la charge de communication entre les différents blocs, puisque les sous-SCN n'ont plus à échanger les valeurs de chaque lien inter-blocs individuellement, mais peuvent communiquer des **agrégats** (moyennes, totaux, etc.), ce qui réduit la surcharge en bande passante et accélère les phases de synchronisation.

Toutefois, cette centralisation partielle par un meta-SCN présente également des limites. La perte de granularité dans l'agrégation peut masquer certaines structures fines qui se développent au niveau microscopique des liaisons individuelles. De plus, l'introduction d'un nœud central ou d'un ensemble de meta-nœuds crée potentiellement des points de congestion ou des risques de panne unique, si ces éléments ne sont pas correctement redondants ou distribués.

D. Conclusion sur l'Approche du Meta-SCN

En résumé, la segmentation d'un SCN en sous-ensembles offre une solution élégante aux problèmes de scalabilité et de complexité lorsque le nombre d'entités est extrêmement grand. Le recours à un **meta-SCN** ou à un **graphon** permet d'agréger l'information des liaisons inter-blocs en une structure de niveau supérieur, facilitant ainsi la coordination globale, la synchronisation et l'analyse des clusters émergents. Cette approche hiérarchique représente un compromis essentiel entre la gestion locale fine des interactions et la nécessité d'une vision globale cohérente du système.

5.7.2.2. Périodicité de la Mise à Jour, Agrégation de Clusters Émergents

Lorsque l'on subdivise un Synergistic Connection Network (SCN) en plusieurs sous-réseaux locaux, il devient indispensable d'établir un mécanisme pour « réunir » ces blocs dans une vision globale cohérente. En effet, chaque sous-SCN, en évoluant selon ses propres dynamiques locales, tend à affiner ses connexions internes. Toutefois, pour qu'un réseau global homogène émerge – où des clusters transcendant les frontières locales se forment – il faut périodiquement synchroniser et agréger les informations relatives aux liens inter-blocs. Ce processus de mise à jour périodique intervient à un niveau méta, permettant de reconstituer la structure globale à partir des agrégats de chaque sous-SCN.

A. Principe d'une Mise à Jour Périodique du Méta-SCN

La démarche consiste à laisser chaque sous-SCN évoluer de manière autonome pendant un certain nombre d'itérations locales, noté T (ou sur un intervalle de temps Δt), pendant lesquelles la règle de mise à jour des pondérations

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)]$$

est appliquée de façon répétée pour les liens internes ou déjà établis entre les sous-SCN. À l'issue de cette phase, chaque sous-SCN compile un résumé de ses liaisons inter-blocs, par exemple en calculant une agrégation telle que la

moyenne ou la somme pondérée des pondérations reliant ses entités aux entités d'un autre sous-SCN. Pour deux sous-SCN, \mathcal{V}_p et \mathcal{V}_q , on peut définir la pondération agrégée

$$\hat{\omega}_{p,q} = \Psi(\{\omega_{i,j} \mid i \in \mathcal{V}_p, j \in \mathcal{V}_q\}),$$

où Ψ désigne une fonction d'agrégation adaptée (par exemple, la moyenne ou la somme). Ainsi, le méta-SCN se construit en tant que graphe de m nœuds (correspondant aux m sous-SCN) et de $\hat{\omega}_{p,q}$ définissant les liaisons entre ces nœuds. Ce schéma permet de limiter le trafic et la charge de calcul en regroupant les mises à jour inter-blocs de manière périodique plutôt qu'en continu, et de préserver l'autonomie locale des sous-SCN entre ces phases de synchronisation.

B. Méthodologie d'Agrégation et Implications sur la Cohérence Globale

Sur le plan mathématique, l'ensemble du système peut être vu comme évoluant à deux échelles distinctes. Pendant les intervalles entre deux synchronisations globales, la dynamique locale est gouvernée par la mise à jour

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)],$$

pour tous les i, j appartenant à un même sous-SCN ou déjà associés par des liaisons inter-blocs précédemment synchronisées. Puis, à chaque période T , une fonction d'agrégation globale, notée G , est appliquée. Formellement, si l'on note $\omega^{\text{global}}(T_k)$ l'ensemble des pondérations inter-blocs au bout du k -ème round global, alors la transition vers le round suivant s'exprime par

$$\omega^{\text{global}}(T_{k+1}) = G(\omega^{\text{global}}(T_k), \{\omega^{(p)}(T_k)\}_{p=1}^m).$$

Cette opération d'agrégation permet de « réinitialiser » la cohérence des valeurs inter-blocs, en compensant les éventuels retards ou divergences accumulées au cours des mises à jour locales. L'équilibre entre la fréquence de synchronisation et la liberté d'évolution locale est déterminant : une synchronisation trop fréquente risque de perturber les dynamiques internes en imposant une correction trop rigide, alors qu'une synchronisation trop espacée peut conduire à des divergences importantes entre les sous-SCN, rendant difficile l'émergence de clusters cohérents au niveau global.

C. Impact sur la Formation des Clusters

Le mécanisme d'agrégation périodique joue un rôle essentiel dans la formation des clusters à l'échelle globale. Chaque sous-SCN, en s'auto-organisant localement, forme des clusters internes qui reflètent la cohésion des entités au sein du bloc. Cependant, la détection de clusters globaux nécessite d'examiner les liaisons inter-blocs : lorsque les agrégats $\hat{\omega}_{p,q}$ indiquent une forte synergie entre des entités de deux sous-SCN distincts, il devient possible de fusionner les clusters locaux correspondants en un cluster global. Ce processus de fusion, réalisé lors des phases de synchronisation globale, permet d'obtenir une vision hiérarchique du réseau. En d'autres termes, la dynamique du SCN est ainsi organisée en deux niveaux : un niveau local, où les sous-SCN optimisent leurs connexions, et un niveau global, où un méta-SCN agrège ces connexions pour détecter des structures de clusters transcendant les frontières locales. Ce couplage multi-échelle favorise une convergence plus robuste et une meilleure adaptation aux variations de la synergie inter-blocs.

D. Conclusion

La périodicité de la mise à jour et l'agrégation des clusters émergents constituent des éléments fondamentaux pour maintenir la cohérence d'un SCN distribué. En permettant aux sous-SCN de se stabiliser localement pendant une période déterminée avant d'intégrer leurs informations au niveau global, on obtient une structure hiérarchique capable de concilier l'efficacité des dynamiques locales avec une vision globale cohérente du réseau. Cette approche, articulée autour d'une synchronisation périodique, assure que les modifications des liaisons inter-blocs soient regroupées et traitées de manière contrôlée, réduisant ainsi les risques d'oscillations et d'incohérences, tout en favorisant l'émergence de clusters significatifs à l'échelle macro.

```
import numpy as np
import matplotlib.pyplot as plt
```

```

import seaborn as sns

# Paramètres de la dynamique locale
eta = 0.05      # Taux d'apprentissage
tau = 1.0       # Coefficient de décroissance
n_total = 200   # Nombre total d'entités
m = 4           # Nombre de sous-SCN (blocs)
T_max = 300     # Nombre total d'itérations locales
T_period = 20   # Période de synchronisation globale (en itérations)

# Distribution homogène des entités entre les blocs
n_block = n_total // m

# Initialisation de la matrice globale omega (dense)
omega = np.random.uniform(0.0, 0.01, (n_total, n_total))
np.fill_diagonal(omega, 0.0) # Pas de liaison auto-connectée

# Initialisation d'une matrice S de synergie aléatoire (pour simplifier)
S = np.random.uniform(0.2, 0.8, (n_total, n_total))
S = (S + S.T) / 2 # Rendre la matrice symétrique
np.fill_diagonal(S, 0.0)

# Stockage des agrégats inter-blocs pour visualisation
meta_history = []

def update_local(omega, S, eta, tau):
    """Mise à jour additive locale pour l'ensemble du réseau."""
    return omega + eta * (S - tau * omega)

def aggregate_inter_block(omega, m, n_block):
    """Agrège les valeurs inter-blocs en calculant la moyenne pour chaque paire de blocs."""
    meta_matrix = np.zeros((m, m))
    for p in range(m):
        for q in range(m):
            # Indices des entités dans le bloc p et q
            idx_p = slice(p * n_block, (p + 1) * n_block)
            idx_q = slice(q * n_block, (q + 1) * n_block)
            # Si p == q, on peut ignorer ou mettre NaN car ce sont des liens internes
            if p == q:
                meta_matrix[p, q] = np.nan
            else:
                # Calcul de la moyenne des omega inter-blocs
                meta_matrix[p, q] = np.mean(omega[idx_p, idx_q])
    return meta_matrix

# Pour visualiser la dynamique globale, on va stocker la matrice meta à chaque période
meta_list = []

# Simulation de la dynamique locale avec synchronisation périodique
for t in range(T_max):
    # Mise à jour locale
    omega = update_local(omega, S, eta, tau)

    # À chaque période T, réaliser l'agrégation inter-blocs
    if (t + 1) % T_period == 0:
        meta = aggregate_inter_block(omega, m, n_block)

```

```

meta_list.append(meta)

# Affichage graphique de la dynamique du meta-SCN
plt.figure(figsize=(8, 6))
# On affiche la dernière matrice meta agrégée
sns.heatmap(meta_list[-1], annot=True, fmt=".2f", cmap="viridis",
             cbar_kws={'label': 'Valeur moyenne des liens inter-blocs'})
plt.title("Matrice Agrégée des Liaisons Inter-blocs après Synchronisation Globale")
plt.xlabel("Bloc q")
plt.ylabel("Bloc p")
plt.show()

# Affichage de l'évolution d'un exemple de lien inter-blocs
# On choisit, par exemple, la liaison entre le bloc 1 et le bloc 2
link_history = [meta[0, 1] for meta in meta_list]

plt.figure(figsize=(8, 4))
plt.plot(np.arange(len(link_history)) * T_period, link_history, marker='o')
plt.title("Évolution de la Valeur Moyenne des Liens Inter-blocs (Bloc 1 à Bloc 2)")
plt.xlabel("Itérations globales")
plt.ylabel("Valeur moyenne de  $\hat{\omega}_{\{1,2\}}$ ")
plt.grid(True)
plt.show()

```

Dans cette implémentation, le **SCN** global est simulé par une matrice ω qui évolue selon la règle additive locale, avec une mise à jour effectuée à chaque itération. Le réseau est divisé en m blocs égaux, chacun correspondant à un sous-SCN. Toutes les TTT itérations, la fonction d'agrégation calcule la moyenne des pondérations inter-blocs pour chaque paire de blocs, constituant ainsi une matrice de niveau supérieur $\hat{\omega}$. Cette matrice agrégée fournit une vue globale sur la dynamique des liens inter-blocs et permet d'identifier visuellement, via une heatmap, les zones où la synergie collective est particulièrement forte ou faible. Le graphe de l'évolution d'un lien inter-blocs (entre le bloc 1 et le bloc 2 dans l'exemple) illustre la manière dont la synchronisation périodique permet de stabiliser et de monitorer la formation de clusters à un niveau macro.

En résumé, la **périodicité** de mise à jour et l'agrégation des clusters émergents jouent un rôle essentiel dans la coordination des sous-SCN. Cette approche hiérarchique assure que la dynamique locale des sous-réseaux se consolide régulièrement dans une structure globale cohérente, facilitant ainsi l'émergence de clusters transcendant les frontières initiales et permettant une meilleure adaptabilité et résilience du SCN global.

5.7.2.3. Cas d'une Architecture Microservices : Chaque Service = Sous-SCN

Dans le cadre d'un **Synergistic Connection Network (SCN)** déployé à grande échelle, la gestion centralisée d'une matrice de pondérations $\{\omega_{i,j}\}$ peut rapidement devenir un goulot d'étranglement, tant au niveau de la **mémoire** que du **calcul**. Pour surmonter ces limitations, il est judicieux de segmenter le SCN en plusieurs sous-réseaux autonomes, chacun étant hébergé dans un **microservice** dédié. Chaque microservice, ou **sous-SCN**, gère un sous-ensemble \mathcal{V}_p d'entités et exécute localement sa propre dynamique de mise à jour des poids. Par ailleurs, une coordination hiérarchique est instaurée pour synchroniser les informations entre les différents microservices, permettant ainsi d'obtenir une vue globale cohérente du réseau. Cette approche modulaire offre de nombreux avantages en termes de **scalabilité**, **résilience** et **flexibilité** architecturale.

A. Conception d'une Architecture Microservices pour un SCN

L'idée fondamentale est de découper l'ensemble des entités $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ en plusieurs groupes disjoints $\mathcal{V}_1, \dots, \mathcal{V}_m$ de taille plus ou moins homogène, de sorte que chaque groupe soit géré par un microservice dédié, désigné par SCN_p pour

$p = 1, \dots, m$. Chaque microservice se charge alors de stocker la matrice locale $\omega^{(p)}$, qui représente les pondérations internes entre les entités appartenant à \mathcal{V}_p . Cette segmentation permet d'éviter la complexité $\mathcal{O}(n^2)$ d'une unique matrice globale, en la divisant en plusieurs blocs de taille réduite, ce qui facilite leur stockage (potentiellement en format *sparse*) et accélère la mise à jour des poids par itération.

L'architecture microservices repose sur le principe de **découplage** fonctionnel : chaque microservice est un module indépendant responsable d'un sous-SCN, et il expose une **API** permettant la communication avec les autres microservices. Par exemple, chaque service peut publier périodiquement les agrégats des liaisons inter-blocs, ou répondre à des requêtes concernant la configuration de ses clusters locaux. Ce mécanisme permet également de gérer la **latence** et d'allouer les ressources de calcul de manière optimale, grâce à une répartition horizontale de la charge. En effet, si le nombre total d'entités n est trop élevé, chaque microservice peut être dimensionné indépendamment ou même subdivisé en services plus petits, assurant ainsi une **scalabilité horizontale** aisée.

B. Considérations Mathématiques et Modélisation

Mathématiquement, si l'on note $\omega_{i,j}^{(p)}(t)$ la pondération entre deux entités i et j au sein du sous-SCN SCN_p , la dynamique locale suit typiquement une équation de mise à jour du type

$$\omega_{i,j}^{(p)}(t+1) = \omega_{i,j}^{(p)}(t) + \eta \left[S^{(p)}(i,j) - \tau \omega_{i,j}^{(p)}(t) \right],$$

où $S^{(p)}(i,j)$ représente la synergie locale entre les entités de \mathcal{V}_p . Pour les liaisons inter-blocs, c'est-à-dire pour $i \in \mathcal{V}_p$ et $j \in \mathcal{V}_q$ avec $p \neq q$, on définit un agrégat global qui peut être noté

$$\hat{\omega}_{p,q} = \Psi \left(\{ \omega_{i,j} \}_{i \in \mathcal{V}_p, j \in \mathcal{V}_q} \right),$$

où Ψ est une fonction d'agrégation (par exemple, la moyenne ou la somme pondérée). Ce sur-graphe, ou **meta-SCN**, permet de représenter la **cohésion** inter-blocs et de prendre des décisions au niveau macro, telles que la fusion de clusters ou la réorganisation des microservices en cas de forte synergie entre eux. Ce modèle mathématique traduit ainsi la hiérarchie du système : la dynamique locale à l'intérieur de chaque microservice et la dynamique globale résultant de l'agrégation des informations inter-blocs.

C. Avantages et Limites de l'Approche Microservices

Du point de vue de l'**ingénierie logicielle**, l'architecture microservices offre plusieurs avantages notables. Premièrement, elle permet un **déploiement indépendant** : chaque sous-SCN peut être développé, testé, mis à jour et déployé sans perturber l'ensemble du réseau. Deuxièmement, cette approche favorise une **modularité** accrue, puisque chaque service est spécialisé dans la gestion d'un sous-ensemble d'entités, ce qui facilite la maintenance et l'extension du système. Troisièmement, la **scalabilité horizontale** est grandement améliorée, car il est possible d'allouer des ressources supplémentaires à un microservice surchargé ou de le subdiviser si nécessaire. Toutefois, cette approche présente également des défis, notamment en ce qui concerne la **coordination** entre les microservices et la gestion des liaisons inter-blocs. La communication entre services doit être soigneusement orchestrée pour éviter les incohérences et assurer la convergence globale du SCN.

Conclusion

L'**approche microservices** pour la gestion d'un SCN distribué se révèle être une solution robuste pour faire face à la complexité induite par un grand nombre d'entités. En assignant à chaque microservice la responsabilité de gérer un sous-SCN local, le système bénéficie d'une meilleure **scalabilité**, d'une plus grande **résilience** en cas de défaillance d'un service individuel, et d'une **modularité** facilitant le développement et la maintenance. Le recours à un meta-SCN pour l'agrégation des synergies inter-blocs permet de conserver une vue globale cohérente du réseau, essentielle pour la formation de clusters significatifs et pour la prise de décisions stratégiques à l'échelle du système. Cette structure hiérarchique, appuyée par des protocoles de communication adaptés, constitue un compromis efficace entre la gestion locale détaillée et la coordination globale nécessaire à un SCN de grande envergure.

5.7.3. Ingénierie Logicielle

Même s'il s'agit d'un ouvrage à dominante mathématique ou théorique, un **SCN** (Synergistic Connection Network) de grande envergure, distribué en plusieurs sous-SCN (5.7.1) et éventuellement chapeauté par un meta-SCN (5.7.2), requiert une **organisation logicielle** suffisamment robuste et modulaire. Le chapitre 5.7.3 vise à souligner les principales approches d'ingénierie employées pour structurer et maintenir un système de ce type, dans la perspective d'une mise en œuvre pérenne ou d'un cadre de recherche-application.

5.7.3.1. Patterns de Conception (Observer, Strategy, etc.) pour Modulariser l'Implémentation

La mise en œuvre d'un **Synergistic Connection Network (SCN)**, tel que défini par ses équations d'évolution de pondération $\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i,j), \dots)$, nécessite non seulement une compréhension rigoureuse de sa dynamique mathématique, mais également une traduction concrète dans un environnement logiciel modulaire et évolutif. Dans ce contexte, l'utilisation de **patterns de conception** (design patterns) issus du génie logiciel orienté objet – notamment les patterns **Strategy** et **Observer** – permet de découpler la logique de mise à jour, la gestion des synergies, et le suivi des événements, tout en facilitant la maintenance et l'extension du système.

A. Contexte et Problématique

Le cœur mathématique d'un SCN se résume souvent à une équation générique telle que

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)],$$

où :

- $\omega_{i,j}(t)$ représente la pondération entre les entités \mathcal{E}_i et \mathcal{E}_j à l'itération t ,
- $S(i,j)$ est la **synergie** entre ces entités,
- η est le **taux d'apprentissage** et τ le coefficient de décroissance.

Cependant, dans une implémentation réelle, cette formule n'est qu'un composant d'un système beaucoup plus vaste qui doit aussi :

- **Calculer** la synergie $S(i,j)$ de manière adaptée aux types d'entités,
- **Adapter** la règle de mise à jour à différentes variantes (par exemple, additive, multiplicative, avec termes stochastiques ou d'inhibition),
- **Observer** et **notifier** les changements de l'état du réseau pour permettre une analyse en temps réel ou une adaptation dynamique.

Pour répondre à ces enjeux, des **patterns de conception** offrent un cadre permettant de modulariser ces préoccupations. Ils permettent d'isoler le « **quoi** » (la formule de mise à jour et le calcul de S) du « **comment** » (la manière dont le code est organisé, exécuté et surveillé). Dans ce cadre, deux patterns se distinguent particulièrement :

- **Strategy** : Pour encapsuler la variation des algorithmes de mise à jour dans des classes interchangeables.
- **Observer** : Pour mettre en place un système de notifications qui informe d'un changement dans l'état des pondérations, sans alourdir le module central.

B. Pattern Strategy : Flexibilité dans la Mise à Jour

Le pattern **Strategy** consiste à définir une **interface** abstraite pour un algorithme, permettant ainsi d'encapsuler différentes implémentations concrètes qui pourront être interchangeables au moment de l'exécution. Dans le cas du SCN, cela signifie créer une interface pour la mise à jour de ω qui se présente, par exemple, sous la forme :

$$\text{applyUpdate}(\omega_{i,j}(t), S(i,j), \text{params}) \rightarrow \omega_{i,j}(t+1).$$

La fonction F qui intervient dans l'équation de mise à jour peut ainsi être décomposée en modules distincts, chacun implémentant une stratégie particulière. Par exemple :

- **AdditiveUpdateRule** :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta[S(i,j) - \tau \omega_{i,j}(t)].$$

- **MultiplicativeUpdateRule** :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) \times [1 + \eta(S(i,j) - \tau \omega_{i,j}(t))].$$

En définissant une interface **UpdateRule** (par exemple, en Python ou en Java), le module central du SCN peut appeler une méthode *applyUpdate* sans se préoccuper des détails de l'algorithme utilisé. Ce découplage facilite l'expérimentation et la comparaison entre différentes dynamiques d'apprentissage.

C. Pattern Observer : Suivi et Notification des Changements

Le pattern **Observer** permet d'établir un mécanisme de **notification** dans lequel un objet (le **sujet**) informe un ensemble d'**observateurs** lorsqu'un événement pertinent se produit. Dans le cadre d'un SCN, le sujet peut être le module qui gère la matrice ω et qui, à chaque itération ou à chaque changement significatif, émet une notification indiquant l'évolution des poids.

Cette approche est utile pour plusieurs raisons :

- Elle permet de **déclencher** des actions supplémentaires, comme l'actualisation de visualisations ou l'extraction de clusters, sans alourdir le code de la mise à jour.
- Elle offre une **décorrélación** entre la logique de mise à jour et les modules d'analyse, ce qui facilite la maintenance.
- Elle permet d'implémenter des **règles adaptatives**, par exemple pour ajuster dynamiquement les paramètres η ou τ en fonction des fluctuations observées.

En pratique, le module central du SCN peut inclure une méthode pour **enregistrer** des observateurs, qui seront appelés par exemple via une méthode *notifyObservers()* dès que l'état de ω est mis à jour.

D. Intégration des Patterns dans une Architecture Modulaire

L'architecture globale du SCN se structure autour d'un **noyau** central qui gère la matrice ω et orchestre la boucle d'auto-organisation. Ce noyau fait appel aux modules suivants :

- Un **UpdateModule** qui, via le pattern Strategy, applique la règle de mise à jour appropriée.
- Un **SynergyModule** qui fournit les valeurs de $S(i,j)$ à partir des représentations des entités.
- Un **ObserverModule** qui, via le pattern Observer, capte les changements dans ω et déclenche des actions (visualisation, logging, ajustement des paramètres).

Mathématiquement, la mise à jour d'un poids peut être réécrite comme :

$$\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i,j), \eta, \tau),$$

où F est encapsulée dans la *Strategy* utilisée, et où le noyau se contente de parcourir la matrice et d'appliquer cette fonction à chaque composante. Les observateurs, quant à eux, reçoivent la notification du changement et effectuent des traitements complémentaires.

E. Conclusion

L'adoption des patterns **Strategy** et **Observer** permet de moduler la mise en œuvre d'un SCN en séparant la logique de mise à jour des poids et le suivi des événements. Cette modularisation confère une grande flexibilité au système, permettant d'expérimenter différentes règles de mise à jour, d'intégrer facilement des fonctionnalités supplémentaires (comme l'inhibition, la saturation ou le recuit stochastique) et d'assurer une surveillance continue de l'évolution de la dynamique auto-organisée. Du point de vue mathématique, le modèle reste inchangé – la fonction F détermine toujours l'évolution des $\omega_{i,j}$ – tandis que l'architecture logicielle garantit que cette dynamique peut être adaptée, étendue et maintenue dans des environnements complexes.

Exemple Pseudo-implémentation en Python

Voici une implémentation simplifiée en Python qui illustre l'utilisation des patterns **Strategy** et **Observer** dans le cadre de la mise à jour des poids d'un SCN :

```
import numpy as np
import matplotlib.pyplot as plt

# Définition des paramètres généraux
params = {'eta': 0.05, 'tau': 1.0}
T_max = 100 # Nombre d'itérations
n = 10      # Nombre d'entités

# Création d'une matrice de synergie S (symétrique, sans auto-connexion)
np.random.seed(42)
S = np.random.uniform(0.5, 1.0, (n, n))
for i in range(n):
    S[i, i] = 0.0
    for j in range(i + 1, n):
        S[j, i] = S[i, j]

# Initialisation de la matrice de poids w
w = np.zeros((n, n))

# -----
# Pattern Strategy: Interface de mise à jour
# -----
class UpdateRule:
    def apply(self, old_weight, synergy, params):
        raise NotImplementedError

class AdditiveUpdateRule(UpdateRule):
    def apply(self, old_weight, synergy, params):
        return old_weight + params['eta'] * (synergy - params['tau'] * old_weight)

class MultiplicativeUpdateRule(UpdateRule):
    def apply(self, old_weight, synergy, params):
        return old_weight * (1 + params['eta'] * (synergy - params['tau'] * old_weight))

# Choix de la stratégie de mise à jour
update_rule = AdditiveUpdateRule() # On peut changer pour MultiplicativeUpdateRule()

# -----
# Pattern Observer: Système de notification
# -----
class Observer:
    def update(self, w, t):
```

```

    raise NotImplementedError

class WeightObserver(Observer):
    def __init__(self):
        self.mean_history = []
        self.max_history = []

    def update(self, w, t):
        self.mean_history.append(np.mean(w))
        self.max_history.append(np.max(w))

# Instanciation de l'observer
observer = WeightObserver()

# Historique pour visualisation
w_history = [w.copy()]

# Boucle d'itération principale du SCN
for t in range(T_max):
    w_next = np.copy(w)
    for i in range(n):
        for j in range(n):
            if i != j:
                w_next[i, j] = update_rule.apply(w[i, j], S[i, j], params)
    w = w_next
    w_history.append(w.copy())
    observer.update(w, t)

# Visualisation des statistiques de poids au fil des itérations
iterations = range(T_max)
plt.figure(figsize=(10, 6))
plt.plot(iterations, observer.mean_history, label="Moyenne des poids")
plt.plot(iterations, observer.max_history, label="Maximum des poids")
plt.xlabel("Itérations")
plt.ylabel("Valeur des poids")
plt.title("Évolution des poids dans le SCN avec mise à jour additive")
plt.legend()
plt.grid(True)
plt.show()

# Visualisation finale de la matrice de poids sous forme de heatmap
plt.figure(figsize=(8, 6))
plt.imshow(w, cmap="viridis", interpolation="nearest")
plt.colorbar(label="Valeur de  $\omega$ ")
plt.title("Heatmap finale de la matrice de poids  $\omega$ ")
plt.xlabel("Index j")
plt.ylabel("Index i")
plt.show()

```

Explications Complémentaires

Dans cette implémentation, le pattern **Strategy** est incarné par l'interface *UpdateRule* et ses implémentations concrètes, ici *AdditiveUpdateRule* (et éventuellement *MultiplicativeUpdateRule*). Le module central de mise à jour du SCN se contente d'appeler la méthode *apply* pour chaque paire (i, j) , ce qui permet d'abstraire le choix de la dynamique

de mise à jour. Cela facilite grandement l'expérimentation, car il suffit de changer l'instance de la stratégie pour tester différentes formules.

Le pattern **Observer** est utilisé via la classe *WeightObserver* qui, à chaque itération, enregistre des statistiques globales (moyenne et maximum des poids). Ces données sont ensuite exploitées pour visualiser l'évolution de la dynamique, offrant ainsi une **vision** en temps réel des comportements émergents.

L'approche adoptée garantit que la **logique mathématique** — la formule de mise à jour — reste distincte de l'**orchestration** logicielle et de la surveillance, assurant ainsi une architecture modulaire et facilement extensible. Les visualisations générées par les graphiques fournissent une illustration claire de la convergence (ou des oscillations éventuelles) des pondérations, permettant ainsi de valider l'auto-organisation du SCN.

En résumé, l'utilisation conjointe des patterns **Strategy** et **Observer** permet de moduler l'implémentation d'un SCN de manière flexible, tout en assurant que la dynamique auto-organisée reste fidèle aux principes mathématiques sous-jacents, et que le système peut être facilement monitoré et étendu dans un cadre logiciel moderne.

5.7.3.2. Approches NoSQL, BDD Orientée Graphe (Neo4j, Titan) pour Stocker ω

La gestion de la **persistance** des pondérations $\omega_{i,j}$ dans un Synergistic Connection Network (SCN) à grande échelle pose des défis majeurs tant sur le plan **mémoire** que sur celui des **requêtes complexes**. Lorsque le nombre d'entités n devient très important, la matrice $\Omega \in \mathbb{R}^{n \times n}$ contenant l'ensemble des $\omega_{i,j}$ peut atteindre une taille de l'ordre de $O(n^2)$. Dans ces conditions, un simple stockage en mémoire vive ou sous forme de fichiers plats ne suffit pas pour assurer des mises à jour efficaces, l'extraction rapide de sous-ensembles pertinents ou l'analyse ultérieure de la structure globale du réseau.

A. Choix d'une Base de Données NoSQL ou Graph

Les **bases de données relationnelles** traditionnelles, malgré leur maturité, peinent à gérer de très grandes quantités de données lorsqu'il s'agit de modéliser des graphes denses. Par exemple, le stockage d'une table *Edges(source, target, weight)* dans une base SQL génère rapidement des contraintes en termes de bande passante et de temps de réponse pour des requêtes complexes telles que la recherche des top- k liens sortants ou la détection de communautés.

Les **bases NoSQL** offrent une solution alternative en se focalisant sur la scalabilité horizontale et la distribution des données. Parmi celles-ci, les bases **orientées graphe** — telles que **Neo4j** et **JanusGraph/Titan** — se distinguent par leur capacité native à représenter directement les **nœuds** et les **arêtes**. Dans ce paradigme, chaque entité \mathcal{E}_i est modélisée comme un nœud, et chaque pondération $\omega_{i,j}$ apparaît comme une arête dotée d'un attribut « weight ». Mathématiquement, on peut assimiler cette représentation à une fonction

$$\omega: V \times V \rightarrow \mathbb{R}^+,$$

où V désigne l'ensemble des nœuds. Les langages de requête dédiés, tels que **Cypher** pour Neo4j ou **Gremlin** pour JanusGraph, permettent d'extraire aisément des sous-graphes, de calculer des agrégats ou d'appliquer des algorithmes de détection de communautés sur le graphe persistant.

B. Intégration dans la Dynamique du SCN

Du point de vue de l'**auto-organisation**, le SCN évolue selon une dynamique décrite par la relation générale

$$\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i,j), \text{paramètres}),$$

où F représente la fonction de mise à jour (qui peut être additive, multiplicative ou comporter des termes stochastiques). Dans un contexte de grande échelle, il devient impératif de persister les valeurs de $\omega_{i,j}$ afin de :

- **Conserver** un historique des évolutions pour des analyses ultérieures (audit, vérification de convergence ou détection d'événements anormaux),
- **Exécuter** des requêtes en temps réel pour extraire les **clusters** ou les **top- k** liens, et

- **Distribuer** la charge en exploitant les capacités de partitionnement et de réplication offertes par les bases NoSQL.

L'intégration de la dynamique dans une base orientée graphe consiste donc à synchroniser, de manière périodique ou en temps réel, la matrice Ω avec la structure persistée. Pour cela, des mécanismes de *batch update* ou de mise à jour épisodique sont mis en place afin de limiter le nombre d'opérations d'écriture, souvent regroupées par des algorithmes d'agrégation sur les liens inter-blocs.

C. Approches de Stockage et de Requête

L'utilisation d'une base **orientée graphe** présente plusieurs avantages du point de vue algorithmique et mathématique. Premièrement, la représentation d'un SCN dans un graphe $G = (V, E)$ se traduit naturellement par :

- Chaque nœud $v \in V$ représente une entité \mathcal{E}_i ,
- Chaque arête $e = (u, v) \in E$ est associée à une pondération $\omega_{u,v}$.

Cette correspondance directe facilite l'implémentation de **requêtes** complexes, telles que la recherche des voisins d'un nœud ayant un poids supérieur à un seuil donné, ou l'extraction des chemins les plus courts pondérés par ω . Par exemple, une requête Cypher dans Neo4j pour récupérer les liens les plus forts pourrait être formulée ainsi :

MATCH (i)-[r]->(j) WHERE r.weight > θ RETURN i, j, r.weight ORDER BY r.weight DESC LIMIT k ,

ce qui permet de visualiser la structure des **clusters** émergents et d'identifier les **communautés** présentes dans le réseau.

De plus, l'aspect distribué des bases telles que **JanusGraph** (souvent couplée à des backends comme Cassandra ou HBase) permet de gérer efficacement des graphes contenant des millions de nœuds et d'arêtes, en assurant une **scalabilité horizontale** ainsi qu'une haute disponibilité.

D. Avantages et Contraintes des Approches NoSQL/Graph

D'un point de vue **mathématique**, l'utilisation d'une base orientée graphe permet de traiter les pondérations $\omega_{i,j}$ comme des composantes d'un opérateur linéaire défini sur l'espace des nœuds, facilitant l'application d'algorithmes de partitionnement ou de clustering. Les **algorithmes de détection de communautés** (par exemple, Louvain ou Label Propagation) peuvent directement exploiter la structure du graphe pour déterminer des clusters à différents niveaux de granularité.

Sur le plan **ingénierie**, l'utilisation d'une BDD NoSQL/Graph offre la possibilité de distribuer le stockage et le calcul, en permettant un **sharding** du graphe sur plusieurs machines. Toutefois, cette approche nécessite une phase de conception soignée pour définir les schémas d'indexation, les stratégies de réplication et les protocoles de mise à jour afin d'éviter des incohérences ou des retards importants dans les opérations d'écriture.

Par ailleurs, la nature même d'un graphe persistant implique que l'on puisse réaliser des **snapshots** ou des **logs** d'évolution, ce qui est particulièrement utile pour l'analyse rétroactive de la dynamique du SCN et la validation des hypothèses théoriques sur la convergence.

Conclusion

L'adoption d'approches NoSQL, et plus particulièrement l'utilisation de bases de données orientées graphe telles que **Neo4j** ou **JanusGraph/Titan**, représente une solution adaptée et pragmatique pour le stockage et la consultation des pondérations $\omega_{i,j}$ dans un SCN à grande échelle. Cette stratégie permet non seulement de modéliser naturellement le réseau sous forme de graphe (où chaque entité est un nœud et chaque connexion une arête), mais aussi de bénéficier d'algorithmes de requête et de détection de communautés optimisés pour ce type de données. Par ailleurs, ces bases offrent une scalabilité horizontale et une flexibilité d'intégration, ce qui est essentiel pour la gestion d'un SCN réparti sur plusieurs machines ou clusters. Sur le plan mathématique, la structure du graphe permet d'appliquer des méthodes analytiques et d'optimisation directement sur la matrice de pondérations, tout en assurant la cohérence et la persistance des données. Ainsi, ces approches facilitent l'extraction de clusters, la surveillance de l'évolution des liens et l'analyse

globale de la dynamique auto-organisée du SCN, en répondant aux exigences de performance et de flexibilité d'un système distribué moderne.

5.7.3.3. Équilibrage de Charge (Sharding de la Matrice ω)

Dans un **Synergistic Connection Network** (SCN) à très grande échelle, la matrice de pondérations Ω qui regroupe les connexions $\omega_{i,j}$ entre toutes les entités peut rapidement devenir volumineuse, avec une complexité mémoire de l'ordre de $O(n^2)$ lorsque le nombre d'entités n augmente. Pour pallier ce problème, il est souvent nécessaire de recourir à des techniques d'**équilibrage de charge** par le *sharding*, c'est-à-dire en partitionnant la matrice en fragments plus petits répartis sur plusieurs nœuds ou serveurs. Cette approche permet non seulement de réduire la charge de calcul et la pression sur la mémoire, mais également de réduire la latence lors de l'accès aux données et de faciliter la mise à l'échelle horizontale du système.

Sur le plan **mathématique**, on peut formaliser le sharding comme suit. Soit $\Omega \in \mathbb{R}^{n \times n}$ la matrice complète des pondérations, et supposons que nous partitionnons l'ensemble des entités $V = \{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ en m sous-ensembles disjoints, notés $\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_m$, de sorte que chaque sous-SCN SCN_p gère la matrice locale $\Omega^{(p)}$ de dimensions $|\mathcal{V}_p| \times |\mathcal{V}_p|$. Dans le cas idéal d'un partitionnement homogène, chaque bloc contient environ $\frac{n}{m}$ entités, et la taille de chaque matrice locale est alors de l'ordre de $O\left(\left(\frac{n}{m}\right)^2\right)$. Ainsi, le coût global de stockage pour les données internes aux shards est réduit à

$$m \times O\left(\left(\frac{n}{m}\right)^2\right) = O\left(\frac{n^2}{m}\right),$$

ce qui, pour un m suffisamment grand, constitue une amélioration significative par rapport à une gestion centralisée.

Le partitionnement par sharding améliore également l'**efficacité des requêtes**. Par exemple, lorsque l'on cherche à extraire les k plus fortes connexions pour une entité donnée, il est beaucoup plus rapide de consulter la ligne correspondante dans le shard local que d'effectuer un parcours exhaustif sur une matrice globale gigantesque. De plus, l'**accès local** aux données réduit la latence réseau et permet une meilleure exploitation du cache dans les environnements distribués.

Toutefois, le sharding introduit la problématique des **liaisons inter-shards**. Les connexions $\omega_{i,j}$ reliant des entités situées dans des shards différents (par exemple, $i \in \mathcal{V}_p$ et $j \in \mathcal{V}_q$ avec $p \neq q$) nécessitent une gestion particulière. La communication entre shards peut être réalisée de deux manières principales :

- Dans une première approche, chaque shard est responsable de stocker et de mettre à jour les liens pour lesquels il détient la ligne correspondante. Par exemple, on peut définir que le shard auquel appartient l'entité \mathcal{E}_i est le maître de toutes les pondérations $\omega_{i,j}$ pour tout j . Les autres shards doivent alors interroger ce maître pour obtenir la version la plus récente de ces liens.
- Dans une seconde approche, chaque shard maintient une **copie locale** des liens inter-shards, et un mécanisme de synchronisation périodique (par exemple, via des barrières synchrones ou des protocoles de type *gossip*) est mis en œuvre afin d'harmoniser les divergences éventuelles.

Sur le plan **ingénierie**, la méthode de partitionnement peut être réalisée via un **partitionnement par hachage** ou par des algorithmes de **graph partitioning** tels que METIS ou Kernighan–Lin. Le partitionnement par hachage répartit les entités de manière uniforme en utilisant, par exemple, l'opération $p = \text{hash}(i) \bmod m$. Le partitionnement par graphes, quant à lui, tente de minimiser le nombre de liens inter-shards en regroupant les entités fortement connectées dans le même fragment.

Les avantages du sharding sont multiples : il permet une **scalabilité horizontale** en répartissant la charge sur plusieurs serveurs, améliore la **localité** des données pour des requêtes plus rapides, et augmente la **résilience** du système, puisque la défaillance d'un shard n'entraîne pas nécessairement la panne de l'ensemble du réseau. Néanmoins, il faut prendre

garde aux risques d'**incohérence** entre les shards, ainsi qu'à la complexité supplémentaire induite par la gestion des mises à jour inter-shards.

En conclusion, l'équilibrage de charge par sharding de la matrice ω est une solution incontournable pour gérer des SCN de très grande échelle. La fragmentation de la matrice en blocs plus petits permet non seulement de réduire le coût mémoire et le temps de calcul par itération, mais aussi d'améliorer la réactivité du système en facilitant l'accès aux données locales. Cependant, cette approche nécessite une attention particulière à la synchronisation des mises à jour inter-shards et à la gestion de la communication entre les différentes partitions, afin de garantir la cohérence globale du SCN.

Exemple de mise en œuvre en Python (sans visualisation graphique)

Bien que nous nous concentrons ici sur l'explication théorique, il est pertinent d'illustrer brièvement comment l'on pourrait organiser le sharding de la matrice ω dans un environnement Python. Supposons que nous partitionnions nos entités en m shards et que nous stockions chaque shard sous forme d'un tableau NumPy. Une implémentation simple pourrait ressembler à :

```
import numpy as np

def initialize_shard(n, m, shard_index):
    """Initialise le shard pour le sous-ensemble d'entités du shard_index.
    n : nombre total d'entités
    m : nombre de shards
    shard_index : index du shard courant (0 <= shard_index < m)
    """
    # On suppose un partitionnement uniforme : chaque shard gère n/m entités
    entities_per_shard = n // m
    # Dimensions du shard: (entities_per_shard x n)
    # On stocke uniquement les lignes correspondantes à ce shard
    shard = np.zeros((entities_per_shard, n))
    return shard

def update_shard(shard, S, eta, tau, gamma):
    """Applique la règle de mise à jour sur le shard.
    S : matrice complète de synergie (n x n)
    shard : matrice du shard courant (entities_per_shard x n)
    Cette fonction met à jour chaque poids selon la formule additive avec inhibition.
    """
    entities_per_shard, n = shard.shape
    new_shard = np.copy(shard)
    for i in range(entities_per_shard):
        global_i = i # Dans un partitionnement uniforme, global_i = shard_index * entities_per_shard + i
        for j in range(n):
            # Calcul de la mise à jour additive
            delta = eta * (S[global_i, j] - tau * shard[i, j])
            # Ajout du terme d'inhibition : somme sur k != j
            inhibition = gamma * (np.sum(shard[i, :]) - shard[i, j])
            new_shard[i, j] = shard[i, j] + delta - inhibition
    return new_shard

# Paramètres
n = 1000 # nombre total d'entités
m = 10 # nombre de shards
eta = 0.05
tau = 1.0
gamma = 0.01
```

```

# Initialisation de la matrice de synergie S (pour l'exemple, valeurs aléatoires)
np.random.seed(42)
S = np.random.uniform(0, 1, (n, n))

# Initialisation des shards
shards = [initialize_shard(n, m, p) for p in range(m)]

# Mise à jour sur plusieurs itérations (par exemple, 100 itérations)
iterations = 100
for t in range(iterations):
    for p in range(m):
        shards[p] = update_shard(shards[p], S, eta, tau, gamma)
    # Ici, on pourrait synchroniser les mises à jour inter-shards si nécessaire (non implémenté)

# À ce stade, chaque shard contient sa partie mise à jour de la matrice  $\omega$ .

```

Ce code illustre l'approche de **sharding** où chaque shard gère un sous-ensemble des lignes de la matrice ω . Bien que cet exemple ne couvre pas la synchronisation des liaisons inter-shards, il démontre la manière dont la mise à jour locale est effectuée sur chaque fragment, réduisant ainsi la charge de calcul par rapport à une matrice centralisée. Dans une application réelle, des mécanismes supplémentaires seraient intégrés pour synchroniser les valeurs des liens inter-blocs et pour assurer une cohérence globale.

En synthèse, l'équilibrage de charge par sharding permet de gérer efficacement des SCN à grande échelle en divisant la matrice ω en blocs plus petits, améliorant ainsi la scalabilité, la localité des accès et la résilience du système tout en maintenant la dynamique mathématique fondamentale du réseau.

5.8. Sécurité, Fiabilité et Vérifications

5.8.1. Vulnérabilités au Bruit ou à l'Attaque

- 5.8.1.1. Si un agent malveillant manipule $\omega_{i,j}$, risque de clusters artificiels ou sabotage de la dynamique.
- 5.8.1.2. Solutions : logs, watchers, checks d'anomalies.

5.8.2. Robustesse Face aux Pannes

- 5.8.2.1. Quid si un module SCN tombe ? le réseau peut-il se reconfigurer ?
- 5.8.2.2. Basculement automatique ou redondance des données ω .

5.8.3. Contrôle d'Intégrité

- 5.8.3.1. Détection de liens aberrants (trop forts trop vite).
- 5.8.3.2. Calcul d'indicateurs de cohérence globale (ex. distribution statistique des ω).

5.8. Sécurité, Fiabilité et Vérifications

Dans une optique de **grande échelle** et de **distribution** (cf. sections précédentes), la sécurité et la fiabilité d'un SCN (Synergistic Connection Network) ne sauraient être laissées de côté, en particulier lorsqu'on envisage des scénarios collaboratifs ou ouverts (accès par divers agents, partage de données, etc.). Le chapitre 5.8 vise à souligner les problématiques de **vulnérabilité** face à des attaques ou manipulations (5.8.1), la robustesse en cas de pannes (5.8.2) et la question du contrôle d'intégrité (5.8.3). Même si l'on se concentre principalement sur l'aspect mathématique et théorique, il est crucial de prévoir des mécanismes de sécurité et de vérification pour un SCN qui puisse fonctionner de façon fiable dans des environnements potentiellement hostiles ou imprévisibles.

5.8.1. Vulnérabilités au Bruit ou à l'Attaque

Dans un SCN largement ouvert — et plus encore s'il est distribué (5.7) —, les entités ou les liens peuvent être soumis à des perturbations extérieures : bruit injecté par le système lui-même, agents malveillants, données corrompues. La sous-section (5.8.1.1) évoque les risques d'attaques et de sabotage, puis (5.8.1.2) propose quelques contre-mesures (logs, watchers, checks d'anomalies, etc.).

5.8.1.1. Si un agent malveillant manipule $\omega_{i,j}$, risque de clusters artificiels ou sabotage de la dynamique

Dans le cadre d'un **Synergistic Connection Network** (SCN), la robustesse de la formation des *clusters* repose sur la mise à jour itérative et locale des pondérations $\omega_{i,j}$ par l'équation

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

où η est le **taux d'apprentissage** et τ le **coefficient de décroissance**. Ce mécanisme vise à renforcer les liaisons correspondant à une **synergie** élevée entre entités tout en assurant une régulation préventive d'une croissance non contrôlée. Cependant, si un **agent malveillant** parvient à intervenir sur $\omega_{i,j}$ ou sur le calcul de $S(i,j)$, il est possible d'injecter des perturbations qui peuvent induire la formation de *clusters artificiels* ou, inversement, empêcher l'émergence des structures attendues, compromettant ainsi la **stabilité** et la **légitimité** du SCN.

A. Enjeu Mathématique et Opérationnel

La dynamique de mise à jour, symbolisée par la fonction F définie par

$$F(\omega_{i,j}(t), S(i,j)) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

est au cœur de la formation des clusters. La **convergence** vers un état stable se traduit par l'atteinte d'un équilibre approximatif $\omega_{i,j}^*$ tel que

$$\omega_{i,j}^* \approx \frac{S(i,j)}{\tau}.$$

L'**intégrité** de cette dynamique dépend de la fiabilité des valeurs de $S(i,j)$ et de $\omega_{i,j}$. Un attaquant qui intervient sur ces paramètres peut, par exemple, **injecter** des valeurs anormalement élevées dans certains éléments de la matrice ω ou altérer le module de calcul de $S(i,j)$ afin de renvoyer des valeurs décalées. En conséquence, l'agent malveillant peut fausser la perception de la *cohérence* entre entités, menant à la formation de **clusters fictifs** — où certaines entités paraissent artificiellement surconnectées — ou, au contraire, paralyser la capacité d'auto-organisation du réseau en annulant les liens essentiels.

Sur le plan **opérationnel**, de telles manipulations peuvent compromettre des systèmes critiques, tels que des algorithmes de recommandation, la coordination de robots ou encore des systèmes de détection d'anomalies, dans

lesquels la configuration de $\omega_{i,j}$ reflète la structure réelle et utile des données. Une altération des pondérations conduit ainsi à une perte de **fiabilité** et de **prédictibilité** dans l'évolution du réseau.

B. Scénarios d'Attaques Possibles

Plusieurs vecteurs d'attaque peuvent être envisagés pour manipuler les valeurs de $\omega_{i,j}$ ou de $S(i,j)$:

Dans ce scénario, un **agent interne** ou une entité ayant acquis un accès privilégié au système modifie directement la matrice des pondérations. Les actions possibles incluent :

- **Gonflement artificiel** : en assignant par exemple $\omega_{i,j} = 100$ pour certaines paires (i,j) , l'attaquant force la dynamique à renforcer de manière excessive des liaisons qui ne reflètent pas la réalité de la synergie entre entités. Ce gonflement crée des *clusters* artificiels, induisant une surconnexion entre des entités qui, en conditions normales, ne devraient pas être regroupées.
- **Affaiblissement ciblé** : en réduisant à zéro ou en diminuant fortement des liens cruciaux pour la cohésion du réseau, l'attaquant peut empêcher la formation de clusters réellement cohésifs, sabotant ainsi le processus d'auto-organisation.

Une autre approche consiste à intercepter ou à altérer le module de calcul de la **synergie**. Par exemple, si le composant $S(i,j)$ est implémenté dans un module nommé *synergyCalculator*, une corruption de ce module peut renvoyer des valeurs exagérément élevées (par exemple $S(i,j) = 10^5$) ou des valeurs très faibles, indépendamment de la relation réelle entre \mathcal{E}_i et \mathcal{E}_j . Cette falsification trompe la mise à jour locale de $\omega_{i,j}$ et conduit le réseau à renforcer des connexions basées sur des indicateurs erronés.

Dans des architectures distribuées, où plusieurs sous-SCN communiquent pour mettre à jour la matrice globale, l'attaquant peut interférer avec la transmission des messages. Les stratégies comprennent :

- **Spoofing des messages** : l'envoi de fausses informations sur les valeurs de $\omega_{i,j}$ ou la synergie calculée, induisant une mise à jour erronée dans les différents nœuds du réseau.
- **Retard ou suppression** : en retardant la transmission ou en supprimant certains messages inter-blocs, l'attaquant crée des incohérences temporelles qui perturbent la convergence du système.

C. Impact sur la Dynamique

Lorsque des valeurs de $\omega_{i,j}$ sont artificiellement gonflées, la dynamique du SCN tend à regrouper les entités associées dans un même cluster, même si elles n'ont pas de synergie naturelle élevée. Le mécanisme de mise à jour renforce ces liaisons aberrantes, menant à l'émergence de *clusters fictifs* qui ne reflètent pas la structure intrinsèque des données. Par exemple, dans un système de recommandation, un groupe d'utilisateurs pourrait être rassemblé artificiellement, faussant ainsi les résultats et la pertinence des recommandations.

Inversement, la réduction ou l'annulation de certains liens cruciaux par l'attaquant empêche la consolidation de clusters légitimes. Le résultat peut être une oscillation continue des valeurs de $\omega_{i,j}$ ou une dispersion générale qui rend impossible l'atteinte d'un état d'équilibre. Dans un contexte de robotique ou de systèmes d'intelligence collective, une telle perturbation conduit à un manque de coordination et à une incapacité d'atteindre une **stabilité opérationnelle**.

Les preuves mathématiques assurant la convergence de la dynamique du SCN reposent sur l'hypothèse d'un comportement aléatoire (et non adversarial) dans la perturbation des pondérations. En présence d'une **perturbation dirigée**, la fonction de mise à jour

$$\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i,j))$$

se voit remplacée par

$$\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i,j)) + \text{Perturbation}_{\text{adv}}(i,j,t),$$

où $\text{Perturbation}_{\text{adv}}(i, j, t)$ est choisie par l'attaquant pour maximiser son effet destructeur. Ainsi, les garanties de stabilité, de convergence et de formation de clusters sont compromises, rendant l'analyse théorique du SCN caduque en conditions adversariales.

D. Dimension Mathématique : Perturbation Adversariale de F

D'un point de vue mathématique, l'attaque peut être formalisée par l'introduction d'un **terme perturbateur** dans l'opérateur de mise à jour. Ainsi, au lieu d'avoir

$$\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i, j)),$$

on considère que

$$\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i, j)) + \Delta_{\text{adv}}(i, j, t),$$

où

$$\Delta_{\text{adv}}(i, j, t) = \text{Perturbation}_{\text{adv}}(i, j, t)$$

représente l'action malveillante. Si cette perturbation est suffisamment grande et choisie de manière stratégique, elle peut modifier la trajectoire dynamique de $\omega_{i,j}(t)$ au point de rendre la convergence vers $\omega_{i,j}^* \approx \frac{S(i,j)}{\tau}$ impossible. En d'autres termes, la présence de Δ_{adv} introduit une **incertitude** et une **instabilité** qui empêchent la formation d'un état stable, rendant ainsi le SCN vulnérable à des modifications arbitraires de sa topologie.

Conclusion

En résumé, un **agent malveillant** qui manipule les pondérations $\omega_{i,j}$ ou falsifie les valeurs de synergie $S(i, j)$ peut gravement compromettre la dynamique d'un SCN. Une injection artificielle de valeurs élevées entraîne la création de *clusters artificiels* dans lesquels les entités se retrouvent surconnectées sans réelle justification, tandis que l'annulation ou l'affaiblissement de liens essentiels sabote la formation de clusters cohésifs, conduisant à une instabilité générale du réseau. Mathématiquement, cela se traduit par l'introduction d'un terme perturbateur dans l'opérateur de mise à jour, annulant ainsi les garanties théoriques de convergence et de stabilité sur lesquelles repose le modèle.

Sur le plan opérationnel, de telles attaques représentent une menace majeure pour les systèmes critiques reposant sur le SCN (voir également les sections 5.7.1 et 5.7.2), car elles compromettent la **fiabilité** et l'**intégrité** du processus d'auto-organisation. Les contre-mesures, qui seront détaillées dans les sections ultérieures (5.8.1.2, 5.8.2, etc.), incluent l'implémentation de protocoles cryptographiques, la vérification de la cohérence des logs et des mécanismes de redondance pour limiter l'impact d'une perturbation adversariale.

En définitive, la robustesse du SCN face à des attaques ciblées sur $\omega_{i,j}$ constitue un enjeu crucial tant du point de vue théorique que pour la mise en œuvre pratique de systèmes d'**intelligence collective** et d'**auto-organisation**.

5.8.1.2. Solutions : Logs, Watchers, Checks d'Anomalies

La robustesse d'un **Synergistic Connection Network** (SCN) face aux attaques malveillantes, telles que celles décrites en **Section 5.8.1.1**, repose sur la mise en place de mécanismes complémentaires destinés à surveiller, tracer et valider en temps réel la dynamique de mise à jour des pondérations $\omega_{i,j}$. Bien que ces solutions ne modifient pas le **cœur mathématique** du modèle – défini par l'équation de mise à jour

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)],$$

– elles jouent un rôle essentiel pour **fiabiliser** l'auto-organisation en introduisant une surcouche de sécurité et de contrôle. Nous détaillons ci-après trois axes complémentaires de défense : la **traçabilité** via les journaux (logs), la surveillance en temps réel par des **watchers** et l'utilisation d'algorithmes de **détection d'anomalies**.

A. Journaux (Logs) et Traçabilité

La **traçabilité** est une condition indispensable pour détecter toute modification suspecte dans la dynamique du SCN. En enregistrant chaque événement de mise à jour des pondérations, il est possible de reconstituer l'historique complet de l'évolution de $\omega_{i,j}$. Pour ce faire, chaque mise à jour est loggée avec des métadonnées détaillées, telles que :

- **Identifiants** des entités concernées, (i, j) ,
- La **variation** appliquée, notée
$$\Delta\omega_{i,j}(t) = \omega_{i,j}(t+1) - \omega_{i,j}(t),$$
- Le **timestamp** de la mise à jour,
- L'**ID de l'agent** ou du composant initiateur de la mise à jour,
- Des **paramètres contextuels** (valeur calculée de la synergie $S(i, j)$, paramètres η et τ , etc.),
- Une **signature cryptographique** ou un *hash* du message, par exemple

$$H(i, j, t) = \text{Hash}(i, j, \omega_{i,j}(t), \Delta\omega_{i,j}(t), \text{timestamp}),$$

Ces **logs** constituent un registre immuable qui peut être exploité postérieurement pour une **analyse forensique**. Leur utilité est double : ils permettent d'identifier en temps différé des anomalies (par exemple, une variation exceptionnelle telle que $\Delta\omega_{i,j}(t)$ passant de 0.2 à 100 en une seule itération) et offrent une base pour **corrélér** les événements avec les actions d'un potentiel attaquant. Dans un contexte distribué, la centralisation ou la synchronisation des logs via des solutions de stockage sécurisées (bases NoSQL, systèmes de fichiers distribués avec intégrité garantie) assure une **traçabilité** à l'échelle du réseau.

B. Watchers et Monitoring en Temps Réel

Les **watchers** sont des modules de surveillance intégrés dans la boucle de mise à jour des pondérations. Leur rôle est de **filtrer** en temps réel les variations de $\omega_{i,j}$ afin de détecter immédiatement toute anomalie qui pourrait résulter d'une action malveillante. Mathématiquement, on peut considérer un Watcher comme une fonction de contrôle W appliquée à chaque mise à jour :

$$W(\Delta\omega_{i,j}(t)) = \begin{cases} 1, & \text{si } |\Delta\omega_{i,j}(t)| > \kappa, \\ 0, & \text{sinon,} \end{cases}$$

où κ est un **seuil critique** défini en fonction de la dynamique attendue du SCN. Lorsqu'une condition d'alerte est satisfaite – par exemple, si

$$|\omega_{i,j}(t+1) - \omega_{i,j}(t)| > \kappa \quad \text{ou} \quad \omega_{i,j}(t+1) > \omega_{\max},$$

– le Watcher déclenche une **alerte en temps réel**. Cette alerte peut conduire à diverses actions immédiates :

- **Blocage** temporaire de la mise à jour pour le lien concerné,
- **Demande de validation manuelle** ou automatisée via des protocoles d'authentification renforcée,
- **Transmission** d'un signal de sécurité à un module central de surveillance ou à d'autres nœuds du réseau pour une **réaction collective**.

En contexte distribué, plusieurs Watchers peuvent coopérer et partager leurs observations via un protocole de consensus, assurant ainsi une **vérification mutuelle** des mises à jour anormales avant qu'elles ne perturbent la dynamique globale.

C. Checks d'Anomalies et Méthodes Avancées

Au-delà des contrôles immédiats et locaux effectués par les Watchers, il est possible d'implémenter des algorithmes d'**anomaly detection** qui s'intéressent à la **distribution** globale des pondérations et à l'évolution des **clusters**. Ces méthodes avancées incluent :

Une approche consiste à surveiller des métriques statistiques sur l'ensemble des pondérations. Par exemple, en calculant la **variance** σ^2 et l'**entropie** H de la distribution des $\omega_{i,j}$:

$$\sigma^2 = \frac{1}{N} \sum_{i < j} (\omega_{i,j} - \bar{\omega})^2, \quad H = - \sum_{i < j} p(\omega_{i,j}) \ln p(\omega_{i,j}),$$

où $\bar{\omega}$ est la moyenne des $\omega_{i,j}$ et $p(\omega_{i,j})$ la distribution empirique des poids. Une variation brutale de σ^2 ou une chute/incrément inattendu de H par rapport aux valeurs historiques peut être le signe d'une **injection malveillante**.

En complément de l'analyse statistique, l'observation de la **topologie** des clusters peut révéler des configurations anormales. Par exemple, l'apparition soudaine d'un cluster isolé dont la cohésion interne serait extrêmement élevée, ou au contraire, la disparition d'un cluster connu, peut être détectée par des techniques de **clustering supervisé ou non supervisé**. On peut alors définir un indice de **cohérence cluster** C tel que :

$$C = \frac{1}{|G|} \sum_{(i,j) \in G} \omega_{i,j},$$

où G désigne l'ensemble des paires au sein d'un cluster donné. Une valeur de C anormalement haute ou basse par rapport aux attentes théoriques indique un dysfonctionnement, pouvant être imputé à une attaque.

Lorsque des anomalies sont détectées, il est impératif de disposer de **mécanismes de réaction** qui permettent de limiter l'impact d'une attaque. Parmi ceux-ci, on peut citer :

- Le **rollback**, consistant à rétablir les valeurs antérieures de $\omega_{i,j}$ issues des logs jugées fiables,
- La mise en quarantaine des nœuds ou des liens suspectés d'avoir été altérés,
- La déclinaison de procédures de **vérification renforcée** (p.ex. via des signatures numériques ou des protocoles de consensus) pour valider les mises à jour futures.

Ces politiques de réaction reposent sur une modélisation mathématique du problème sous la forme d'un opérateur de perturbation, où l'on cherche à minimiser l'écart entre la trajectoire observée et la trajectoire attendue définie par $F(\omega(t))$.

Conclusion

Les mesures de **sécurité** déployées autour du SCN – incluant la **consignation détaillée des logs**, la **surveillance en temps réel** via des Watchers et l'implémentation d'algorithmes de **détection d'anomalies** – constituent une couche de défense essentielle pour garantir la **fiabilité** et la **stabilité** de la dynamique d'auto-organisation. Du point de vue **mathématique**, ces solutions n'altèrent pas la fonction de mise à jour fondamentale $\omega(t+1) = F(\omega(t))$ mais introduisent un mécanisme de contrôle externe qui valide ou rejette les évolutions des pondérations en fonction de critères prédéfinis. Du point de vue **opérationnel**, elles permettent de détecter et d'intervenir rapidement en cas de comportement aberrant, réduisant ainsi le risque de formation de clusters artificiels ou de sabotage de la convergence du SCN. Ces solutions, intégrées de manière cohérente dans une architecture distribuée (cf. Sections 5.7 et 5.8), renforcent la **confiance** dans le système et permettent d'en assurer la résilience face à des attaques potentielles dans des environnements critiques.

5.8.2. Robustesse Face aux Pannes

Même lorsque le **SCN** (Synergistic Connection Network) est doté de mécanismes de sécurité (5.8.1) et d'une architecture soigneusement construite (5.7), il subsiste toujours la possibilité de **pannes** : un sous-SCN peut cesser de fonctionner, un serveur peut tomber en panne, ou un module logiciel critique peut se bloquer. La robustesse d'un système à ces défaillances est alors un point essentiel : comment assurer que le **réseau** global ne s'écroule pas et qu'il puisse, si nécessaire, se **reconfigurer** pour continuer de fonctionner ? Dans cette section (5.8.2), nous examinons d'abord (5.8.2.1) les scénarios de panne d'un module SCN et la possibilité de reconfiguration, puis (5.8.2.2) discute les approches de basculement automatique et de redondance.

5.8.2.1. Quid si un module SCN tombe ? Le réseau peut-il se reconfigurer ?

Dans un **Synergistic Connection Network** (SCN) distribué, la robustesse et la résilience du système dépendent de sa capacité à absorber les perturbations, notamment lorsque l'un des modules ou sous-réseaux, noté SCN_p , tombe en panne. Un tel module gère un sous-ensemble d'entités, \mathcal{V}_p , ainsi que les liaisons internes associées, c'est-à-dire l'ensemble $\{\omega_{i,j} \mid i, j \in \mathcal{V}_p\}$. La question se pose alors de savoir si le réseau global peut se reconfigurer pour continuer à fonctionner malgré la perte partielle ou totale d'un module, et quelles stratégies – tant du point de vue opérationnel que mathématique – permettent d'assurer une telle résilience.

A. Scénarios de Panne dans un SCN Distribué

Dans un système décomposé en plusieurs modules SCN_1, \dots, SCN_m , les pannes peuvent se manifester de diverses manières, affectant différemment la dynamique globale. Nous distinguons principalement trois scénarios :

Tout d'abord, dans le **cas mineur**, la panne d'un module SCN_p est de courte durée. Par exemple, un crash temporaire dû à une défaillance matérielle ou à un incident réseau provoque l'arrêt de la mise à jour des liaisons concernant les entités \mathcal{V}_p pendant une période limitée. Pendant ce temps, le reste du SCN poursuit son évolution selon la dynamique standard

$$\omega(t+1) = F(\omega(t)) \quad \text{où} \quad F(\omega(t)) = \omega(t) + \eta [S - \tau \omega(t)].$$

Les liens impliquant \mathcal{V}_p peuvent être alors mis en veille ou temporairement ignorés. Une fois le module rétabli, une phase de **re-synchronisation** est nécessaire pour réintégrer les entités affectées dans le calcul global, en récupérant les valeurs sauvegardées (logs, snapshots) afin de minimiser la rupture de la dynamique.

Ensuite, dans le **cas majeur**, la panne est définitive. On peut alors envisager deux issues complémentaires : soit les données associées à \mathcal{V}_p sont irrémédiablement perdues, soit il est nécessaire de migrer ces entités vers un autre module SCN_q . Dans ce dernier cas, la réaffectation implique une **réorganisation de la partition** initiale $\{\mathcal{V}_1, \dots, \mathcal{V}_m\}$ de l'ensemble des entités. Concrètement, les lignes et colonnes de la matrice ω correspondant aux indices de \mathcal{V}_p sont soit supprimées, soit transférées dans un nouvel espace de calcul, ce qui se traduit par une modification structurelle de la dynamique.

Enfin, un **cas hybride** peut survenir lorsqu'une panne initialement mineure se prolonge, rendant la réintégration problématique. Dans ce scénario, la solution de reconfiguration doit prendre en compte la durée d'inactivité et les conséquences sur la **topologie** du réseau, en adoptant par exemple une approche hybride qui combine le gel temporaire des liaisons avec une éventuelle réallocation progressive des entités affectées.

B. Reconfiguration ou Dégradation du SCN

La résilience du réseau repose sur sa capacité à se reconfigurer en réponse à la perte d'un module. Deux approches principales peuvent être envisagées :

Dans le cas d'une panne définitive ou prolongée, la stratégie la plus souhaitable est la **migration** des entités \mathcal{V}_p vers un autre module SCN_q . Cette procédure de réaffectation s'effectue en plusieurs étapes :

- **Sauvegarde et restauration** : Si des mécanismes de backup ou des logs détaillés existent, ils permettent de reconstruire l'historique des mises à jour des liaisons $\omega_{i,j}$ pour $i, j \in \mathcal{V}_p$. On dispose alors d'une base de données pour réintégrer ces valeurs dans le nouveau module.
- **Réaffectation de la partition** : La partition initiale $\{\mathcal{V}_1, \dots, \mathcal{V}_m\}$ est redéfinie en intégrant les entités de \mathcal{V}_p dans un autre sous-ensemble, par exemple en formant un nouvel ensemble $\mathcal{V}_q' = \mathcal{V}_q \cup \mathcal{V}_p$. La mise à jour de la dynamique se fait alors sur une matrice ω' de dimension réduite ou réallouée, avec
$$\omega': \mathcal{V}_{\text{nouvelle}} \times \mathcal{V}_{\text{nouvelle}} \rightarrow \mathbb{R}^+.$$
- **Re-synchronisation inter-blocs** : Une fois les entités migrées, il est crucial que les liaisons inter-blocs soient actualisées afin de restaurer la cohérence globale. Ceci peut être réalisé par un protocole de synchronisation qui met à jour les valeurs de $S(i, j)$ pour les nouveaux liens entre \mathcal{V}_q' et les autres sous-ensembles.

Si la panne est de courte durée, il est souvent préférable d'adopter une **stratégie de dégradation** temporaire. Dans ce cas, les entités appartenant à \mathcal{V}_p sont mises hors circuit, et le SCN global continue à fonctionner en excluant les mises à jour associées à ce bloc. Mathématiquement, cela revient à définir une fonction indicatrice I_p telle que :

$$I_p(i) = \begin{cases} 0, & \text{si } i \in \mathcal{V}_p, \\ 1, & \text{sinon.} \end{cases}$$

La dynamique de mise à jour est alors adaptée de la forme

$$\omega_{i,j}(t+1) = I_p(i)I_p(j) \left[\omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)] \right],$$

ce qui garantit que les liens impliquant des entités hors ligne ne perturbent pas la convergence du réseau. Lorsque le module défaillant revient en ligne, un protocole de **reconnexion** permet de réintégrer progressivement les entités concernées dans la dynamique globale.

C. Analyse Mathématique de la Panne

D'un point de vue formel, la panne d'un module SCN se traduit par la suppression des lignes et colonnes correspondant aux entités \mathcal{V}_p dans la matrice des pondérations ω . Soit $\omega(t)$ une matrice $N \times N$ décrivant la dynamique initiale, et soit $P \subset \{1, \dots, N\}$ l'ensemble des indices associés à \mathcal{V}_p . La panne entraîne la considération d'une nouvelle matrice réduite

$$\omega_{\text{réduit}}(t) = (\omega_{i,j}(t))_{i,j \notin P}.$$

La dynamique de mise à jour s'exprime alors par

$$\omega_{\text{réduit}}(t+1) = \omega_{\text{réduit}}(t) + \eta [S_{\text{réduit}} - \tau \omega_{\text{réduit}}(t)],$$

où $S_{\text{réduit}}$ désigne la restriction de la fonction de synergie aux entités encore actives. Plusieurs questions se posent à ce stade :

- **Robustesse de la convergence** : Les théorèmes de convergence établissant que $\omega(t)$ tend vers un équilibre de la forme $\omega^* \approx S/\tau$ reposent sur la structure complète du réseau. La suppression d'un sous-ensemble de nœuds représente une perturbation du système dynamique qui doit être analysée à l'aide des théories de la robustesse des systèmes dynamiques et de la résilience des graphes.
- **Impact sur la connectivité** : La disparition d'un bloc peut modifier la **connectivité** globale du SCN, affectant ainsi la capacité du réseau à former des clusters cohérents. Des résultats de théorie des graphes indiquent que la suppression d'un sous-ensemble de nœuds (ou de liens) peut, dans le pire des cas, décomposer le graphe en plusieurs composantes, mais si le réseau initial possède une **redondance** suffisante, la connectivité résiduelle permettra au système de continuer à converger de manière partielle.
- **Temps de re-synchronisation** : La migration des entités vers un autre module ou le retour d'un module en panne induit un délai pendant lequel la matrice ω évolue sous une topologie modifiée. Il est alors nécessaire

d'étudier le temps de convergence de la dynamique modifiée, qui peut être exprimé en fonction des paramètres η et τ , ainsi que de la proportion d'entités affectées.

Ces considérations mathématiques traduisent le fait que, bien que la disparition d'un module perturbe la dynamique initiale, si la structure du réseau est suffisamment redondante et si des mécanismes de reconfiguration sont en place, le système peut retrouver un nouvel état d'équilibre qui, bien que différent de celui initial, reste fonctionnel.

Conclusion

Lorsqu'un module SCN, SCN_p , tombe en panne dans un environnement distribué, le système global se trouve confronté à une **perturbation structurelle** importante. Trois scénarios principaux se dégagent : la dégradation temporaire du réseau (où les entités de \mathcal{V}_p sont mises hors ligne), la reconfiguration par migration des entités vers d'autres modules, ou la perte définitive des entités affectées. D'un point de vue mathématique, cela correspond à une réduction de dimension dans la matrice ω et à une réinitialisation partielle de la dynamique $\omega(t + 1) = F(\omega(t))$.

Les stratégies de réallocation – telles que la migration vers un autre bloc, la réorganisation de la partition ou le maintien d'un mode dégradé – permettent de préserver, dans la mesure du possible, la capacité du réseau à converger et à former des clusters cohérents malgré la disparition d'un sous-ensemble d'entités. La clé réside dans la redondance et dans la rapidité d'intervention pour rétablir la connectivité inter-blocs. Ainsi, le SCN, conçu pour être résilient, peut se reconfigurer et continuer à évoluer, assurant ainsi la continuité des services même en présence de défaillances matérielles ou logicielles.

En définitive, la gestion des pannes dans un SCN distribué repose sur un équilibre délicat entre la **reconfiguration dynamique** des sous-ensembles d'entités et la **préservation** des propriétés de convergence et de stabilité du système global. Les mécanismes de sauvegarde, de migration et de synchronisation constituent autant d'outils indispensables pour garantir la résilience d'un réseau de grande envergure dans des environnements potentiellement hostiles.

5.8.2.2. Basculement Automatique ou Redondance des Données ω

Dans un **Synergistic Connection Network** (SCN) distribué, la dynamique des pondérations $\omega(t)$ – régie par la mise à jour itérative

$$\omega(t + 1) = F(\omega(t)) = \omega(t) + \eta [S - \tau \omega(t)]$$

– doit continuer à évoluer de manière cohérente même en cas de défaillance d'un sous-réseau ou d'un microservice. Pour ce faire, il est essentiel d'implémenter des mécanismes de **basculement automatique** (failover) et de **redondance** (réplication) des données ω . Ces dispositifs, couramment utilisés en ingénierie des systèmes distribués, permettent d'assurer la continuité de l'auto-organisation et la résilience du SCN, en minimisant l'impact d'une panne prolongée ou définitive d'un module.

A. Basculement Automatique (Failover)

Le basculement automatique, ou failover, se définit comme le processus par lequel la responsabilité d'un module défaillant est transférée à un nœud opérationnel afin de garantir la disponibilité du service. Dans le contexte d'un SCN composé de plusieurs sous-réseaux SCN_1, \dots, SCN_m , chaque sous-ensemble d'entités \mathcal{V}_p est géré par son bloc dédié. Lorsque le module SCN_p devient injoignable en raison d'une panne (causée par exemple par un crash matériel, un problème de réseau ou une défaillance logicielle), un orchestrateur de surveillance, qui utilise des signaux de type « heartbeat », détecte l'absence de réponse et déclenche un mécanisme de failover.

Ce mécanisme repose sur la re-partition de l'ensemble des entités initialement réparties selon la collection $\{\mathcal{V}_1, \dots, \mathcal{V}_m\}$. Dès lors, les entités de \mathcal{V}_p sont transférées vers un autre bloc opérationnel, par exemple SCN_q . Mathématiquement, cela revient à redéfinir la partition du système en posant, pour un basculement réussi,

$$\mathcal{V}'_q = \mathcal{V}_q \cup \mathcal{V}_p,$$

et à mettre à jour la dynamique locale en appliquant la fonction F sur la nouvelle matrice de pondérations $\omega'(t)$ qui est reconstruite à partir de la réunion des données des blocs concernés. En pratique, cette opération nécessite que les informations relatives aux pondérations $\omega_{i,j}$ pour $i, j \in \mathcal{V}_p$ soient immédiatement accessibles via un système de stockage redondant. La transparence de ce mécanisme repose sur la rapidité avec laquelle l'orchestrateur détecte la panne et redirige les flux de données, de manière à ce que la continuité de la dynamique globale ne soit que temporairement perturbée.

B. Redondance (Réplication) des Données ω

La redondance des données constitue le second pilier de la résilience d'un SCN. Sans réplication, la panne d'un module entraînerait la perte définitive des pondérations associées aux entités de ce module, rendant impossible toute opération de failover. La réplication consiste à stocker simultanément plusieurs copies des données critiques, en l'occurrence les valeurs de ω , sur des nœuds ou dans des clusters de stockage distribués.

Plusieurs stratégies de réplication peuvent être mises en œuvre :

D'abord, la stratégie **N-Way** consiste à répliquer chaque portion de la matrice ω sur N nœuds distincts. Formellement, pour chaque lien $\omega_{i,j}$ lié aux entités d'un sous-ensemble \mathcal{V}_p , on maintient N copies, notées $\omega_{i,j}^{(1)}, \omega_{i,j}^{(2)}, \dots, \omega_{i,j}^{(N)}$, telles que la mise à jour de l'une ou plusieurs d'entre elles garantit la disponibilité de l'information en cas de défaillance d'un nœud.

Une autre approche courante est le modèle **Master-Slave**, dans lequel un nœud maître détient la version actuelle et définitive de $\omega_{i,j}$ pour un sous-ensemble donné, et plusieurs nœuds esclaves reçoivent les mises à jour en quasi temps réel. En cas de défaillance du maître, l'un des esclaves peut prendre le relais, en assurant la continuité du service. On peut représenter ce mécanisme par l'équation de synchronisation suivante : si $\omega_{i,j}^{(M)}(t)$ désigne la version maître et $\omega_{i,j}^{(S)}(t)$ la version esclave, alors

$$\omega_{i,j}^{(S)}(t) = \omega_{i,j}^{(M)}(t) + \delta_{i,j}(t),$$

où $\delta_{i,j}(t)$ est un terme de décalage que l'on s'efforce de maintenir très faible grâce à des protocoles de consensus (par exemple, via le protocole Paxos ou Raft).

Enfin, des méthodes telles que **Erasur Coding** permettent d'optimiser l'espace de stockage tout en garantissant la possibilité de reconstruire la matrice ω en cas de perte de certaines portions de données. Dans ce cadre, ω est codée en plusieurs fragments, dont un nombre suffisant permet de reconstituer l'information d'origine, suivant un schéma de correction d'erreurs.

C. Conséquences sur la Continuité de la Dynamique

L'intégration de mécanismes de basculement automatique et de redondance a pour objectif fondamental de permettre au SCN de poursuivre sa dynamique d'auto-organisation malgré la perte ou l'inaccessibilité temporaire d'un module. Lorsqu'un basculement est déclenché, la reconfiguration du réseau se traduit par une mise à jour de la partition des entités et par la restauration des pondérations via les copies redondantes, de sorte que la dynamique

$$\omega(t+1) = F(\omega(t))$$

peut reprendre son évolution sans perte significative de la structure initialement acquise.

Le basculement automatique assure que, dès qu'un module défaillant est détecté, un nouveau nœud héberge les entités concernées et continue la mise à jour. La redondance des données ω garantit quant à elle que l'historique des liaisons, et par extension la « mémoire » du réseau, est préservée. Même si une légère asynchronie peut être introduite – en raison de délais dans la synchronisation des copies – des protocoles d'agrégation et de réconciliation (par exemple, l'utilisation de vecteurs d'horodatage ou de mécanismes de fusion d'updates) permettent de minimiser ces écarts et d'assurer la cohérence de la dynamique.

L'architecture résiliente ainsi obtenue permet au SCN de s'adapter à des environnements imprévisibles en maintenant une **convergence partielle** ou, idéalement, une **reconvergence** vers un nouvel équilibre qui préserve les propriétés essentielles de l'auto-organisation, notamment la formation de clusters et la stabilité de la dynamique.

Conclusion

Le basculement automatique et la redondance des données ω constituent des mécanismes indispensables pour assurer la résilience d'un SCN distribué. Grâce au basculement (failover), le système peut rediriger instantanément les entités d'un module défaillant vers un autre bloc opérationnel, modifiant ainsi dynamiquement la partition des entités tout en préservant la continuité des mises à jour. Parallèlement, la réplication ou la redondance des pondérations garantit que, même en cas de panne, l'information essentielle sur la dynamique des liaisons est conservée et peut être rapidement restaurée. Du point de vue mathématique, ces mécanismes n'altèrent pas la forme fondamentale de la mise à jour

$$\omega(t + 1) = F(\omega(t)),$$

mais assurent que cette dynamique puisse se poursuivre de manière fiable malgré des défaillances ponctuelles ou prolongées. En somme, ces stratégies de basculement et de redondance sont cruciales pour déployer un SCN de grande échelle, capable de résister aux aléas d'une infrastructure distribuée, tout en maintenant l'intégrité et la continuité de son processus d'auto-organisation.

5.8.3. Contrôle d'Intégrité

Alors même que l'on déploie des mécanismes de sécurité (5.8.1) et assure la robustesse face aux pannes (5.8.2), il reste un volet essentiel : **vérifier l'intégrité** de la dynamique ω en continu, de manière à détecter à la fois des anomalies locales (liens "invraisemblables") et d'éventuelles incohérences globales (perte de cohérence statistique du réseau). Dans un SCN à grande échelle ou évolutif, un tel **contrôle d'intégrité** garantit que la formation des clusters ou la stabilisation des pondérations n'est pas subrepticement dévoyée par des artefacts (erreurs, manipulations, accidents). La section 5.8.3 détaille d'abord (5.8.3.1) la détection de liens aberrants, puis (5.8.3.2) évoque des indicateurs de cohérence globale.

5.8.3.1. Détection de Liens Aberrants (Trop Forts Trop Vite)

Dans le cadre d'un **Synergistic Connection Network** (SCN), la stabilité de la dynamique d'auto-organisation repose sur des mises à jour progressives et contrôlées des pondérations $\omega_{i,j}$. En effet, la règle de mise à jour classique

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)]$$

est conçue pour induire une évolution graduelle, dans laquelle les valeurs des liens se renforcent ou décroissent de façon cohérente avec les paramètres du système, notamment le taux η et le coefficient de décroissance τ . Cependant, il peut survenir des situations où un lien évolue de manière anormale, c'est-à-dire où $\omega_{i,j}$ « explose » ou augmente de façon disproportionnée d'une itération à l'autre. Ce phénomène, désigné par l'expression « trop forts trop vite », constitue un indice de comportement aberrant, qu'il soit provoqué par des erreurs de calcul, des dysfonctionnements logiciels ou des attaques malveillantes. La détection de ces liens anormaux est essentielle afin de préserver la cohérence du réseau et de déclencher, le cas échéant, des mesures correctives.

A. Motivation et Contexte

Le principe fondamental de l'auto-organisation dans un SCN repose sur la mise à jour itérative des pondérations. En temps normal, cette mise à jour induit une progression contrôlée, telle que décrite par l'équation

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)],$$

où la valeur de $S(i, j)$ – la synergie entre les entités i et j – détermine l'orientation du renforcement ou de l'affaiblissement du lien, et où le facteur η module l'amplitude des modifications. Dans un système bien calibré, on

s'attend à ce que les variations $\Delta\omega_{i,j} = |\omega_{i,j}(t+1) - \omega_{i,j}(t)|$ soient relativement modestes, se conformant à une distribution normale ou à un intervalle prédéfini de variations.

Pourtant, lorsqu'un lien subit une croissance brutale – par exemple, une augmentation de la valeur par un facteur de 100 en une seule itération – il devient impératif de qualifier ce comportement d'**aberrant**. Ce type d'évolution peut être le résultat d'une erreur logicielle (panne dans le module de calcul de la synergie ou bug dans l'implémentation de la règle de mise à jour), d'un dysfonctionnement numérique (problèmes d'overflow ou d'arrondis en environnement GPU/HPC), ou encore d'une manipulation malveillante (voir section 5.8.1 pour le contexte d'attaques).

B. Définition et Caractéristiques d'un Lien Aberrant

Dans un SCN mature, la majorité des pondérations $\omega_{i,j}$ se situent dans un intervalle relativement restreint, par exemple entre 0 et 1, voire un peu au-dessus en fonction de la normalisation appliquée. Pour quantifier ce qui constitue une valeur « hors norme », on peut définir des indicateurs statistiques tels que la moyenne μ et l'écart-type σ de la distribution des ω . Un lien est considéré comme aberrant s'il se trouve en dehors de l'intervalle

$$\mu \pm k \sigma,$$

où k est un paramètre choisi en fonction du degré de tolérance souhaité (par exemple, $k = 3$ pour une détection de valeurs extrêmes dans une distribution gaussienne). De même, la définition d'un seuil statique ω_{limit} – par exemple, une valeur de 2.0 ou 10.0 – permet d'identifier rapidement des liens dont la valeur excède largement la plage normale.

Au-delà de la valeur absolue d'un lien, l'analyse de la vitesse d'évolution, c'est-à-dire du saut

$$\Delta\omega_{i,j} = |\omega_{i,j}(t+1) - \omega_{i,j}(t)|,$$

est tout aussi cruciale. Dans un contexte de mise à jour graduelle, le paramètre η limite généralement l'amplitude des incréments. Ainsi, si dans des conditions standards on observe des variations d'ordre $O(10^{-2})$ ou $O(10^{-1})$, une variation de $O(1)$ ou supérieure en une seule itération suggère une anomalie. On peut introduire un test simple sous la forme

$$|\omega_{i,j}(t+1) - \omega_{i,j}(t)| > \delta,$$

où δ est un seuil défini en fonction de la dynamique attendue. Le franchissement de ce seuil constitue un signal d'alerte indiquant qu'un lien évolue de manière trop rapide.

C. Mécanismes de Détection

La détection des liens aberrants dans un SCN peut s'appuyer sur plusieurs mécanismes complémentaires qui, ensemble, forment une couche de sécurité supplémentaire dans la boucle de mise à jour des pondérations.

La méthode la plus directe consiste à imposer un seuil statique, noté ω_{limit} , au-delà duquel un lien est automatiquement marqué comme suspect. Parallèlement, on peut effectuer un contrôle sur la variation entre deux itérations. Concrètement, si l'on observe

$$\omega_{i,j}(t+1) > \omega_{\text{limit}} \quad \text{ou} \quad |\omega_{i,j}(t+1) - \omega_{i,j}(t)| > \delta,$$

alors le lien concerné est considéré comme aberrant. Ces seuils peuvent être définis de manière statique pour un SCN donné ou ajustés dynamiquement en fonction de la phase de convergence du réseau.

Une approche plus globale consiste à analyser la distribution des valeurs de ω sur l'ensemble du réseau. En estimant la moyenne μ et l'écart-type σ des pondérations, il est possible d'identifier les valeurs qui se trouvent en dehors d'un intervalle de confiance, par exemple $\mu \pm 3\sigma$. Cette approche permet de repérer non seulement des liens isolés, mais aussi des motifs d'anomalies lorsque plusieurs liens s'écartent simultanément des valeurs attendues. De plus, l'analyse des quantiles (par exemple, considérer que 99 % des liens se trouvent en dessous d'une valeur donnée) peut servir d'indicateur pour fixer dynamiquement un seuil.

Les techniques de **machine learning** appliquées à la détection d'anomalies offrent une méthode avancée pour identifier les comportements aberrants dans le réseau. Ces algorithmes, entraînés sur des données historiques ou simulées du

SCN, permettent de reconnaître des patterns anormaux, tels que des sauts brusques de plusieurs liens simultanément ou l'apparition soudaine d'un cluster entier présentant des valeurs extrêmes. L'intégration de ces méthodes peut se faire en parallèle de la mise à jour des pondérations et fournir des alertes en temps réel sur la base d'un score d'anomalie calculé pour chaque lien.

D. Réactions et Mesures Correctives

La détection d'un lien aberrant n'est que la première étape ; il est essentiel de mettre en place des mécanismes de réaction pour limiter l'impact de tels événements.

Lorsqu'un lien est détecté comme aberrant, un système de surveillance (souvent appelé « Watcher ») peut générer une alerte immédiate, notifiant un administrateur ou un module de contrôle central. Cette alerte peut inclure des informations détaillées telles que l'identifiant du lien, la valeur observée, le saut $\Delta\omega_{i,j}$, et l'instant de l'événement, facilitant ainsi une analyse ultérieure.

Dans des environnements critiques, la détection d'un comportement anormal peut déclencher la suspension temporaire du lien concerné. Le système peut alors appliquer un mécanisme de rollback, annulant la mise à jour problématique et rétablissant la valeur précédente de $\omega_{i,j}$ jusqu'à ce qu'un recalcul soit effectué en toute sécurité. Par exemple, on peut définir la mise à jour corrective suivante :

$$\omega_{i,j}(t+1) = \min\{\omega_{i,j}(t+1), \omega_{\text{limit}}\},$$

ce qui impose un clamp (ou clip) sur la valeur du lien afin d'empêcher toute croissance incontrôlée.

Enfin, en complément des réactions immédiates, le SCN peut être équipé d'un module de réévaluation qui, après détection d'un lien aberrant, effectue une nouvelle estimation de la synergie $S(i,j)$ ou applique une mise à jour plus prudente pour ce lien. Cette stratégie peut inclure la mise en œuvre d'un facteur de pondération adaptatif temporaire ou d'un filtre qui atténue la réponse de mise à jour dans la zone de détection de l'anomalie.

Conclusion

La détection de liens aberrants dans un SCN est une composante cruciale pour assurer la robustesse et la fiabilité de la dynamique d'auto-organisation. En se fondant sur l'observation des valeurs absolues et des variations de $\omega_{i,j}$, des techniques statistiques telles que l'analyse de la distribution (moyenne, écart-type, quantiles) permettent de qualifier un lien comme anormal lorsque sa valeur dépasse un seuil préétabli ou lorsqu'un saut brutal est constaté. Des mécanismes complémentaires, tels que l'implémentation de tests statiques et dynamiques (par exemple, la condition $|\Delta\omega_{i,j}| > \delta$), ainsi que le déploiement d'algorithmes d'anomaly detection basés sur le machine learning, offrent une approche globale pour repérer et signaler les anomalies. Sur le plan opérationnel, la mise en place d'un module de surveillance (Watcher) intégré dans la boucle de mise à jour permet de déclencher des alertes, de suspendre ou de rétablir les mises à jour suspectes, et d'appliquer des mesures correctives telles que le rollback ou le clamp. Ces dispositifs, en collaboration avec des systèmes de logging détaillé et des contrôles de cohérence, garantissent que la dynamique du SCN reste fidèle aux principes d'auto-organisation, même face à des variations imprévues ou malveillantes.

19. 5.8.3.2. Calcul d'Indicateurs de Cohérence Globale (ex. Distribution Statistique des ω)

Dans le contexte d'un **Synergistic Connection Network** (SCN), la dynamique d'auto-organisation repose sur l'évolution contrôlée des pondérations $\omega_{i,j}$ qui relient les entités. Alors que la détection locale de liens aberrants (voir Section 5.8.3.1) permet d'identifier des sauts brusques dans la mise à jour de $\omega_{i,j}$, il est tout aussi crucial d'assurer qu'à l'échelle globale la matrice ω conserve une structure cohérente et ne dérive pas vers un état artificiellement faussé. La surveillance globale repose sur le calcul et le suivi d'indicateurs statistiques qui quantifient la « bonne santé » de la distribution des pondérations, garantissant ainsi la pertinence des clusters formés par le SCN. Nous développerons ici en détail les motivations, les exemples d'indicateurs, les méthodes de surveillance et les considérations tant mathématiques que pratiques associées à cette démarche.

A. Motivation et Contexte de la Surveillance Globale

L'évolution normale d'un SCN repose sur une mise à jour graduelle de $\omega_{i,j}$ selon la règle

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)],$$

où les paramètres η et τ régulent respectivement le taux d'apprentissage et la décroissance. Dans un système bien calibré, la majorité des liens évolue dans une plage prédéfinie et stable, souvent centrée autour d'une valeur moyenne avec une dispersion modérée. Toutefois, plusieurs risques peuvent compromettre cette cohérence globale :

- **Dérive systémique** : Même en l'absence d'erreurs locales, un mauvais paramétrage (valeurs inappropriées de η ou τ) peut entraîner une dérive collective des valeurs de ω . Par exemple, une croissance exponentielle de la somme totale des pondérations,

$$\Sigma(t) = \sum_{i,j} \omega_{i,j}(t),$$

ou un effondrement généralisé vers zéro, pourraient indiquer un problème de convergence global.

- **Manipulations malveillantes** : Comme évoqué dans la Section 5.8.1, un attaquant peut altérer certaines valeurs, créant des biais qui, même s'ils ne se manifestent pas immédiatement sur un lien isolé, perturbent l'ensemble de la distribution.
- **Effets numériques** : Des erreurs d'arrondi, des overflow ou des problèmes de précision dans des environnements de calcul intensif (GPU, HPC) peuvent provoquer des modifications subtiles mais cumulatives dans la répartition des ω .

Ainsi, il est essentiel de disposer d'indicateurs globaux qui, en agrégeant les données sur l'ensemble du SCN, permettent de détecter des dérives ou des anomalies pouvant compromettre la formation et la stabilité des clusters.

B. Exemples d'Indicateurs de Cohérence Globale

Pour évaluer la « santé » globale de la matrice ω , plusieurs indicateurs peuvent être calculés et suivis dans le temps.

Une première approche consiste à analyser la distribution statistique des valeurs de ω en calculant des moments de la distribution. La **moyenne** $\bar{\omega}$ et la **variance** $\text{Var}(\omega)$ sont définies par :

$$\bar{\omega} = \frac{1}{N} \sum_{i,j} \omega_{i,j}, \quad \text{Var}(\omega) = \frac{1}{N} \sum_{i,j} (\omega_{i,j} - \bar{\omega})^2,$$

où N est le nombre total de liaisons. En outre, l'analyse des **quantiles** permet de connaître la répartition des valeurs : par exemple, on peut déterminer le 95^e percentile et vérifier que 95 % des liens se situent en dessous d'une certaine valeur. Des écarts brusques dans ces indicateurs, par exemple une augmentation soudaine de la moyenne ou un élargissement de la variance, peuvent signaler un déséquilibre dans la dynamique globale.

L'**entropie** $H(\omega)$ offre une mesure de la diversité des pondérations et se définit par :

$$H(\omega) = - \sum_b p(b) \log p(b),$$

où $p(b)$ est la probabilité qu'un lien $\omega_{i,j}$ se situe dans le bin b d'un histogramme construit sur les valeurs de ω . Une entropie élevée indique une distribution diversifiée, tandis qu'une entropie faible pourrait suggérer une homogénéisation anormale (par exemple, si presque tous les ω convergent vers une valeur unique). Un changement soudain de l'entropie au fil du temps peut donc être un indicateur puissant d'une dérive globale.

Le suivi de la **somme globale** des pondérations,

$$\Sigma(t) = \sum_{i,j} \omega_{i,j}(t),$$

permet de détecter des tendances globales d'emballlement ou d'inhibition. Une croissance exponentielle de $\Sigma(t)$ ou, inversement, une diminution drastique vers zéro, signale une perturbation de la dynamique du réseau. Par ailleurs, le ratio entre la valeur maximale et la valeur minimale,

$$R(t) = \frac{\max_{i,j} \omega_{i,j}(t)}{\min_{i,j} \omega_{i,j}(t)},$$

peut servir d'indicateur de dispersion extrême. Si $R(t)$ augmente de manière anormale, cela suggère que certaines liaisons se renforcent de manière disproportionnée par rapport aux autres, ce qui peut être symptomatique d'une dérive globale.

Les propriétés topologiques du SCN, telles que la **modularité** et la **connectivité** des clusters, sont également des indicateurs pertinents. La modularité Q d'une partition du graphe, qui quantifie la densité des liens à l'intérieur des clusters par rapport aux liens entre les clusters, peut être calculée pour évaluer la qualité des communautés formées. Une fluctuation soudaine de Q pourrait indiquer que la structure interne du SCN se modifie de manière inattendue, potentiellement en raison d'un emballlement des ω .

C. Méthodes de Surveillance et Seuils d'Alarme

Afin de garantir que le SCN reste dans un état de cohérence global acceptable, il est nécessaire de mettre en place des systèmes de surveillance qui calculent périodiquement ces indicateurs. Deux approches complémentaires peuvent être déployées :

- **Surveillance en continu** : À chaque itération (ou à intervalles réguliers), on collecte un échantillon représentatif de la matrice ω et on calcule les indicateurs (moyenne, variance, entropie, somme globale). Ces valeurs sont comparées à des plages acceptables, définies à partir d'un historique ou d'un modèle théorique. Un dépassement des seuils prédéfinis déclenche une alerte.
- **Analyses « offline »** : Des snapshots de la matrice ω sont stockés périodiquement pour permettre une analyse plus approfondie des tendances sur le long terme. Ces analyses rétrospectives offrent la possibilité de détecter des dérives lentes mais significatives et de reconfigurer le système en cas de besoin.

D. Considérations Mathématiques et Pratiques

D'un point de vue mathématique, l'étude de la distribution des pondérations peut être assimilée à une analyse de la stabilité d'un système dynamique. Le SCN, qui évolue selon

$$\omega(t+1) = F(\omega(t)),$$

doit converger vers un état stable ou présenter une distribution stationnaire. L'introduction de contrôles statistiques (moyenne, variance, entropie) constitue une forme de rétroaction qui permet de comparer l'état actuel à l'état attendu, en appliquant des tests d'hypothèse pour détecter des écarts significatifs. Par exemple, si l'on définit une fenêtre de confiance $\mu \pm k\sigma$ (avec k fixé par la théorie ou l'expérience), toute valeur dépassant cette fenêtre peut être considérée comme une anomalie.

Sur le plan pratique, ces calculs doivent être implémentés de manière efficace, notamment dans des environnements distribués où le nombre de liaisons peut être très élevé. L'utilisation de techniques d'échantillonnage et l'agrégation des statistiques sur des bases de données NoSQL ou des systèmes de traitement de graphes permettent de réduire le coût computationnel tout en assurant une surveillance robuste.

Conclusion

Le calcul d'indicateurs de cohérence globale est une étape essentielle pour assurer que la matrice ω d'un SCN conserve une distribution cohérente et conforme aux attentes théoriques, malgré les perturbations ou manipulations potentielles. En surveillant des indicateurs tels que la moyenne, la variance, l'entropie, la somme globale et la modularité, il est possible de détecter des dérives systémiques qui pourraient compromettre la formation des clusters et l'auto-organisation du réseau. Ces indicateurs, calculés en continu

ou de manière périodique, permettent de déclencher des alarmes et d'initier des actions correctives pour maintenir la qualité et la stabilité du SCN. Sur le plan mathématique, ces outils fournissent un regard global sur la distribution des pondérations et servent de base pour ajuster dynamiquement les paramètres du système (par exemple, η et τ) ou activer des mécanismes de contrôle supplémentaire pour prévenir toute dérive systématique. Ainsi, le suivi des indicateurs de cohérence globale est un levier crucial pour garantir la résilience et la fiabilité d'un SCN en fonctionnement.

5.9. Exemples d'Implémentations et Études de Cas

5.9.1. SCN Multimodal

- 5.9.1.1. Schéma modulaire : *ModuleSynergySub*, *ModuleSynergySym*, *SCNCore*, etc.
- 5.9.1.2. Expliquer comment se fait la boucle de mise à jour concrètement.

5.9.2. SCN Hybride (Symbolique + Sub-symbolique)

- 5.9.2.1. Entités logiques, vecteurs, gestion via la synergie multiple.
- 5.9.2.2. Rapide cas d'usage, mini-dataset.

5.9.3. Robotique ou Multi-Agent

- 5.9.3.1. Rappeler la distribution possible, la synchro inter-robots.
- 5.9.3.2. Visualisation d'un SCN en essaim robotique.

5.9. Exemples d'Implémentations et Études de Cas

Dans cette dernière partie du chapitre 5, nous proposons d'illustrer **concrètement** comment un **SCN** (Synergistic Connection Network) peut être **implémenté** ou employé dans des situations variées, montrant ainsi la **mise en œuvre** pratique des concepts théoriques et d'architecture présentés jusque-là. Les études de cas (5.9.1, 5.9.2, 5.9.3) visent à donner des **exemples** concrets — chacun abordant un type différent de données ou d'environnement — et à détailler la **structure logicielle** (modules, boucles de mise à jour, etc.) nécessaire pour un fonctionnement effectif. Même si l'on s'est concentré en amont sur la dimension mathématique et la distribution (chap. 5.7), ces exemples montrent comment passer du cadre théorique à l'expérimentation ou à l'application dans des domaines multimodaux, hybrides (symbolique/sub-symbolique) ou robotiques.

5.9.1. SCN Multimodal

Lorsqu'un SCN doit manipuler plusieurs **types** de données (images, sons, textes, etc.) au sein d'un même réseau, il doit non seulement organiser la distribution ou la gestion de la matrice ω , mais aussi prévoir des **modules** de synergie adaptés à chaque modalité. La sous-section (5.9.1.1) décrit un **schéma modulaire** commun, puis (5.9.1.2) s'attache à la **boucle de mise à jour** concrète.

Dans cette section, nous développons en détail le schéma modulaire appliqué à un **Synergistic Connection Network** (SCN) multimodal, en mettant en évidence les différents modules qui interviennent dans le calcul de la synergie entre entités hétérogènes, ainsi que leur intégration dans le cœur du SCN. Nous présenterons d'abord le cadre théorique et mathématique de cette approche, puis nous proposerons une implémentation en Python de l'ensemble des modules évoqués, à savoir : *ModuleSynergySub*, *ModuleSynergySym* et *SCNCore*. Cette démarche permet d'illustrer comment, en séparant la logique de calcul de similarité selon les modalités (sub-symbolique vs. symbolique), on obtient une architecture évolutive, lisible et extensible.

I. Schéma Modulaire : Cadre Théorique et Mathématique

A. Contexte Multimodal et Besoin de Modulation

Dans de nombreux systèmes d'auto-organisation, notamment dans le cadre du **Deep Synergy Learning (DSL)**, les entités $\{\mathcal{E}_i\}$ ne sont pas homogènes. Certaines représentent des caractéristiques d'images (extraits d'un CNN), d'autres des descripteurs audio (par exemple, MFCC, spectrogrammes), et d'autres encore des tokens ou des concepts issus du traitement du langage naturel. Chaque entité est ainsi associée à un espace particulier \mathcal{X}_i , ce qui impose de définir la fonction de synergie $S(i, j)$ de manière adaptée à la modalité considérée.

Pour gérer cette hétérogénéité, il est naturel de séparer le calcul de la synergie en plusieurs modules spécialisés :

- **ModuleSynergySub** s'occupe des représentations sub-symboliques (images, audio, signaux sensoriels) et implémente des méthodes de calcul de similarité adaptées à des vecteurs continus (par exemple, similarité cosinus pour les images, distance inversée pour l'audio).
- **ModuleSynergySym** prend en charge le calcul de la synergie pour des entités symboliques, par exemple en exploitant des méthodes issues de l'analyse sémantique ou de la co-occurrence dans des espaces discrets.
- Le **SCNCore** constitue le noyau du système. Il maintient la matrice de pondérations ω qui encode les liaisons entre entités et applique la règle de mise à jour typique, par exemple dans sa forme additive :

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)].$$

Pour calculer $S(i, j)$, le *SCNCore* délègue l'opération au module spécialisé approprié, selon la nature des entités i et j .

Cette factorisation, qui revient mathématiquement à écrire

$$S(i, j) = f_{m(i, j)}(\mathcal{E}_i, \mathcal{E}_j),$$

où $m(i, j)$ désigne la modalité (ou la combinaison de modalités) concernée, permet non seulement d'alléger la complexité logicielle (en évitant un "god object" regroupant toutes les conditions) mais aussi d'assurer une extensibilité du système, chaque module pouvant être mis à jour ou remplacé indépendamment.

B. Bénéfices du Schéma Modulaire

- **Séparation des responsabilités** : chaque module est responsable d'un type spécifique de calcul de similarité. Cela permet de garantir la clarté du code, de faciliter le débogage et la maintenance, ainsi que l'ajout de nouvelles modalités.
- **Extensibilité** : l'architecture modulaire autorise l'intégration de nouvelles sources de données sans modifier le cœur de la dynamique $\omega(t)$. Par exemple, l'ajout d'une nouvelle modalité (données Lidar, signaux physiologiques, etc.) ne nécessite que l'implémentation d'un nouveau module de calcul de synergie.
- **Réutilisabilité et Déploiement Distribué** : chaque module peut être déployé sous forme de microservice indépendant, permettant une architecture distribuée où chaque sous-SCN conserve une structure modulaire similaire.

II. Implémentation en Python

Nous proposons ci-dessous une implémentation en Python de l'architecture modulaire décrite. Le code comporte trois classes principales :

- **ModuleSynergy** (classe de base abstraite)
- **ModuleSynergySub** et **ModuleSynergySym**, qui dérivent de la classe de base et implémentent chacune une méthode de calcul de similarité adaptée aux représentations sub-symboliques et symboliques, respectivement.
- **SCNCore** qui gère la matrice de pondérations ω et orchestre les mises à jour en appelant le module de synergie approprié selon la modalité des entités.

Pour illustrer cette implémentation, nous définirons également une classe **Entity** permettant de représenter les entités multimodales.

Voici le code complet :

```
import numpy as np
from abc import ABC, abstractmethod
from typing import Any, Dict, List, Tuple

# Définition d'une classe Entity pour encapsuler les entités multimodales
class Entity:
    """
    Représente une entité dans le SCN.

    Attributes:
    id (str): Identifiant unique de l'entité.
    modality (str): Type de modalité ('sub' pour sub-symbolique, 'sym' pour symbolique).
    data (Any): Données associées à l'entité.
```



```

"""
def __init__(self, id: str, modality: str, data: Any):
    self.id = id
    self.modality = modality
    self.data = data

# Classe de base abstraite pour les modules de calcul de synergie
class ModuleSynergy(ABC):
    """
    Classe abstraite définissant l'interface pour le calcul de la synergie entre deux entités.
    """

    @abstractmethod
    def compute_similarity(self, entity1: Entity, entity2: Entity) -> float:
        """
        Calcule et renvoie un score de similarité entre entity1 et entity2.
        """

        pass

# ModuleSynergySub : pour les représentations sub-symboliques (images, audio, etc.)
class ModuleSynergySub(ModuleSynergy):
    """
    Module pour le calcul de la synergie entre entités sub-symboliques.
    On suppose que les données sont des vecteurs numpy.
    """

    def compute_similarity(self, entity1: Entity, entity2: Entity) -> float:
        # Vérifier que les deux entités appartiennent à la modalité sub-symbolique
        if entity1.modality != 'sub' or entity2.modality != 'sub':
            raise ValueError("ModuleSynergySub ne peut traiter que des entités sub-symboliques.")

        # Par exemple, on peut utiliser la similarité cosinus
        vec1 = np.array(entity1.data)
        vec2 = np.array(entity2.data)
        # Calcul de la similarité cosinus
        norm1 = np.linalg.norm(vec1)
        norm2 = np.linalg.norm(vec2)
        if norm1 == 0 or norm2 == 0:
            return 0.0
        similarity = np.dot(vec1, vec2) / (norm1 * norm2)
        # On peut normaliser pour qu'elle soit dans [0, 1] (si besoin)
        return max(0.0, similarity)

# ModuleSynergySym : pour les entités symboliques (textes, concepts, etc.)
class ModuleSynergySym(ModuleSynergy):
    """
    Module pour le calcul de la synergie entre entités symboliques.
    Ici, on utilise une méthode simplifiée basée sur la correspondance exacte ou une mesure de similarité simple.
    """

    def compute_similarity(self, entity1: Entity, entity2: Entity) -> float:
        if entity1.modality != 'sym' or entity2.modality != 'sym':
            raise ValueError("ModuleSynergySym ne peut traiter que des entités symboliques.")

        # Par exemple, si les données sont des chaînes de caractères représentant des concepts,
        # on renvoie 1.0 si elles sont identiques, 0.0 sinon, ou on pourrait utiliser une mesure plus fine.
        if entity1.data == entity2.data:
            return 1.0
        else:

```

```

    return 0.0

# SCNCORE : le cœur du SCN qui maintient la matrice des pondérations et orchestre la mise à jour.
class SCNCORE:
    """
    SCNCORE gère la dynamique du réseau en conservant la matrice des pondérations  $\omega$ 
    et en appliquant la règle de mise à jour sur la base des scores de synergie.
    """

    def __init__(self, entities: List[Entity], eta: float = 0.1, tau: float = 0.2):
        """
        Initialise le SCNCORE avec la liste d'entités, le taux d'apprentissage (eta) et le coefficient de décroissance (tau).
        La matrice  $\omega$  est initialisée aléatoirement pour toutes les paires d'entités.
        """

        self.entities = entities
        self.N = len(entities)
        self.eta = eta
        self.tau = tau
        # Initialisation de la matrice  $\omega$  avec de petites valeurs aléatoires
        self.omega = np.random.uniform(0.01, 0.05, size=(self.N, self.N))
        # Pour garantir la symétrie, on peut symétriser la matrice
        self.omega = (self.omega + self.omega.T) / 2.0

        # Instanciation des modules de synergie
        self.module_synergy_sub = ModuleSynergySub()
        self.module_synergy_sym = ModuleSynergySym()

    def compute_synergy(self, i: int, j: int) -> float:
        """
        Calcule le score de synergie entre les entités d'indices i et j
        en fonction de leur modalité.
        """

        entity1 = self.entities[i]
        entity2 = self.entities[j]
        # Si les deux entités ont la modalité sub-symbolique, utiliser ModuleSynergySub
        if entity1.modality == 'sub' and entity2.modality == 'sub':
            return self.module_synergy_sub.compute_similarity(entity1, entity2)
        # Si les deux entités ont la modalité symbolique, utiliser ModuleSynergySym
        elif entity1.modality == 'sym' and entity2.modality == 'sym':
            return self.module_synergy_sym.compute_similarity(entity1, entity2)
        # Pour des paires de modalités différentes, on définit une stratégie de fusion (ici, on prend la moyenne)
        else:
            score_sub = 0.0
            score_sym = 0.0
            count = 0
            if entity1.modality == 'sub' or entity2.modality == 'sub':
                try:
                    score_sub = self.module_synergy_sub.compute_similarity(entity1, entity2)
                    count += 1
                except ValueError:
                    pass
            if entity1.modality == 'sym' or entity2.modality == 'sym':
                try:
                    score_sym = self.module_synergy_sym.compute_similarity(entity1, entity2)
                    count += 1
                except ValueError:
                    pass

```

```

        # Si aucun module n'est applicable, renvoyer 0
        if count == 0:
            return 0.0
        return (score_sub + score_sym) / count

def update_weights(self):
    """
    Met à jour la matrice des pondérations  $\omega$  selon la règle :
     $\omega(t+1) = \omega(t) + \eta [S(i,j) - \tau \omega(t)]$ 
    Pour chaque paire (i,j) avec  $i < j$ , la mise à jour est effectuée et
    la symétrie est rétablie.
    """
    new_omega = self.omega.copy()
    for i in range(self.N):
        for j in range(i+1, self.N):
            S_ij = self.compute_synergy(i, j)
            # Calcul de la mise à jour pour la paire (i,j)
            delta = self.eta * (S_ij - self.tau * self.omega[i, j])
            new_val = self.omega[i, j] + delta
            new_omega[i, j] = new_val
            new_omega[j, i] = new_val # Symétrie
    self.omega = new_omega

def run_iterations(self, iterations: int = 10):
    """
    Exécute un nombre d'itérations de mise à jour de la matrice  $\omega$  et affiche les valeurs moyennes.
    """
    for t in range(iterations):
        self.update_weights()
        mean_val = np.mean(self.omega)
        print(f"Itération {t+1}: moyenne des  $\omega$  = {mean_val:.4f}")

# Exemple d'utilisation du schéma modulaire dans un SCN multimodal
def main():
    # Création d'une liste d'entités avec différentes modalités
    entities = [
        # Entités sub-symboliques (images ou audio) : on simule avec des vecteurs numpy
        Entity("E1", "sub", [0.2, 0.4, 0.6]),
        Entity("E2", "sub", [0.1, 0.3, 0.5]),
        Entity("E3", "sub", [0.25, 0.35, 0.45]),
        # Entités symboliques (textuelles ou conceptuelles)
        Entity("E4", "sym", "chat"),
        Entity("E5", "sym", "chat"),
        Entity("E6", "sym", "chien"),
        # Entités mixtes : ici nous les traiterons par fusion (le calcul renverra la moyenne des scores disponibles)
        Entity("E7", "sub", [0.5, 0.2, 0.1]),
        Entity("E8", "sym", "oiseau")
    ]

    # Initialisation du cœur du SCN avec les entités
    scn_core = SCNCORE(entities, eta=0.1, tau=0.2)

    # Affichage initial de la matrice des pondérations
    print("Matrice  $\omega$  initiale:")
    print(np.round(scn_core.omega, 4))

```

```

# Exécuter plusieurs itérations de mise à jour
scn_core.run_iterations(iterations=10)

# Affichage final de la matrice des pondérations
print("Matrice  $\omega$  finale:")
print(np.round(scn_core.omega, 4))

if __name__ == "__main__":
    main()

```

Explications Complémentaires

1. ModuleSynergySub

Ce module est dédié au calcul de la synergie entre des entités dont les représentations sont sub-symboliques, c'est-à-dire des vecteurs continus (par exemple, les sorties d'un CNN pour des images ou des descripteurs audio). Dans l'implémentation proposée, nous utilisons la **similarité cosinus** :

$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

ce qui permet d'obtenir un score de similarité compris entre 0 et 1. Ce module lève une exception si des entités d'un type incompatible lui sont passées.

2. ModuleSynergySym

Ce module gère les entités de nature symbolique. Pour simplifier, nous utilisons ici un test d'égalité entre les données associées aux entités. Dans un contexte réel, on pourrait envisager des mesures plus fines (par exemple, des distances sémantiques basées sur des graphes ontologiques ou des embeddings pré-entraînés).

3. SCNCORE

Le cœur du SCN, *SCNCORE*, orchestre la dynamique globale en maintenant la matrice des pondérations ω et en appliquant la règle de mise à jour. Pour chaque paire d'entités, il interroge le module de synergie approprié (en fonction de la modalité) pour obtenir $S(i, j)$ et met à jour $\omega_{i,j}$ selon le schéma DSL. La symétrie de la matrice est assurée lors de la mise à jour.

4. Cohabitation et Fusion

Dans le cas où deux entités présentent des modalités différentes (par exemple, une entité sub-symbolique et une entité symbolique), nous avons implémenté une stratégie simple qui tente de combiner les scores obtenus par chacun des modules. Si l'un des modules ne peut pas traiter la paire, le score de l'autre est utilisé seul. Ce mécanisme de fusion simple peut être étendu ou modifié selon les besoins spécifiques d'intégration multimodale.

Conclusion

La conception modulaire du SCN, telle que présentée dans la Section 5.9.1.1, repose sur la séparation des tâches de calcul de la synergie selon la nature des entités. En divisant le problème en sous-modules spécialisés (*ModuleSynergySub* pour les données sub-symboliques et *ModuleSynergySym* pour les données symboliques) et en centralisant la dynamique de mise à jour dans le *SCNCORE*, nous obtenons une architecture claire, extensible et facilement maintenable. L'implémentation en Python ci-dessus traduit cette approche de manière concrète, offrant un cadre pédagogique complet pour la compréhension et l'expérimentation de SCN multimodaux dans un contexte de Deep Synergy Learning.

5.9.1.2. Expliquer comment se fait la boucle de mise à jour concrètement

Dans un SCN multimodal, la boucle de mise à jour est structurée de manière à intégrer deux aspects complémentaires :

- D'une part, le calcul du score de **synergie** $S(i, j)$ entre chaque paire d'entités, qui dépend de la nature (modalité) des données de chaque entité ;
- D'autre part, la mise à jour de la matrice des pondérations ω selon une règle d'auto-organisation (par exemple, une règle additive) qui se base sur le score $S(i, j)$.

Nous allons décrire concrètement les étapes d'une itération complète de mise à jour du SCN.

A. Les Étapes de la Boucle de Mise à Jour

Pour chaque paire d'entités (i, j) , le SCN doit déterminer la valeur de synergie qui mesure la « compatibilité » ou l'**affinité** entre les deux entités. Dans un SCN multimodal, cette opération se fait de la manière suivante :

- **Identification de la modalité** : Chaque entité \mathcal{E}_i porte une étiquette (par exemple, « sub » pour sub-symbolique, « sym » pour symbolique). Le **SCNCore** détermine ainsi si la paire (i, j) doit être traitée par le module *ModuleSynergySub* (données continues, vecteurs issus d'images ou d'audio) ou par *ModuleSynergySym* (données symboliques ou conceptuelles).
- **Délégation du calcul** : Le **SCNCore** appelle le module de synergie approprié pour obtenir un score $S(i, j)$. Ce score est généralement normalisé (par exemple, dans l'intervalle $[0, 1]$).

Une fois le score $S(i, j)$ connu, la mise à jour des pondérations se fait via une règle d'auto-organisation. Dans la forme additive classique, la mise à jour s'exprime par :

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)]$$

où :

- η est le **taux d'apprentissage**,
- τ est le **coefficient de décroissance** qui assure une régulation en empêchant $\omega_{i,j}$ de croître indéfiniment.

Ce calcul est effectué pour chaque paire (i, j) concernée. Dans une implémentation pratique, on ne met à jour que les liaisons actives (par exemple, celles qui sont au-dessus d'un certain seuil ou dans un voisinage défini) afin de limiter le coût computationnel lorsque le nombre d'entités est très grand.

Après la mise à jour des valeurs :

- **Logs et Traçabilité** : Chaque modification de $\omega_{i,j}$ est enregistrée dans un journal (log) pour permettre le suivi et la détection d'anomalies ultérieures.
- **Watchers** : Des modules de surveillance vérifient que l'incrément $\Delta\omega_{i,j}$ n'est pas trop important (par exemple, qu'il ne dépasse pas un seuil δ).
- **Synchronisation (pour les architectures distribuées)** : Dans un SCN réparti, la mise à jour locale est synchronisée avec d'autres blocs via des messages inter-blocs afin de garantir la cohérence globale du réseau.

Une fois que toutes les liaisons ont été mises à jour et que les contrôles ont été effectués, la nouvelle matrice $\omega(t + 1)$ sert d'état de départ pour la prochaine itération. La boucle se répète ainsi, permettant au SCN de converger vers une organisation stable et de former des clusters représentatifs des synergies entre entités.

B. Implémentation en Python

Nous présentons ici un exemple d'implémentation qui intègre les modules suivants :

- **ModuleSynergySub** : pour le calcul de la synergie entre entités sub-symboliques (par exemple, en utilisant la similarité cosinus).
- **ModuleSynergySym** : pour le calcul de la synergie entre entités symboliques (ici, une simple comparaison d'égalité).
- **SCNCore** : le cœur du SCN qui maintient la matrice ω et orchestre la boucle de mise à jour.

Vous trouverez ci-dessous le code complet, avec des commentaires détaillés pour expliquer chaque étape.

```
import numpy as np
from abc import ABC, abstractmethod
from typing import Any, List

# Définition d'une classe Entity pour représenter une entité multimodale
class Entity:
    """
    Représente une entité dans le SCN.

    Attributs:
    id (str): Identifiant unique de l'entité.
    modality (str): 'sub' pour sub-symbolique, 'sym' pour symbolique.
    data (Any): Données associées (par exemple, un vecteur pour sub ou une chaîne pour sym).
    """
    def __init__(self, id: str, modality: str, data: Any):
        self.id = id
        self.modality = modality
        self.data = data

# Classe de base abstraite pour le calcul de synergie
class ModuleSynergy(ABC):
    @abstractmethod
    def compute_similarity(self, entity1: Entity, entity2: Entity) -> float:
        """
        Calcule et renvoie un score de similarité entre entity1 et entity2.
        """
        pass

# Module pour les données sub-symboliques
class ModuleSynergySub(ModuleSynergy):
    def compute_similarity(self, entity1: Entity, entity2: Entity) -> float:
        # Vérification des modalités
        if entity1.modality != 'sub' or entity2.modality != 'sub':
            raise ValueError("ModuleSynergySub ne peut traiter que des entités sub-symboliques.")
        # On suppose que les données sont des vecteurs numpy
        vec1 = np.array(entity1.data)
        vec2 = np.array(entity2.data)
        norm1 = np.linalg.norm(vec1)
        norm2 = np.linalg.norm(vec2)
        if norm1 == 0 or norm2 == 0:
            return 0.0
        similarity = np.dot(vec1, vec2) / (norm1 * norm2)
        # On normalise pour obtenir un score dans [0, 1]
```

```

    return max(0.0, similarity)

# Module pour les données symboliques
class ModuleSynergySym(ModuleSynergy):
    def compute_similarity(self, entity1: Entity, entity2: Entity) -> float:
        if entity1.modality != 'sym' or entity2.modality != 'sym':
            raise ValueError("ModuleSynergySym ne peut traiter que des entités symboliques.")
        # Méthode simple : renvoie 1 si les données sont identiques, 0 sinon
        return 1.0 if entity1.data == entity2.data else 0.0

# SCNCORE : le cœur du SCN
class SCNCORE:
    def __init__(self, entities: List[Entity], eta: float = 0.1, tau: float = 0.2):
        """
        Initialise le SCN avec une liste d'entités, et définit les paramètres  $\eta$  et  $\tau$ .
        La matrice  $\omega$  est initialisée avec de petites valeurs aléatoires.
        """
        self.entities = entities
        self.N = len(entities)
        self.eta = eta
        self.tau = tau
        # Initialisation de la matrice  $\omega$  (symétrique) de dimension  $N \times N$ 
        self.omega = np.random.uniform(0.01, 0.05, size=(self.N, self.N))
        self.omega = (self.omega + self.omega.T) / 2.0

        # Instanciation des modules de synergie
        self.module_synergy_sub = ModuleSynergySub()
        self.module_synergy_sym = ModuleSynergySym()

    def compute_synergy(self, i: int, j: int) -> float:
        """
        Calcule le score de synergie  $S(i,j)$  en fonction des modalités des entités.
        """
        e1 = self.entities[i]
        e2 = self.entities[j]
        if e1.modality == 'sub' and e2.modality == 'sub':
            return self.module_synergy_sub.compute_similarity(e1, e2)
        elif e1.modality == 'sym' and e2.modality == 'sym':
            return self.module_synergy_sym.compute_similarity(e1, e2)
        else:
            # Pour les cas mixtes, on combine les scores (ici par moyenne simple)
            scores = []
            try:
                scores.append(self.module_synergy_sub.compute_similarity(e1, e2))
            except ValueError:
                pass
            try:
                scores.append(self.module_synergy_sym.compute_similarity(e1, e2))
            except ValueError:
                pass
            return np.mean(scores) if scores else 0.0

    def update_weights(self):
        """
        Met à jour la matrice  $\omega$  pour chaque paire  $(i, j)$  en appliquant la règle :
         $\omega(t+1) = \omega(t) + \eta [ S(i,j) - \tau \omega(t) ]$ 
        """

```

```

"""
new_omega = self.omega.copy()
for i in range(self.N):
    for j in range(i + 1, self.N):
        S_ij = self.compute_synergy(i, j)
        delta = self.eta * (S_ij - self.tau * self.omega[i, j])
        new_val = self.omega[i, j] + delta
        # Application éventuelle d'un mécanisme de saturation (ici simple clamping)
        new_val = min(new_val, 1.0) # par exemple, on impose  $\omega_{max} = 1.0$ 
        new_omega[i, j] = new_val
        new_omega[j, i] = new_val # symétrie
self.omega = new_omega

def run_update_cycle(self, iterations: int = 10):
    """
    Exécute la boucle de mise à jour pour un nombre donné d'itérations.
    À chaque itération, on met à jour  $\omega$  et on peut enregistrer l'état ou effectuer des contrôles.
    """
    for t in range(iterations):
        self.update_weights()
        mean_omega = np.mean(self.omega)
        print(f"Itération {t+1} : moyenne de  $\omega = \{mean\_omega:.4f\}")
        # Ici, on pourrait ajouter des appels à des modules de logs ou watchers

# Exemple d'utilisation
def main():
    # Création d'un ensemble d'entités multimodales
    entities = [
        # Entités sub-symboliques : représentées par des vecteurs (ex. d'images ou d'audio)
        Entity("E1", "sub", [0.2, 0.4, 0.6]),
        Entity("E2", "sub", [0.1, 0.3, 0.5]),
        Entity("E3", "sub", [0.25, 0.35, 0.45]),
        # Entités symboliques : représentées par des chaînes de caractères
        Entity("E4", "sym", "chat"),
        Entity("E5", "sym", "chat"),
        Entity("E6", "sym", "chien"),
        # Entités mixtes, pour démonstration, nous utilisons un calcul de fusion simple
        Entity("E7", "sub", [0.5, 0.2, 0.1]),
        Entity("E8", "sym", "oiseau")
    ]

    # Initialisation du SCNCORE avec les entités
    scn_core = SCNCORE(entities, eta=0.1, tau=0.2)
    print("Matrice  $\omega$  initiale :")
    print(np.round(scn_core.omega, 4))

    # Exécution de la boucle de mise à jour sur 10 itérations
    scn_core.run_update_cycle(iterations=10)

    print("Matrice  $\omega$  finale :")
    print(np.round(scn_core.omega, 4))

if __name__ == "__main__":
    main()$ 
```


A. Cycle d'Itération

Chaque itération dans le SCN se décompose en plusieurs étapes :

20. Calcul de $S(i, j)$

Pour chaque paire (i, j) (typiquement, on parcourt uniquement $i < j$ pour maintenir la symétrie), le **SCNCore** interroge le module de synergie adéquat (sub-symbolique ou symbolique) via la méthode `compute_synergy`. Cette fonction retourne un score qui reflète l'affinité entre les entités i et j .

21. Application de la règle de mise à jour

À partir de la valeur $S(i, j)$ obtenue, la mise à jour se fait en ajoutant un terme $\eta [S(i, j) - \tau \omega_{i,j}]$ à la valeur actuelle de $\omega_{i,j}$. Dans notre exemple, nous appliquons aussi une saturation (clamping à 1.0) afin d'éviter des valeurs trop élevées.

22. Contrôles et Enregistrement

Après chaque mise à jour, des opérations de contrôle peuvent être exécutées (par exemple, via des watchers ou des logs) pour assurer la stabilité et la traçabilité de la dynamique. Dans notre implémentation, nous affichons la moyenne de la matrice ω pour suivre l'évolution globale.

23. Itération Suivante

La nouvelle matrice ω devient l'état de départ pour l'itération suivante. Cette boucle permet de simuler l'auto-organisation du réseau, qui converge progressivement vers une structure stable.

B. Gestion de la Multimodalité

Le **dispatching** de la fonction de calcul de synergie se fait dans la méthode `compute_synergy` de la classe `SCNCore`. Selon que les entités soient de type « sub » ou « sym », la méthode délègue le calcul au module approprié. Pour les cas mixtes, une stratégie simple de fusion (ici, la moyenne) est appliquée, mais cette logique peut être enrichie selon les exigences de votre système.

C. Extension et Intégration

Dans un système réel, d'autres modules pourraient être intégrés à ce pipeline :

- Des modules d'**inhibition** ou de **saturation** plus élaborés, qui ajusteraient les mises à jour en fonction de la dynamique globale.
- Des mécanismes de **logging** et de **surveillance** (voir les sections 5.8) qui enregistreraient et contrôlraient en temps réel la validité des mises à jour.
- Des protocoles de **synchronisation** pour des SCN distribués, permettant de garantir que la mise à jour se fasse de manière cohérente entre plusieurs nœuds.

Conclusion

La boucle de mise à jour dans un SCN multimodal se réalise en plusieurs étapes : d'abord, le calcul de la synergie $S(i, j)$ via des modules spécialisés qui traitent les modalités propres à chaque type d'entité, puis l'application de la règle de mise à jour sur la matrice des pondérations ω dans le **SCNCore**, suivi de contrôles complémentaires et enfin le passage à l'itération suivante. L'implémentation Python proposée illustre concrètement ce processus en séparant la logique de calcul de synergie (via `ModuleSynergySub` et `ModuleSynergySym`) de la gestion globale de la dynamique

d'auto-organisation (dans *SCNCore*). Ce schéma modulaire permet ainsi une architecture claire, extensible et apte à gérer efficacement la multimodalité, tout en assurant la robustesse et la cohérence du système.

Ci-dessous, vous trouverez un code complet qui reprend la structure modulaire déjà développée (avec les classes *Entity*, *ModuleSynergySub*, *ModuleSynergySym* et *SCNCore*), auquel nous ajoutons une classe **SynergyDispatcher** et une fonction **scn_distributed_iteration**. Ces éléments illustrent la manière dont le calcul de la synergie peut être dispatché selon le type des entités et comment, dans un scénario distribué, la mise à jour des pondérations se fait en tenant compte de l'appartenance locale ou non d'une entité.

```
import numpy as np
from abc import ABC, abstractmethod
from typing import Any, Dict, List, Tuple

#####
# 1. Classes de Base pour les Entités et Modules de Synergie
#####

class Entity:
    """
    Représente une entité dans le SCN.

    Attributs:
    id (str): Identifiant unique de l'entité.
    modality (str): Type de modalité, par exemple 'subsymbolic' ou 'symbolic'.
    data (Any): Données associées à l'entité (vecteur pour subsymbolic, chaîne pour symbolic).
    """
    def __init__(self, id: str, modality: str, data: Any):
        self.id = id
        self.modality = modality # Exemple : 'subsymbolic' ou 'symbolic'
        self.data = data

# Classe abstraite pour le calcul de synergie
class ModuleSynergy(ABC):
    @abstractmethod
    def compute_similarity(self, entity1: Entity, entity2: Entity) -> float:
        """
        Calcule et renvoie un score de similarité entre entity1 et entity2.
        """
        pass

# Module pour les données subsymboliques (ex. images, audio)
class ModuleSynergySub(ModuleSynergy):
    def compute_similarity(self, entity1: Entity, entity2: Entity) -> float:
        if entity1.modality != 'subsymbolic' or entity2.modality != 'subsymbolic':
            raise ValueError("ModuleSynergySub ne peut traiter que des entités subsymboliques.")
        # Exemple : similarité cosinus sur des vecteurs
        vec1 = np.array(entity1.data)
        vec2 = np.array(entity2.data)
        norm1 = np.linalg.norm(vec1)
        norm2 = np.linalg.norm(vec2)
        if norm1 == 0 or norm2 == 0:
            return 0.0
        similarity = np.dot(vec1, vec2) / (norm1 * norm2)
```

```

        # On renvoie une valeur entre 0 et 1
        return max(0.0, similarity)

# Module pour les données symboliques (ex. mots, concepts)
class ModuleSynergySym(ModuleSynergy):
    def compute_similarity(self, entity1: Entity, entity2: Entity) -> float:
        if entity1.modality != 'symbolic' or entity2.modality != 'symbolic':
            raise ValueError("ModuleSynergySym ne peut traiter que des entités symboliques.")
        # Exemple simple : 1.0 si les données sont identiques, 0 sinon.
        return 1.0 if entity1.data == entity2.data else 0.0

#####
# 2. SynergyDispatcher
#####

class SynergyDispatcher:
    """
    Classe qui répartit le calcul de la synergie selon le type des entités.
    Elle fait appel aux modules spécialisés pour les entités subsymboliques et symboliques.
    """

    def __init__(self, synergy_sub: ModuleSynergy, synergy_sym: ModuleSynergy):
        self.sub = synergy_sub
        self.sym = synergy_sym

    def get_synergy(self, entity1: Entity, entity2: Entity) -> float:
        # Si les deux entités sont subsymboliques
        if entity1.modality == "subsymbolic" and entity2.modality == "subsymbolic":
            return self.sub.compute_similarity(entity1, entity2)
        # Si les deux entités sont symboliques
        elif entity1.modality == "symbolic" and entity2.modality == "symbolic":
            return self.sym.compute_similarity(entity1, entity2)
        else:
            # Pour un cas cross-modal, on peut par exemple renvoyer la moyenne des scores obtenus sur les deux module
            scores = []
            try:
                scores.append(self.sub.compute_similarity(entity1, entity2))
            except Exception:
                pass
            try:
                scores.append(self.sym.compute_similarity(entity1, entity2))
            except Exception:
                pass
            return np.mean(scores) if scores else 0.0

#####
# 3. SCNCore (Version Simplifiée)
#####

class SCNCore:
    """
    Le cœur du SCN qui gère la matrice des pondérations  $\omega$  et orchestre la mise à jour.
    """

    def __init__(self, entities: List[Entity], eta: float = 0.1, tau: float = 0.2):
        self.entities = entities
        self.N = len(entities)

```

```

self.eta = eta
self.tau = tau
# Initialisation de la matrice  $\omega$  avec de petites valeurs aléatoires (symétrique)
self.omega = np.random.uniform(0.01, 0.05, size=(self.N, self.N))
self.omega = (self.omega + self.omega.T) / 2.0
# Instanciation du dispatcher pour calculer la synergie
self.dispatcher = SynergyDispatcher(ModuleSynergySub(), ModuleSynergySym())

def compute_synergy(self, i: int, j: int) -> float:
    """
    Calcule  $S(i,j)$  en déléguant au dispatcher.
    """
    return self.dispatcher.get_synergy(self.entities[i], self.entities[j])

def update_weights(self):
    """
    Met à jour la matrice  $\omega$  selon la règle additive :
     $\omega(t+1) = \omega(t) + \eta [ S(i,j) - \tau \omega(t) ]$ 
    """
    new_omega = self.omega.copy()
    for i in range(self.N):
        for j in range(i + 1, self.N):
            S_ij = self.compute_synergy(i, j)
            delta = self.eta * (S_ij - self.tau * self.omega[i, j])
            new_val = self.omega[i, j] + delta
            # Optionnel : clamp pour éviter un dépassement excessif
            new_val = min(new_val, 1.0)
            new_omega[i, j] = new_val
            new_omega[j, i] = new_val # symétrie
    self.omega = new_omega

def run_iterations(self, iterations: int = 10):
    for t in range(iterations):
        self.update_weights()
        print(f"Itération {t+1}: moyenne de  $\omega = \{np.mean(self.omega):.4f\}$ ")

#####
# 4. Fonction pour la Mise à Jour dans un SCN Distribué
#####

def is_local(entity: Entity, local_block_ids: List[str]) -> bool:
    """
    Simule une fonction qui détermine si une entité appartient au bloc local.
    Ici, on suppose que l'identifiant de la modalité peut servir à cet effet.
    """
    return entity.id in local_block_ids

def apply_inhibition_saturation(w_value: float, i: int, j: int) -> float:
    """
    Applique éventuellement un mécanisme d'inhibition ou de saturation.
    Pour cet exemple, on effectue simplement un clamping à 1.0.
    """
    return min(w_value, 1.0)

# Exemple d'interface réseau simplifiée pour la communication inter-blocs.
class NetInterface:

```

```

def request_synergy_other_block(self, entity1: Entity, entity2: Entity) -> float:
    """
    Simule une requête pour obtenir  $S(i, j)$  depuis un autre bloc.
    Dans une implémentation réelle, cette méthode enverrait une requête réseau.
    """
    # Pour simplifier, on renvoie une valeur fixe (par exemple, 0.5)
    return 0.5

def send_local_updates(self, block_id: str, w_local: Dict[Tuple[int, int], float]):
    """
    Simule l'envoi des mises à jour locales vers un orchestrateur central.
    """
    print(f"Bloc {block_id}: envoi des mises à jour, nombre de liens mis à jour = {len(w_local)}")

def scn_distributed_iteration(block_id: str,
                             w_local: Dict[Tuple[int, int], float],
                             local_links: List[Tuple[int, int]],
                             synergy_module: SynergyDispatcher,
                             update_rule,
                             entities: List[Entity],
                             net_interface: NetInterface):
    """
    Fonction simulant une itération de mise à jour dans un SCN distribué.

    Paramètres:
    block_id : identifiant du bloc.
    w_local : dictionnaire local des pondérations  $\omega$  pour les liaisons gérées par ce bloc.
    local_links : liste des paires  $(i, j)$  appartenant à ce bloc.
    synergy_module : instance de SynergyDispatcher pour calculer  $S(i, j)$  localement.
    update_rule : fonction qui applique la règle DSL, par exemple:
        update_rule(w_old, s_val) =  $w_{old} + \eta (s_{val} - \tau * w_{old})$ 
    entities : liste globale des entités.
    net_interface : interface pour échanger les mises à jour avec d'autres blocs.
    """
    for (i, j) in local_links:
        # Détermine si l'entité j est locale au bloc courant.
        # Pour cet exemple, nous supposons que la liste local_links ne contient que des paires locales.
        s_val = synergy_module.get_synergy(entities[i], entities[j])
        w_old = w_local.get((i, j), 0.0)
        w_new = update_rule(w_old, s_val)
        w_new = apply_inhibition_saturation(w_new, i, j)
        w_local[(i, j)] = w_new

    # Envoi des mises à jour locales via l'interface réseau
    net_interface.send_local_updates(block_id, w_local)

#####
# 5. Fonction d'Update Rule DSL (Exemple Additif)
#####

def update_rule_additive(w_old: float, s_val: float, eta: float = 0.1, tau: float = 0.2) -> float:
    """
    Applique la règle additive :  $w_{new} = w_{old} + \eta (s_{val} - \tau * w_{old})$ 
    """
    return w_old + eta * (s_val - tau * w_old)

```

```
#####
# 6. Exemple d'Utilisation Globale
#####
```

```
def main():
    # Création d'une liste d'entités (certaines subsymboliques et certaines symboliques)
    entities = [
        Entity("E1", "subsymbolic", [0.2, 0.4, 0.6]),
        Entity("E2", "subsymbolic", [0.1, 0.3, 0.5]),
        Entity("E3", "subsymbolic", [0.25, 0.35, 0.45]),
        Entity("E4", "symbolic", "chat"),
        Entity("E5", "symbolic", "chat"),
        Entity("E6", "symbolic", "chien")
    ]

    # Initialisation du SCNCORE pour une utilisation non distribuée
    scn_core = SCNCORE(entities, eta=0.1, tau=0.2)
    print("Matrice  $\omega$  initiale:")
    print(np.round(scn_core.omega, 4))

    scn_core.run_iterations(iterations=5)
    print("Matrice  $\omega$  finale:")
    print(np.round(scn_core.omega, 4))

    # Exemple de mise à jour distribuée
    # Supposons que le bloc local gère les entités E1, E2, E3 (indices 0,1,2)
    local_block_ids = ["E1", "E2", "E3"]
    local_links = [(0, 1), (0, 2), (1, 2)]
    # Initialisation d'un store local des pondérations pour ce bloc
    w_local = {(i, j): 0.05 for (i, j) in local_links}
    # Utilisation d'une instance de SynergyDispatcher (dépendant de la modalité)
    dispatcher = SynergyDispatcher(ModuleSynergySub(), ModuleSynergySym())
    # Création d'une interface réseau simulée
    net_interface = NetInterface()
    print("\nMise à jour distribuée pour le bloc 'Bloc_A' sur les entités locales E1, E2, E3:")
    scn_distributed_iteration("Bloc_A", w_local, local_links, dispatcher, update_rule_additive, entities, net_interface)
    print("Store local w après update:")
    print({k: round(v, 4) for k, v in w_local.items()})

if __name__ == "__main__":
    main()
```

Explications Détaillées

1. La Boucle de Mise à Jour dans le SCNCORE

- **Calcul de la Synergie $S(i, j)$:**

La méthode `compute_synergy` du **SCNCORE** fait appel au **SynergyDispatcher**, lequel, en fonction des modalités des entités i et j , appelle le module approprié (*ModuleSynergySub* ou *ModuleSynergySym*). Ainsi, la logique de calcul de la synergie est encapsulée et séparée du reste du système.

- **Application de la Règle DSL :**

La méthode *update_weights* parcourt toutes les paires (i, j) avec $i < j$, calcule $S(i, j)$, et applique la règle additive

$$\omega_{i,j}(t + 1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)]$$

Le résultat est ensuite éventuellement passé par une fonction de saturation (ici, on impose un maximum de 1.0).

- **Cycle d'Itération :**

La méthode *run_iterations* effectue plusieurs itérations de mise à jour et affiche la moyenne de la matrice ω afin de suivre la progression globale.

2. La Fonction *scn_distributed_iteration*

Cette fonction simule le comportement d'un bloc dans un environnement distribué. Pour chaque liaison locale (définie dans *local_links*), la fonction :

- Vérifie si la paire est locale (pour simplifier, nous considérons ici que toutes les paires sont locales, mais la fonction pourrait être étendue pour interroger d'autres blocs via *net_interface*).
- Calcule $S(i, j)$ via le module de synergie (dispatcher).
- Applique la règle DSL (ici, *update_rule_additive*) pour obtenir la nouvelle valeur de $\omega_{i,j}$.
- Applique un mécanisme d'inhibition/saturation.
- Envoie, en fin de traitement, les mises à jour locales vers un orchestrateur via *net_interface.send_local_updates*.

3. La Classe *SynergyDispatcher*

Cette classe permet de déléguer le calcul de la synergie en fonction du type des entités. Elle teste la modalité de chaque entité et appelle le module correspondant. En cas de modalités mixtes, une stratégie de fusion (ici, une moyenne) est appliquée.

Conclusion

Ce développement illustre concrètement la boucle de mise à jour d'un SCN multimodal. La structure modulaire est assurée par la séparation entre le calcul de la synergie (géré par les modules *ModuleSynergySub*, *ModuleSynergySym* et leur dispatcher associé) et la gestion centrale de la dynamique des pondérations (dans *SCNCore*). De plus, dans un contexte distribué, la fonction *scn_distributed_iteration* démontre comment intégrer la mise à jour locale des pondérations avec la communication inter-blocs via une interface réseau. Cette architecture modulaire permet ainsi d'assurer une grande flexibilité et une évolutivité dans le traitement de données multimodales tout en maintenant la cohérence de la dynamique du SCN.

5.9.2.1. Entités Logiques, Vecteurs, Gestion via la Synergie Multiple

Dans un **Synergistic Connection Network** (SCN) hybride, l'objectif est de traiter simultanément des entités de nature très différente. D'une part, certaines entités sont purement sub-symboliques : il s'agit typiquement de vecteurs numériques issus de descripteurs (ex. : embeddings d'images, audio ou texte). D'autre part, d'autres entités sont purement symboliques et représentent des objets logiques, des formules ou des concepts issus d'ontologies. Enfin, il arrive fréquemment que certaines entités combinent ces deux aspects, comme par exemple un concept lexicalisé qui dispose à la fois d'un embedding vectoriel et d'une définition logique formalisée. Pour répondre à cette hétérogénéité, le SCN doit être capable de calculer plusieurs fonctions de similarité ou de compatibilité, puis de fusionner ces résultats

en un unique score $S(i, j)$ qui guidera l'auto-organisation du réseau. Ce mécanisme est désigné par le terme de **synergie multiple**.

A. Rappel : Sub-symbolique et Symbolique

Sur le plan mathématique, les **entités sub-symboliques** sont généralement représentées par des vecteurs x_i^{sub} appartenant à un espace vectoriel $\mathcal{X}_{\text{sub}} \subseteq \mathbb{R}^d$. Ces représentations numériques proviennent de techniques d'extraction de features, telles que celles issues des réseaux de neurones convolutionnels (CNN) pour les images, ou des méthodes de traitement du signal pour l'audio. La similarité entre deux entités sub-symboliques peut être évaluée par des mesures continues comme la similarité cosinus ou une fonction exponentielle de la distance euclidienne, par exemple :

$$S_{\text{sub}}(i, j) = \exp(-\alpha \|x_i^{\text{sub}} - x_j^{\text{sub}}\|^2) \quad \text{ou} \quad S_{\text{sub}}(i, j) = \frac{x_i^{\text{sub}} \cdot x_j^{\text{sub}}}{\|x_i^{\text{sub}}\| \|x_j^{\text{sub}}\|},$$

où $\alpha > 0$ est un paramètre de réglage.

En revanche, les **entités symboliques** sont représentées par des objets logiques ou des symboles, notés ξ_i^{sym} , qui vivent dans un espace non vectoriel, voire discret, \mathcal{X}_{sym} . Le calcul de leur synergie ne repose pas sur des distances métriques classiques, mais plutôt sur des critères de compatibilité sémantique ou logique. Par exemple, pour deux concepts issus d'une ontologie, on peut définir :

$$S_{\text{sym}}(i, j) = g(\xi_i^{\text{sym}}, \xi_j^{\text{sym}}),$$

où g est une fonction qui évalue le degré de similarité sémantique, pouvant être basée sur des mesures d'alignement ou de co-occurrence dans un graphe conceptuel.

B. Synergie Multiple : Principe et Fusion des Scores

Face à la nécessité de gérer des entités hétérogènes, le SCN hybride définit une **synergie globale** pour un couple d'entités (i, j) en combinant les contributions issues des aspects sub-symboliques et symboliques. Mathématiquement, cette combinaison se traduit par :

$$S_{\text{hyb}}(i, j) = \alpha S_{\text{sub}}(i, j) + \beta S_{\text{sym}}(i, j),$$

où α et β sont des poids non négatifs qui déterminent l'importance relative de chaque modalité. Le choix des coefficients α et β peut être fixe ou adaptatif en fonction des caractéristiques des entités :

- **Entités sub-symboliques pures** : on peut imposer $\beta = 0$ afin que seule la contribution vectorielle soit prise en compte.
- **Entités symboliques pures** : on peut fixer $\alpha = 0$.
- **Entités mixtes** : des valeurs telles que $\alpha = 0.7$ et $\beta = 0.3$ permettent de donner une importance plus forte à la représentation sub-symbolique tout en conservant une influence significative de la dimension symbolique.

D'autres stratégies de fusion sont envisageables. Par exemple, le score global pourrait être défini par le **produit** des sous-scores, ou par une fonction non linéaire telle qu'un maximum ou une combinaison pondérée par une fonction softmax. Ces stratégies ont pour objectif de garantir que le score $S(i, j)$ reflète à la fois la proximité dans l'espace vectoriel et la cohérence sémantique ou logique, renforçant ainsi la pertinence de la mise à jour de la matrice ω .

C. Stockage des Entités et Mises à Jour dans un Espace Hybride

Chaque entité \mathcal{E}_i dans un SCN hybride est représentée par une paire de composantes :

$$\mathcal{E}_i = (x_i^{\text{sub}}, \xi_i^{\text{sym}}),$$

où x_i^{sub} est un vecteur dans \mathbb{R}^d et ξ_i^{sym} est une représentation symbolique. La fonction de synergie globale $S_{\text{hyb}}(i, j)$ est alors utilisée dans la règle de mise à jour de la matrice des pondérations, qui demeure inchangée par rapport à la formulation classique du DSL :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S_{\text{hyb}}(i, j) - \tau \omega_{i,j}(t)].$$

Ici, l'apport de la synergie multiple permet d'enrichir le processus d'auto-organisation en tenant compte des deux dimensions de l'information. Ce mécanisme assure que la connexion entre deux entités est renforcée si elles présentent à la fois une similarité vectorielle élevée et une forte compatibilité sémantique.

Sur le plan **ingénierie**, cette approche se traduit par l'implémentation d'un module de type *ModuleSynergyHyb* ou, de manière plus générale, par un **dispatcher** capable d'orienter le calcul vers les modules spécialisés appropriés, puis de fusionner les résultats selon une stratégie choisie (par exemple, la somme pondérée présentée ci-dessus).

Conclusion

La gestion des entités logiques et sub-symboliques dans un SCN hybride repose sur une approche de **synergie multiple** qui combine plusieurs fonctions de similarité. Chaque entité \mathcal{E}_i est dotée d'une représentation duale, x_i^{sub} pour les aspects sub-symboliques et ξ_i^{sym} pour les aspects symboliques, et la synergie globale entre deux entités est obtenue par une combinaison linéaire (ou autre) des sous-scores :

$$S_{\text{hyb}}(i, j) = \alpha f_{\text{sub}}(x_i^{\text{sub}}, x_j^{\text{sub}}) + \beta g_{\text{sym}}(\xi_i^{\text{sym}}, \xi_j^{\text{sym}}).$$

D'un point de vue mathématique, cette approche enrichit la définition habituelle de $S(i, j)$ en la rendant capable de capturer la dualité des informations disponibles, tandis que, d'un point de vue ingénierie, elle s'appuie sur des modules spécialisés qui se combinent dans un schéma modulaire et extensible. Ce modèle hybride permet ainsi au SCN de former des clusters qui sont cohérents à la fois sur le plan des représentations numériques et sur le plan des relations sémantiques ou logiques, garantissant une auto-organisation plus robuste et pertinente dans des environnements complexes et multimodaux.

5.9.2.2. Rapide cas d'usage, mini-dataset

Dans un SCN hybride, l'objectif est de combiner des entités provenant de deux mondes différents : les entités sub-symboliques qui sont représentées par des vecteurs numériques et vivent dans un espace continu $\mathcal{X}_{\text{sub}} \subseteq \mathbb{R}^d$, et les entités symboliques qui se présentent sous forme de règles logiques ou d'axiomes et qui n'ont pas de représentation vectorielle directe. Dans de nombreuses applications, il est intéressant de fusionner ces deux dimensions afin de mieux capturer la richesse des informations disponibles sur les entités.

A. Description du Mini-Dataset

Pour illustrer le concept, considérons un mini-dataset composé de 10 entités réparties comme suit :

24. Entités Sub-symboliques (les entités \mathcal{E}_1 à \mathcal{E}_5)

Chaque entité est représentée par un vecteur de dimension 4. Par exemple :

- $\mathcal{E}_1: (0.2, 0.8, 0.1, 0.1)$
- $\mathcal{E}_2: (0.1, 0.75, 0.05, 0.3)$

- $\mathcal{E}_3: (0.6, 0.2, 0.05, 0.15)$
- $\mathcal{E}_4: (0.3, 0.6, 0.2, 0.1)$
- $\mathcal{E}_5: (0.25, 0.65, 0.15, 0.2)$
- **Entités Symboliques** (les entités \mathcal{E}_6 à \mathcal{E}_{10})

Chaque entité est représentée par un ensemble simple de règles ou d'axiomes. Par exemple :

- $\mathcal{E}_6: \{A \rightarrow B, B \wedge C \rightarrow D\}$
- $\mathcal{E}_7: \{X \rightarrow Y, \neg(Z \wedge Y)\}$
- $\mathcal{E}_8: \{P \rightarrow Q\}$
- $\mathcal{E}_9: \{A \rightarrow B, B \rightarrow C\}$
- $\mathcal{E}_{10}: \{M \rightarrow N, N \wedge O \rightarrow P\}$

Ce mini-dataset représente ainsi un échantillon réduit d'entités où la première partie apporte une information continue (ex. issue d'un moteur NLP ou d'un CNN), et la seconde des informations purement logiques. L'idée est d'observer comment le SCN peut, à travers le calcul d'une **synergie multiple**, regrouper ces entités selon des critères qui tiennent compte simultanément de la proximité vectorielle et de la compatibilité sémantique.

B. Principe de la Synergie Multiple

Pour chaque paire d'entités (i, j) , on définit la synergie globale $S(i, j)$ comme une combinaison des deux sous-synergies :

$$S(i, j) = \alpha S_{\text{sub}}(i, j) + \beta S_{\text{sym}}(i, j),$$

où :

- $S_{\text{sub}}(i, j)$ est la similarité entre les représentations vectorielles (ex. similarité cosinus) pour les entités sub-symboliques ;
- $S_{\text{sym}}(i, j)$ est un score de compatibilité logique ou sémantique, qui peut être calculé par le recouplement ou l'alignement des règles ;
- α et β sont des coefficients (non négatifs) qui pondèrent l'influence de chaque modalité. Par exemple, pour des entités mixtes, on pourra choisir $\alpha = 0.7$ et $\beta = 0.3$.

Si les entités sont de même nature (pures sub-symboliques ou purement symboliques), on peut adapter ces coefficients en conséquence (ex. $\beta = 0$ pour des paires sub-symboliques pures).

C. La Règle de Mise à Jour DSL dans le Contexte Hybride

La mise à jour de la matrice des pondérations ω suit la règle classique du DSL, c'est-à-dire :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i, j) - \tau \omega_{i,j}(t)],$$

où le score $S(i, j)$ est ici le score hybride $S(i, j) = \alpha S_{\text{sub}}(i, j) + \beta S_{\text{sym}}(i, j)$. Ainsi, le renforcement de la liaison entre deux entités repose sur l'accord simultané de leurs représentations vectorielles et de leurs composantes logiques. L'objectif est que la dynamique renforce les liens entre entités « proches » sur les deux dimensions, formant ainsi des clusters cohérents tant sur le plan neuronal (sub-symbolique) que sur le plan sémantique (symbolique).

Nous proposons ci-dessous un code Python qui illustre la gestion d'un mini-dataset hybride par un SCN. Ce code définit :

- Une classe **Entity** qui intègre une représentation sub-symbolique et/ou symbolique.
- Deux modules de synergie, l'un pour les entités subsymboliques et l'autre pour les entités symboliques.
- Un module hybride qui fusionne ces deux scores à l'aide des coefficients α et β .
- Un **SCNCore** qui applique la règle de mise à jour DSL sur la matrice des pondérations ω à partir des scores hybrides.

```
import numpy as np
from abc import ABC, abstractmethod
from typing import Any, List

#####
# 1. Classe Entity pour le Mini-dataset Hybride
#####

class Entity:
    """
    Représente une entité hybride dans le SCN.

    Chaque entité combine une représentation sub-symbolique (vecteur)
    et une représentation symbolique (ensemble de règles ou axiomes).

    Attributes:
        id (str): Identifiant unique.
        modality (str): 'hybrid' pour les entités combinées.
        sub_data (List[float]): Vecteur numérique (peut être None pour entités purement symboliques).
        sym_data (Any): Représentation symbolique (par exemple, une liste ou un set de règles).
    """
    def __init__(self, id: str, sub_data: List[float] = None, sym_data: Any = None):
        self.id = id
        self.modality = 'hybrid'
        self.sub_data = sub_data
        self.sym_data = sym_data

#####
# 2. Modules de Synergie pour les Composantes
#####

class ModuleSynergySub(ABC):
    @abstractmethod
    def compute_similarity(self, vec1: List[float], vec2: List[float]) -> float:
        pass

class CosineSimilarity(ModuleSynergySub):
    def compute_similarity(self, vec1: List[float], vec2: List[float]) -> float:
        v1 = np.array(vec1)
        v2 = np.array(vec2)
        norm1 = np.linalg.norm(v1)
        norm2 = np.linalg.norm(v2)
        if norm1 == 0 or norm2 == 0:
```

```

        return 0.0
    similarity = np.dot(v1, v2) / (norm1 * norm2)
    return max(0.0, similarity)

class ModuleSynergySym(ABC):
    @abstractmethod
    def compute_similarity(self, rules1: Any, rules2: Any) -> float:
        pass

class SimpleRuleMatch(ModuleSynergySym):
    def compute_similarity(self, rules1: List[str], rules2: List[str]) -> float:
        """
        Calcule la similarité comme le rapport du nombre de règles communes sur le nombre total de règles.
        """
        if not rules1 or not rules2:
            return 0.0
        set1, set2 = set(rules1), set(rules2)
        common = set1.intersection(set2)
        total = set1.union(set2)
        return len(common) / len(total) if total else 0.0

#####
# 3. Module de Synergie Hybride (Dispatcher)
#####

class ModuleSynergyHyb:
    """
    Calcule le score hybride  $S(i, j)$  pour des entités hybrides.

    La synergie globale est définie par :
    
$$S_{hyb}(i, j) = \alpha * S_{sub}(i, j) + \beta * S_{sym}(i, j)$$

    """
    def __init__(self, alpha: float = 0.7, beta: float = 0.3):
        self.alpha = alpha
        self.beta = beta
        self.sub_module = CosineSimilarity()
        self.sym_module = SimpleRuleMatch()

    def compute_hybrid_similarity(self, entity1: Entity, entity2: Entity) -> float:
        # Calcul de la similarité sub-symbolique
        if entity1.sub_data is not None and entity2.sub_data is not None:
            s_sub = self.sub_module.compute_similarity(entity1.sub_data, entity2.sub_data)
        else:
            s_sub = 0.0
        # Calcul de la similarité symbolique
        if entity1.sym_data is not None and entity2.sym_data is not None:
            s_sym = self.sym_module.compute_similarity(entity1.sym_data, entity2.sym_data)
        else:
            s_sym = 0.0
        # Fusion linéaire des deux scores
        return self.alpha * s_sub + self.beta * s_sym

#####
# 4. SCNCORE pour le SCN Hybride
#####

```

```

class SCNCORE:
    """
    SCNCORE gère la matrice des pondérations  $\omega$  et la mise à jour selon la règle DSL hybride.
    """

    def __init__(self, entities: List[Entity], eta: float = 0.1, tau: float = 0.2):
        self.entities = entities
        self.N = len(entities)
        self.eta = eta
        self.tau = tau
        # Initialisation de  $\omega$  avec de petites valeurs aléatoires (symétrique)
        self.omega = np.random.uniform(0.01, 0.05, size=(self.N, self.N))
        self.omega = (self.omega + self.omega.T) / 2.0
        # Instanciation du module de synergie hybride
        self.hybrid_module = ModuleSynergyHyb(alpha=0.7, beta=0.3)

    def compute_synergy(self, i: int, j: int) -> float:
        return self.hybrid_module.compute_hybrid_similarity(self.entities[i], self.entities[j])

    def update_weights(self):
        """
        Met à jour  $\omega$  selon la règle DSL hybride.
        """
        new_omega = self.omega.copy()
        for i in range(self.N):
            for j in range(i + 1, self.N):
                S_ij = self.compute_synergy(i, j)
                delta = self.eta * (S_ij - self.tau * self.omega[i, j])
                new_val = self.omega[i, j] + delta
                # Clamping à 1.0
                new_val = min(new_val, 1.0)
                new_omega[i, j] = new_val
                new_omega[j, i] = new_val
        self.omega = new_omega

    def run_iterations(self, iterations: int = 10):
        for t in range(iterations):
            self.update_weights()
            print(f"Itération {t+1} : moyenne de  $\omega$  = {np.mean(self.omega):.4f}")

#####
# 5. Mini-dataset et Exemple d'Utilisation
#####

def main():
    # Création d'un mini-dataset de 10 entités hybrides
    # Entités 1 à 5 : sub-symboliques (vecteurs de dimension 4)
    entities = [
        Entity("E1", sub_data=[0.2, 0.8, 0.1, 0.1]),
        Entity("E2", sub_data=[0.1, 0.75, 0.05, 0.3]),
        Entity("E3", sub_data=[0.6, 0.2, 0.05, 0.15]),
        Entity("E4", sub_data=[0.3, 0.6, 0.2, 0.1]),
        Entity("E5", sub_data=[0.25, 0.65, 0.15, 0.2]),
        # Entités 6 à 10 : symboliques (ensembles de règles)
        Entity("E6", sym_data=["A->B", "B^C->D"]),
        Entity("E7", sym_data=["X->Y", "¬(Z^Y)"]),
        Entity("E8", sym_data=["P->Q"]),
    ]

```

```

Entity("E9", sym_data=["A->B", "B->C"]),
Entity("E10", sym_data=["M->N", "NΛO->P"])
]

# Exemple : certaines entités mixtes pourraient avoir les deux types de données.
# Ici, nous ne créons que des entités purement sub-symboliques ou symboliques pour la simplicité.

# Initialisation du SCNCORE pour le mini-dataset hybride
scn_core = SCNCORE(entities, eta=0.1, tau=0.2)
print("Matrice ω initiale:")
print(np.round(scn_core.omega, 4))

# Exécute 10 itérations de mise à jour
scn_core.run_iterations(iterations=10)

print("Matrice ω finale:")
print(np.round(scn_core.omega, 4))

if __name__ == "__main__":
    main()

```

Explications Complémentaires

A. Mini-Dataset Hybride

Le mini-dataset contient 10 entités réparties de la manière suivante :

- Les entités **sub-symboliques** (E1 à E5) sont définies par des vecteurs en \mathbb{R}^4 . Ces vecteurs pourraient provenir d'un modèle d'embedding ou d'un descripteur d'image/audio.
- Les entités **symboliques** (E6 à E10) sont définies par des listes de chaînes représentant des règles logiques ou des axiomes. Dans un contexte réel, ces données pourraient être extraites d'une ontologie ou d'un système de raisonnement logique.

B. Calcul de la Synergie Multiple

Le module **ModuleSynergyHyb** combine le score de similarité sub-symbolique, calculé par la similarité cosinus, et le score symbolique, évalué ici par un simple ratio de règles communes. Le score hybride est obtenu via une combinaison linéaire pondérée par α et β . Par exemple, avec $\alpha = 0.7$ et $\beta = 0.3$, si deux entités possèdent un score subsymbolique de 0.8 et un score symbolique de 1.0, le score hybride sera :

$$S_{\text{hyb}} = 0.7 \times 0.8 + 0.3 \times 1.0 = 0.56 + 0.3 = 0.86.$$

Ce score est ensuite utilisé dans la mise à jour des pondérations.

C. Règle de Mise à Jour DSL Hybride

La mise à jour de $\omega_{i,j}$ se fait toujours selon la formule :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S_{\text{hyb}}(i,j) - \tau \omega_{i,j}(t)],$$

ce qui garantit que les liens se renforcent lorsque la synergie est élevée et se décroissent lorsque le score est faible, tout en maintenant une dynamique stable.

D. Observation des Résultats

En exécutant le script, vous pourrez observer la matrice ω évoluer au fil des itérations. On s’attend, par exemple, à voir que les entités sub-symboliques similaires (ex. E1 et E2) voient leur lien se renforcer, tandis que les entités symboliques identiques (E6 et E9 pourraient présenter une forte similarité si leurs règles se recoupent) voient également leur connexion s’accroître. Par ailleurs, des liens entre des entités de modalités différentes (par exemple, entre E2 et une entité symbolique) recevront un score fusionné qui, même s’il est modeste, pourra créer un pont hybride entre les deux mondes.

Conclusion

Le cas d’usage présenté dans cette section illustre de manière concrète la gestion des entités hybrides dans un SCN. À partir d’un mini-dataset composé d’entités sub-symboliques (vecteurs) et symboliques (règles logiques), le SCN combine plusieurs fonctions de synergie pour calculer un score global $S(i, j)$. Ce score hybride, intégré dans la règle de mise à jour DSL, permet au réseau de renforcer les liens entre entités similaires sur plusieurs dimensions simultanément. L’implémentation en Python montre comment structurer ce processus à l’aide de modules dédiés – le module de synergie hybride, le SCNCORE et la règle DSL – afin d’assurer une auto-organisation cohérente et robuste dans un contexte multimodal.

5.9.3.1. Rappeler la Distribution Possible, la Synchro Inter-Robots

Dans les systèmes robotiques ou multi-agents, un SCN (Synergistic Connection Network) permet de modéliser et de renforcer la coopération entre chaque robot ou agent à travers des liaisons pondérées $\omega_{i,j}$. Ces pondérations reflètent la capacité de deux entités à collaborer et s’auto-organiser en fonction de leurs états locaux, lesquels intègrent des informations telles que la position, les capteurs, et même des objectifs spécifiques. Dans ce contexte, la distribution des $\omega_{i,j}$ au sein du réseau et la synchronisation inter-robots sont des éléments essentiels pour assurer une cohérence globale, même dans des environnements dynamiques et distribués.

Nous détaillons ci-après les principaux aspects de cette problématique.

A. Pourquoi un SCN Distribué pour la Robotique ou le Multi-Agent ?

1. Multiplicité et Autonomie des Robots

Dans un essaim composé de dizaines voire de centaines de robots, chacun possède un état local caractérisé par sa position, son orientation, ses mesures capteurs et ses objectifs propres. Mathématiquement, on considère l’ensemble des robots comme un ensemble d’entités $\mathcal{E} = \{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ et chaque lien $\omega_{i,j}$ représente la force de coopération entre l’agent i et l’agent j . La synergie $S(i, j)$ peut être définie, par exemple, en fonction de la proximité géographique ou de la complémentarité dans la tâche (ex. couverture de zone, échange d’informations), et la mise à jour des $\omega_{i,j}$ via une règle DSL permet de renforcer les connexions efficaces et de diminuer celles qui sont moins pertinentes.

2. Décentralisation et Scalabilité

Pour un essaim de grande taille, il est impraticable de disposer d’un serveur central collectant toutes les informations et calculant globalement l’ensemble des $\omega_{i,j}$. Ainsi, dans un SCN distribué, chaque robot gère localement une portion des liens, souvent ceux qui concernent ses voisins immédiats. La synchronisation inter-robots s’effectue alors par échanges périodiques de mises à jour ou de résumés d’état. Du point de vue mathématique, on peut modéliser le système comme un ensemble de sous-systèmes autonomes qui communiquent via des opérateurs de synchronisation, de manière à obtenir un équilibre global malgré des retards et une mise à jour asynchrone.

3. Adaptation à un Environnement Dynamique

Les robots évoluent dans des environnements changeants et leur état se modifie en continu. Un SCN distribué doit donc permettre une mise à jour dynamique et continue des pondérations ω . Le fait que chaque robot ne connaisse qu’une partie locale du réseau, avec des échanges inter-robots ponctuels, rend le système non-autonome. Cela signifie

que la fonction de mise à jour F appliquée aux ω dépend de variables externes et temporelles, rendant la convergence vers un état fixe moins réaliste qu'une adaptation continue aux changements. L'important est alors d'assurer que les communications et la synchronisation permettent de maintenir une cohérence globale acceptable, même si la matrice ω évolue en permanence.

B. Synchronisation Inter-Robots et Paramètres de Mise à Jour

1. Protocole de Communication et Synchronisation

Dans un environnement distribué, la synchronisation inter-robots repose sur un protocole de communication qui permet d'échanger les mises à jour des pondérations ou des informations résumées sur les états locaux. Typiquement, chaque robot diffuse périodiquement des messages contenant ses mises à jour $\Delta\omega_{i,j}$ aux agents voisins ou à un orchestrateur local. Ces messages peuvent être envoyés de manière asynchrone, ce qui induit des délais et des déphasages dans la synchronisation. Sur le plan mathématique, on peut considérer que la mise à jour locale d'un robot i est donnée par un opérateur F_i et que la synchronisation globale se fait par une agrégation de ces opérateurs, par exemple :

$$\omega_{i,j}(t+1) = F_i(\omega_{i,j}(t)) + \Delta_{comm}(t),$$

où $\Delta_{comm}(t)$ représente la correction apportée par les échanges inter-robots pour harmoniser les valeurs de $\omega_{i,j}$ dans l'ensemble du réseau. Des mécanismes de consensus (comme ceux inspirés des algorithmes de Paxos ou de Raft) peuvent être utilisés pour minimiser les incohérences dues aux retards.

2. Paramètres de Mise à Jour : η et τ

La règle de mise à jour classique est donnée par :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)].$$

Ici :

- η est le **taux d'apprentissage** qui détermine la réactivité du système aux changements de synergie. Un η élevé rend le système très sensible aux variations, mais peut également conduire à des oscillations, notamment en présence de retards de synchronisation.
- τ est le **coefficient de décroissance** qui sert de régulateur pour éviter une croissance excessive des pondérations. Dans un environnement robotique dynamique, un choix judicieux de τ permet de modérer les variations en fonction des perturbations externes (par exemple, des changements rapides de position).

Le réglage de ces paramètres doit prendre en compte le niveau de bruit dans les communications et les délais de synchronisation. Une réactivité trop forte (valeur élevée de η) peut compromettre la stabilité en raison des communications asynchrones entre robots.

3. Convergence et Dynamique Continue

Contrairement à des systèmes statiques qui cherchent une convergence vers un état fixe, les systèmes robotiques fonctionnent dans des environnements en perpétuel changement. Ainsi, la matrice ω est amenée à évoluer continuellement. Le protocole de synchronisation doit alors assurer que, malgré des retards et des mises à jour locales asynchrones, l'ensemble du SCN conserve une **cohérence globale** suffisante pour permettre la formation de clusters opérationnels. Par exemple, dans un essaim de drones surveillant une zone, même si les valeurs de $\omega_{i,j}$ changent en fonction des mouvements et des échanges, la synchronisation assure que les drones restent regroupés en sous-équipes effectuant des missions spécifiques.

C. Impact sur la Coordination et la Répartition des Tâches

L'architecture distribuée d'un SCN offre plusieurs avantages pour la coordination inter-robots :

- **Formation de Groupes** : Les robots dont les synergies sont élevées se regroupent, formant des clusters qui correspondent à des équipes collaborant sur des tâches communes (par exemple, surveillance d'une zone, cartographie, ou transport d'objets).
- **Adaptation Dynamique** : En fonction des changements d'état (position, orientation, capteurs) et des échanges inter-robots, le réseau s'adapte en temps réel. Un robot peut changer de cluster s'il se rapproche d'un autre groupe, ce qui permet une répartition flexible des missions.
- **Robustesse Distribuée** : Chaque robot ne gère qu'une partie locale des informations et les synchronise avec ses voisins. Cela rend le système moins vulnérable à une panne centralisée et facilite la redondance et la reprise après une défaillance.

Sur le plan **mathématique**, on modélise le système comme un ensemble d'équations différentielles discrètes couplées, où chaque robot applique localement une mise à jour selon la règle DSL, et où les corrections de synchronisation agissent comme des perturbations qui tendent à harmoniser l'ensemble du réseau. Cette approche permet de garantir, malgré l'asynchronie et les délais de communication, que le SCN atteint un état de coordination fonctionnelle.

Conclusion

L'application d'un SCN distribué dans le domaine de la robotique ou des systèmes multi-agents repose sur deux axes essentiels : d'une part, la capacité de chaque robot à gérer localement ses liaisons $\omega_{i,j}$ via un état propre et des échanges périodiques avec ses voisins ; d'autre part, l'existence d'un protocole de synchronisation inter-robots qui permet de consolider ces informations de manière asynchrone tout en maintenant une cohérence globale.

Du point de vue **mathématique**, la mise à jour de la matrice ω se fait via des opérateurs locaux F_i couplés par des termes de synchronisation correctifs, garantissant ainsi un équilibre dynamique dans un environnement non-stationnaire. Du point de vue **ingénierie**, cette architecture distribuée assure que chaque robot peut adapter sa participation à l'auto-organisation en temps réel, facilitant la formation spontanée de clusters et la répartition efficace des tâches dans un essaim.

En résumé, la distribution possible des pondérations et la synchronisation inter-robots constituent des mécanismes clés pour que l'auto-organisation d'un SCN soit à la fois robuste, flexible et adaptée aux environnements dynamiques, garantissant ainsi une coordination efficace et distribuée au sein d'un essaim robotique ou d'un système multi-agent.

5.9.3.2. Visualisation d'un SCN en Essaim Robotique

I. Analyse et Motivations de la Visualisation d'un SCN en Essaim Robotique

A. Objectifs et Bénéfices

Dans un environnement multi-agent, la matrice des pondérations ω issue du SCN encode l'intensité des interactions entre robots. Toutefois, lorsqu'on manipule un grand nombre de robots, il devient difficile d'extraire intuitivement la structure du réseau à partir de simples tableaux ou listes de valeurs numériques. La visualisation permet alors de :

25. Comprendre le comportement collectif

Une représentation graphique (par exemple, un graphe en 2D) permet de détecter visuellement la formation de clusters ou de groupes coopératifs. Chaque robot est représenté par un nœud, et les arêtes qui relient deux robots sont colorées ou dont l'épaisseur varie en fonction de la valeur de $\omega_{i,j}$. Ainsi, des arêtes épaisses et de

couleurs vives indiquent une forte coopération, tandis que des arêtes fines ou absentes traduisent une faible synergie.

26. Diagnostiquer et Ajuster les Paramètres

En visualisant l'évolution des liens, un opérateur peut détecter des anomalies (par exemple, des liens excessivement forts ou faibles) qui pourraient résulter d'un mauvais paramétrage ou d'une défaillance du système. Ceci aide à régler les paramètres de la dynamique d'auto-organisation, tels que le taux d'apprentissage η et le coefficient de décroissance τ .

27. Suivre l'Adaptation Dynamique

Dans un environnement robotique dynamique, la disposition des robots change en continu. La visualisation permet de suivre en temps réel comment les robots se regroupent, se séparent ou se réorganisent en fonction de leur position et des échanges de données (par exemple, les retours capteurs ou les informations de mission).

B. Approches de Visualisation

Plusieurs méthodes de représentation graphique sont envisageables :

28. Graphe en 2D ou 3D

On positionne chaque robot en fonction de ses coordonnées physiques (par exemple, sur un plan réel) et on trace des arêtes entre les robots. La couleur et l'épaisseur des arêtes sont déterminées par la valeur de $\omega_{i,j}$. Un *layout force-directed* peut être utilisé pour obtenir une représentation qui révèle la structure topologique des interactions.

29. Heatmap ou Matrice de Pondérations

Pour un essaim de taille modérée, une heatmap de la matrice ω fournit une vision globale des niveaux d'interaction. Les valeurs élevées apparaissent en couleurs vives, ce qui facilite l'identification des clusters internes.

30. Vue de Clusters

Un algorithme de détection de communautés (ex. Louvain) peut être appliqué pour extraire les groupes d'agents fortement connectés. La visualisation se fait alors par des couleurs ou des labels indiquant à quel cluster appartient chaque robot.

C. Exemple d'Application : Essaim d'une Douzaine de Robots

Considérons un essaim de 12 robots terrestres évoluant dans une zone. Chaque robot, assimilé à une entité \mathcal{E}_i , possède des liaisons $\omega_{i,j}$ avec les autres robots, qui évoluent selon la dynamique d'auto-organisation décrite par :

$$\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)].$$

Pour faciliter la compréhension du comportement collectif, on peut visualiser le réseau de robots à intervalles réguliers. Ainsi, sur une carte 2D, chaque robot est représenté par un nœud (placé selon sa position réelle ou simulée), et les arêtes entre les robots sont dessinées avec une épaisseur ou une couleur proportionnelle à la valeur de $\omega_{i,j}$. Cette représentation permet de voir par exemple :

- Des clusters où des robots se regroupent pour effectuer une mission commune (fortes interactions),
- Des robots isolés ou en dérive, dont les liens sont faibles,
- La répartition spatiale des sous-groupes et l'évolution de la coordination dans le temps.

II. Implémentation Python pour la Visualisation d'un Essaim de 12 Robots

Nous proposons ci-après un programme Python complet qui simule un essaim de 12 robots. Le programme crée des positions aléatoires pour chaque robot sur un plan, génère une matrice de pondérations ω simulant des interactions (par exemple, basée sur la distance entre robots), et affiche le graphe du réseau en utilisant la bibliothèque **NetworkX** et **Matplotlib**.

Le programme intègre les étapes suivantes :

- Génération des positions des robots,
- Calcul d'une matrice de pondérations simulée, où une valeur élevée de $\omega_{i,j}$ correspond à des robots proches ou en coopération,
- Visualisation du graphe avec des arêtes dont l'épaisseur et la couleur reflètent la force $\omega_{i,j}$.

```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from typing import Any, List, Tuple # Importation de Tuple et d'autres types

# Pour assurer la reproductibilité
np.random.seed(42)

#####
# 1. Génération des Positions des Robots
#####

def generate_robot_positions(num_robots: int, area_size: Tuple[float, float] = (100, 100)) -> np.ndarray:
    """
    Génère des positions aléatoires pour num_robots dans une zone de dimensions area_size.
    Renvoie un tableau de forme (num_robots, 2).
    """
    x_positions = np.random.uniform(0, area_size[0], num_robots)
    y_positions = np.random.uniform(0, area_size[1], num_robots)
    return np.column_stack((x_positions, y_positions))

#####
# 2. Calcul de la Matrice de Pondérations
#####

def compute_weight_matrix(positions: np.ndarray, sigma: float = 20.0) -> np.ndarray:
    """
    Calcule une matrice de pondérations  $\omega$  basée sur une fonction gaussienne de la distance.
    Plus deux robots sont proches, plus  $\omega$  est élevé.
    La fonction utilisée est :  $\omega(i,j) = \exp(-d(i,j)^2 / (2 * \sigma^2))$ .
    """
    num_robots = positions.shape[0]
    omega = np.zeros((num_robots, num_robots))
    for i in range(num_robots):
        for j in range(i + 1, num_robots):
            distance = np.linalg.norm(positions[i] - positions[j])
            weight = np.exp(-(distance ** 2) / (2 * sigma ** 2))
            omega[i, j] = weight
            omega[j, i] = weight # symétrie
```

```

    return omega

#####
# 3. Visualisation du SCN
#####

def visualize_scn(positions: np.ndarray, omega: np.ndarray, threshold: float = 0.2):
    """
    Visualise le SCN sous forme de graphe où les nœuds sont les positions des robots.
    Seules les arêtes dont la pondération est supérieure au seuil sont affichées.
    L'épaisseur et la couleur des arêtes varient en fonction de la valeur de  $\omega$ .
    """

    num_robots = positions.shape[0]
    G = nx.Graph()
    # Ajout des nœuds avec leur position
    for i in range(num_robots):
        G.add_node(i, pos=(positions[i, 0], positions[i, 1]))

    # Ajout des arêtes avec les poids
    for i in range(num_robots):
        for j in range(i + 1, num_robots):
            weight = omega[i, j]
            if weight > threshold: # Seuil pour afficher une liaison
                G.add_edge(i, j, weight=weight)

    pos = nx.get_node_attributes(G, 'pos')
    weights = [G[u][v]['weight'] for u, v in G.edges()]

    # Normaliser les poids pour l'épaisseur des arêtes
    max_weight = max(weights) if weights else 1.0
    widths = [4 * (w / max_weight) for w in weights]

    # Choix d'une palette de couleurs (ici, on utilise la colormap viridis)
    edge_colors = [plt.cm.viridis(w) for w in weights]

    plt.figure(figsize=(10, 8))
    nx.draw_networkx_nodes(G, pos, node_size=300, node_color='lightblue')
    nx.draw_networkx_labels(G, pos)
    nx.draw_networkx_edges(G, pos, width=widths, edge_color=edge_colors)
    plt.title("Visualisation d'un SCN en Essaim Robotique")
    plt.axis('off')
    plt.show()

#####
# 4. Programme Principal
#####

def main():
    num_robots = 12
    positions = generate_robot_positions(num_robots, area_size=(100, 100))
    omega = compute_weight_matrix(positions, sigma=20.0)

    # Affichage des positions
    plt.figure(figsize=(8, 6))
    plt.scatter(positions[:, 0], positions[:, 1], c='blue', s=100)
    for i, (x, y) in enumerate(positions):

```

```

plt.text(x, y, f"{i}", fontsize=12, color='white', ha='center', va='center')
plt.title("Positions des robots dans l'essaim")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()

# Visualisation du SCN sous forme de graphe
visualize_scn(positions, omega, threshold=0.2)

if __name__ == "__main__":
    main()

```

Explications

31. Importation des Types

Nous avons ajouté *from typing import Tuple* (ainsi que d'autres types utiles) pour pouvoir utiliser le type *Tuple[float, float]* dans la fonction *generate_robot_positions*.

32. Génération des Positions

La fonction *generate_robot_positions* crée un tableau de positions aléatoires pour un nombre donné de robots dans une zone définie par *area_size*.

33. Calcul de la Matrice de Pondérations

La fonction *compute_weight_matrix* calcule une matrice ω en appliquant une fonction gaussienne sur la distance entre chaque paire de robots. Plus deux robots sont proches, plus la valeur retournée sera proche de 1.

34. Visualisation du SCN

La fonction *visualize_scn* construit un graphe avec NetworkX où chaque robot est un nœud, et les arêtes sont tracées uniquement pour des liaisons dont le poids dépasse un seuil donné. L'épaisseur et la couleur des arêtes sont proportionnelles à la valeur des pondérations, ce qui permet de visualiser la force des connexions.

35. Programme Principal

La fonction *main()* orchestre l'ensemble du processus :

- Génération des positions pour 12 robots,
- Calcul de la matrice ω ,
- Affichage des positions sur un scatter plot,
- Visualisation du graphe du SCN.

Conclusion

Ce cas d'usage illustre comment un SCN appliqué à un essaim robotique permet non seulement de modéliser les interactions et la coopération entre les robots, mais aussi de visualiser en temps réel ou par snapshots la structure émergente du réseau. La visualisation – réalisée ici via un graphe 2D où l'épaisseur et la couleur des arêtes reflètent la force des liaisons $\omega_{i,j}$ – fournit un outil puissant de diagnostic et de suivi de la dynamique d'auto-organisation, essentiel

pour le débogage, l'ajustement des paramètres, et la gestion des missions dans des environnements robotiques complexes.

Le programme Python proposé offre une simulation simple d'un essaim de 12 robots et permet d'observer comment les liaisons se répartissent et se renforcent en fonction des distances. Cette approche peut être étendue à des systèmes de plus grande envergure ou intégrée dans des outils de supervision pour des déploiements réels de systèmes multi-agents ou robotiques distribués.

5.10. Conclusion et Ouverture

5.10.1. Bilan du Chapitre

- Récapituler la mise en place d'une **architecture** SCN :
 1. Structures de données (matrices, adjacency),
 2. Modules (synergie, mise à jour, inhibition),
 3. Interfaces (ajout d'entités, extraction de clusters),
 4. Distribution, sécurité, etc.

5.10.2. Lien vers Chapitres 6, 7, 8

- Chap. 6 : Apprentissage Synergique Multi-Échelle (exploitation de cette architecture pour gérer différents niveaux).
- Chap. 7 : Algorithmes d'Optimisation et Méthodes d'Adaptation Dynamique (performances, recuit plus avancé).
- Chap. 8 : DSL Multimodal (comment la structure SCN se déploie à grande échelle avec divers flux).

5.10.3. Conclusion Générale

- Souligner que ce **Chapitre 5** fait le lien entre la **dynamique** (Ch. 4) et les **approches plus spécialisées** (Ch. 6, 7, 8) :
 - On dispose désormais de la $\textit{colonne vertébrale}$ pour implanter un SCN, en modulaire ou en distribué, assurant robustesse et évolutivité.

5.10. Conclusion et Ouverture

Après avoir exploré la **dynamique** d'un SCN (chapitre 4) et esquissé divers **exemples** de mise en œuvre dans des cas multimodaux, hybrides et distribués, ce chapitre 5 s'est consacré à la **structure générale** d'un SCN sur le plan « architecture et ingénierie ». Nous avons vu comment l'on passe du schéma mathématique $\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i,j), \dots)$ à une organisation modulaire — avec des **modules** dédiés (calcul de la synergie, mise à jour, inhibition, etc.) —, des interfaces temps réel (ajout/suppression d'entités, extraction de clusters), et une approche distribuée (sharding, synchronisation, robustesse) garantissant la **scalabilité** et la **fiabilité** du système.

5.10.1. Bilan du Chapitre

Le présent **chapitre 5** a exposé les fondements **techniques** et **architecturaux** nécessaires à l'**implémentation** d'un **Synergistic Connection Network (SCN)** à partir des principes **mathématiques** vus antérieurement. L'ensemble des **équations** formant la dynamique de mise à jour $\omega(t+1) = F(\omega(t))$ a ainsi trouvé sa transposition dans des structures de données appropriées, des “modules” logiques, et des mécanismes de distribution et de sécurité. Il convient de souligner plusieurs avancées majeures qui ont jalonné ce chapitre :

Dans un premier temps, la notion de **structure de données** a été clarifiée. Suivant la **taille** du SCN et la **parsimonie** attendue (c'est-à-dire la proportion de liens réellement actifs), on opte pour une **matrice dense** $\{\omega_{i,j}\}$ ou une **représentation sparsée** : listes d'adjacence, dictionnaires (hashmaps), etc. D'un point de vue **mathématique**, cela revient à discriminer quelles paires (i,j) ont un $\omega_{i,j}$ pertinent, et à gérer la “portion active” de ω . Dans un SCN de grande dimension, les mécanismes de “top-k” ou de filtrage ϵ -radius (cf. Section 5.4.3) encouragent la parcimonie, rendant l'implémentation scalable.

Ensuite, le concept de **module** s'est imposé comme un moyen de séparer la **logique** du calcul de la **synergie** $S(i,j)$ (Section 5.4) de la logique du **noyau** (ou *core*) qui pilote la **mise à jour** de ω . On a souligné que le **Module Synergie** pouvait décliner plusieurs formules : par exemple, une fonction S_{sub} pour des entités sub-symboliques (vecteurs), une fonction S_{sym} pour des entités symboliques, etc. Parallèlement, le **Module Mise à Jour** (Section 5.5) orchestre la **règle DSL** (additive, multiplicative, etc.), ainsi que divers compléments (inhibition compétitive, saturation) en s'appuyant sur les équations décrites au chapitre 4. L'organisation en “modules” favorise la **clarté** et la **modularité** : on introduit ou modifie une règle d'inhibition sans toucher au code de similarité, et inversement.

La question des **interfaces** (Section 5.6) a ensuite été abordée. On s'est penché sur les moyens d'**ajouter** ou de **retirer** des entités \mathcal{E}_i en temps réel, provoquant une reconfiguration de la matrice ω . Cette flexibilité s'avère cruciale dans des environnements où l'ensemble des entités n'est pas figé (flux de données, scénarios robotiques, bases évolutives). De plus, il a été question des méthodes pour **extraire** les clusters ou les liens forts, afin d'alimenter des modules externes de visualisation ou de classification. Ces interfaces s'intègrent parfaitement à la logique d'**auto-organisation** du SCN, tout en permettant un usage “ouvert” vers d'autres systèmes.

Enfin, le chapitre a insisté sur la **distribution** (Section 5.7) et la **sécurité** (Section 5.8). La distribution, qu'elle se concrétise par des **sous-SCN** interconnectés ou par un paradigme de **microservices**, vise à gérer des **SCN** de très grande échelle, tout en évitant un point central unique. Sur le plan **mathématique**, cela correspond à décomposer la matrice ω en shards, et à déployer un protocole de **synchronisation** (asynchrone ou synchrone) pour conserver la **cohérence** du réseau. D'un point de vue **ingénierie**, des solutions de **sharding**, d'équilibrage de charge, et de **méta-SCN** (Section 5.7.2) concrétisent ce modèle. Concernant la **sécurité**, on a introduit des méthodes de **logs**, de **watchers**, et de **checks d'anomalies** (cf. 5.8.1, 5.8.3), ainsi que des techniques de **basculement automatique** et de **redondance** (5.8.2) pour qu'un SCN distribué ne cède pas face à une attaque ou à une panne matérielle. Le chapitre a ainsi présenté la “colonne vertébrale” d'une architecture résiliente, apte à faire tourner un SCN en conditions réelles ou hostiles.

En synthèse, ce **chapitre 5** constitue un “canevas” technique pour **réaliser** un SCN à partir d'un socle **mathématique** (mise à jour $\omega(t+1) = F(\omega(t))$, synergie S). Son propos s'est montré large : il a exposé les structures de données, l'organisation modulaire (Module Synergie, Module Mise à Jour), l'ouverture d'interfaces (ajout/retrait d'entités, extraction de clusters), la gestion distribuée et sécurisée. Ce sont autant d'éléments indispensables pour que la **dynamique d'auto-organisation** décrite dans les chapitres précédents devienne un **système** informatique fiable et évolutif, qu'il s'agisse d'un **SCN** multimodal, robotique ou d'un usage encore différent.

5.10.2. Lien vers Chapitres 6, 7, 8

Au terme de ce **chapitre 5**, où l’on a soigneusement défini la **colonne vertébrale** d’un SCN (Synergistic Connection Network) du point de vue **architectural** et **implémentatif**, il est naturel de se pencher sur la manière dont cette infrastructure va s’articuler avec les développements plus **avancés** des chapitres ultérieurs. Les **modules**, le **partage** ou la **distribution**, la **sécurité**, et l’ensemble des mécanismes présentés servent de **fondation** à des thématiques plus spécialisées, telles que la **multi-échelle** (Chapitre 6), les **optimisations** et **adaptations** (Chapitre 7), ou encore l’**expansion** du DSL à de vastes environnements **multimodaux** (Chapitre 8).

Chap. 6 : Apprentissage Synergique Multi-Échelle

Le **Chapitre 6** aborde la gestion d’un SCN fonctionnant à **plusieurs niveaux** ou **plusieurs échelles** d’entités et de synergies. L’idée consiste à permettre, par exemple, qu’un SCN se décompose simultanément en **micro-clusters** très fins et en **macro-structures** plus globales, évoquant différents degrés de granularité ou d’abstraction. D’un point de vue **mathématique**, on retrouve la notion de répéter la logique d’**auto-organisation** (règle DSL, inhibition, saturation) à plusieurs *couches*, ou de passer par un “méta-niveau” supervisant l’assemblage des clusters locaux. D’un point de vue **architecture** (développée dans ce chapitre 5), on mobilise la séparation en **modules** (synergie, mise à jour) et la **distribution** (Section 5.7), pour distinguer différentes **couches** ou **blocs**. Le **multi-échelle** enrichit la capacité d’**auto-organisation** : le SCN n’agit plus seulement à un plan unique, mais adopte un “zoom” local et global, ce qui implique une orchestration subtile. Le **Chapitre 6** prolonge donc l’**infrastructure** décrite ici, en l’appliquant à une situation où plusieurs **niveaux** de représentation s’interconnectent.

Chap. 7 : Algorithmes d’Optimisation et Méthodes d’Adaptation Dynamique

Le **Chapitre 7** explore des **mécanismes** plus avancés pour améliorer la convergence, la stabilité ou le temps de réponse d’un SCN. S’il est vrai que les **règles** DSL de base (additive, multiplicative, etc.) constituent un socle déjà efficace, on peut vouloir intégrer des techniques plus élaborées (recuit simulé raffiné, heuristiques inspirées de la physique statistique, descentes adaptatives, etc.) pour atteindre un **minimum** d’énergie plus global ou pour éviter certaines bifurcations indésirables. Or, l’**architecture** (Chap. 5) décrivant la séparation “**Module** Mise à Jour” (Section 5.5) se prête idéalement à ces substitutions ou compléments : on peut injecter un algorithme d’optimisation “avancé” dans la boucle de mise à jour, sans toucher à la structure sous-jacente (modules de synergie, distribution). Le **Chapitre 7** mettra donc l’accent sur la notion de **performance** et de **résilience** algorithmique d’un SCN, tirant profit de l’organisation modulaire et des protocoles distribués détaillés ici.

Chap. 8 : DSL Multimodal

Si l’on veut aller plus loin dans la **multimodalité** — c’est-à-dire manipuler, à *grande échelle*, plusieurs types de données (images, texte, audio, signaux divers) —, le **Chapitre 8** se penchera sur les méthodes requises pour gérer un SCN imposant, agrégeant des entités de natures variées. Les concepts d’architecture (Section 5.9) et de distribution (Section 5.7) prennent une importance cruciale : lorsque le nombre d’entités et de flux multimodaux grandit, il devient indispensable de recourir aux solutions de **sharding** ($O(n^2)$ liens), de **synchronisation** inter-blocs, et de **sécurisation** (Section 5.8) présentées en détail dans ce chapitre. Le **Chapitre 8** focalisera sur la **logique** DSL appliquée à des sources massives (vision, langage, son), mobilisant la modularité (ModuleSynergy pour chaque modalité) et l’éventuel recuit (5.5.4) pour une **auto-organisation** multimodale puissante.

Conclusion sur le Lien avec Chapitres 6, 7, 8

Le **Chapitre 5** a défini l’**ossature** : structures de données, organisation en “modules” (synergie, mise à jour), interfaces pour la distribution et la sécurité. Ce **canevas** permet de **concrétiser** la dynamique $\omega(t) \rightarrow \omega(t + 1)$ dans des cas d’usages complexes. Les **chapitres 6, 7, 8** profitent directement de cette infrastructure :

36. Le **Chap. 6** poussera l’idée de **multi-échelle**, montrant comment la modularité et les protocoles distribués soutiennent une auto-organisation hiérarchique.
37. Le **Chap. 7** traitera des **algorithmes d’optimisation** avancés, trouvant dans la structure modulaire (Section 5.5) un champ favorable pour injecter de nouvelles heuristiques.

38. Le **Chap. 8** explorera la **multimodalité** à vaste échelle, s'appuyant sur les outils de distribution (5.7) et de robustesse (5.8) pour orchestrer un SCN en présence de multiples flux (images, textes, signaux audio, etc.).

Au total, ce passage vers les **chapitres 6, 7, 8** montre comment l'**architecture** établie dans ce chapitre 5 sert de **fondation** pratique et conceptuelle, permettant d'**élargir** la portée du DSL vers des dispositifs multi-niveaux, des algorithmes plus fins, et des environnements massivement multimédias ou hétérogènes.

5.10.3. Conclusion Générale

Le **chapitre 5** établit la véritable **charnière** entre la formulation mathématique de la **dynamique** d'un **SCN** et les usages avancés (multi-échelle, optimisations, multimodalité) que l'on retrouvera dans les prochains chapitres. Jusque-là, le **chapitre 4** avait détaillé la mise à jour $\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i,j))$ et ses variations (additive, multiplicative, etc.). Toutefois, ces principes demeurent purement **théoriques** tant qu'ils ne sont pas insérés dans une **architecture** capable d'héberger la matrice ω , de gérer le calcul du score de synergie, et d'assurer la gestion distribuée ou la robustesse nécessaire à un usage concret.

Ce **chapitre 5** a précisément comblé cet écart, en introduisant un **canevas** technique, du niveau des **structures de données** jusqu'aux mécanismes de **sécurité**.

Dans un premier mouvement, on a rappelé que la **taille** du SCN (et la nature plus ou moins parcimonieuse de ses liaisons) détermine le choix des **structures** : matrice dense $\{\omega_{i,j}\}$ ou schémas plus **spars** (listes d'adjacence, hashmaps, etc.). D'un point de vue **mathématique**, cela revient à définir plus précisément le "domaine" de ω : quelles paires (i,j) se retrouvent effectives dans la boucle, et comment on indexe ou on stocke la partie active.

On a ensuite consolidé l'idée de séparer un **Module de Synergie** (Section 5.4), qui encapsule le calcul $S(i,j)$ — éventuellement différent selon les modalités sub-symboliques ou symboliques —, et un **Module Mise à Jour** (Section 5.5), chargé d'appliquer la règle DSL, d'incorporer l'inhibition, la saturation ou le recuit. Cette **modularité** se révèle décisive pour l'extensibilité : on peut changer l'algorithme de synergie ou expérimenter une nouvelle règle d'évolution $\omega_{i,j}(t+1)$ sans déstabiliser tout le code.

Le chapitre a aussi présenté l'existence d'**interfaces** (5.6) permettant d'**ajouter** ou de **supprimer** des entités \mathcal{E}_i en temps réel, de **consulter** des clusters ou des liens dominants, etc. Sur le plan **ingénierie**, cela confère un **visage** plus "ouvert" au SCN : on peut l'intégrer à d'autres modules, l'utiliser pour la visualisation ou l'analyse d'un graphe, ou procéder à des modifications dynamiques (scénario en flux continu).

Puis on a traité la **distribution** (Section 5.7), moment-clé pour gérer des SCN de forte dimension ou distribués géographiquement. Les principes de **sharding** (couper la matrice ω en fragments), de **communication inter-blocs** (messages synchrones ou asynchrones), et de **synchronisation** (invariants, bornes) assurent une **scalabilité** horizontale et une résilience face au volume massif de liaisons. D'un point de vue **mathématique**, on retrouve un ensemble d'opérateurs locaux couplés, où la mise à jour $\omega_{i,j}(t+1) = F(\omega_{i,j}(t), S(i,j))$ doit être cohérente même avec les retards et la répartition topologique.

Enfin, les **aspects de sécurité** (Section 5.8) et de **robustesse** donnent les briques manquantes pour qu'un SCN opère dans des contextes éventuellement hostiles ou soumis à défaillance : détection d'anomalies ($\omega_{i,j}$ qui croît anormalement), logs, watchers, réactivation en cas de panne, duplication (failover, redondance). Le SCN n'est plus simplement un algorithme isolé : il devient un **système** complet, capable de résister aux sabotages et aux coupures, tout en maintenant son auto-organisation.

Lien direct avec la Dynamique (Chap. 4).

L'essentiel des équations de mise à jour, comme $\omega_{i,j}(t+1) = \omega_{i,j}(t) + \eta [S(i,j) - \tau \omega_{i,j}(t)]$, prennent **forme concrète** lorsqu'on sait où stocker ω , comment appeler la fonction $S(i,j)$, et comment manipuler ω en flux continu (ajout/suppression d'entités). Les sections 5.4 (Module Synergie) et 5.5 (Module Mise à Jour) traduisent directement les formulations mathématiques (Chap. 4) en entités logicielles, paramétrables et agencées dans un pipeline.

Porte d'accès aux Chapitres 6, 7 et 8.

Les chapitres qui suivent s'adosent à l'infrastructure présentée ici :

- Le **Chap. 6** traite du “multi-échelle”, c'est-à-dire de SCN capables de se structurer selon plusieurs niveaux (micro-clusters, macro-structures). Il mobilise la répartition (Section 5.7) et la modularité (Sections 5.4, 5.5) pour illustrer l'auto-organisation dans des contextes hiérarchiques ou à échelles imbriquées.
- Le **Chap. 7** introduit des **algorithmes d'optimisation** et des heuristiques d'amélioration (recuit plus poussé, stratégies de minimisation d'énergie, etc.). On y voit comment la “colonne vertébrale” du SCN, déjà paramétrée (Section 5.5, 5.5.4), se prête à l'insertion de techniques plus avancées, tant dans un cadre local que distribué.
- Le **Chap. 8** approfondit la notion de **DSL Multimodal**, où l'on manipule potentiellement des flux massifs (vision, audio, langage). Les principes d'architecture (modules de synergie spécialisés, distribution) et de sécurité (Section 5.8) se révèlent alors indispensables pour supporter la “montée en charge” et la gestion de multiples types d'entités.

Robustesse et Évolutivité.

On retiendra que le **chapitre 5** propose une **colonne vertébrale** : choix de structures (dense, sparse, NoSQL), organisation modulaire (Synergie, Mise à Jour), distribution à grande échelle (sharding, synchronisation), mécanismes de logs, watchers, reprise sur panne. L'ensemble confère à la **dynamique DSL** (Chap. 4) la solidité nécessaire pour s'exécuter dans des conditions **réalistes** et **évolutives**. On peut donc envisager un SCN opéré en cluster, mis à jour en flux, protégé contre des anomalies ou des sabotages, et capable de réassigner ses entités au gré de l'évolution.

Conclusion générale du Chapitre 5 :

On dispose désormais des **bases** d'ingénierie et d'architecture pour qu'un SCN devienne un système complet, joignant l'**auto-organisation** interne (mise à jour ω) à une gestion modulaire (synergie, distribution, sécurité) et à la **flexibilité** (ajout/suppression d'entités, extraction de liens). Les **chapitres suivants** (6, 7, 8) puiseront dans cette infrastructure pour développer :

- La **multi-échelle** (Chap. 6),
- Les **algorithmes d'optimisation** (Chap. 7),
- Les **extensions multimodales** plus complexes (Chap. 8).

Ce faisant, la **puissance** du Deep Synergy Learning se déploiera pleinement, soutenue par un canevas technique et algorithmique solide.