Rapport de Stage Ingénieur :

Réseaux neuronaux bioinspirés

Du 3 avril 2017 au 18 aout 2017

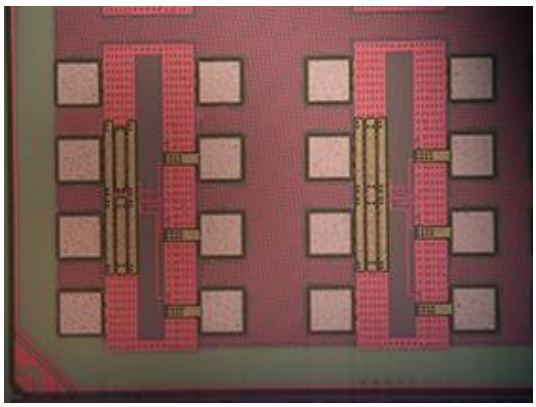
Maître de stage : Alain Cappy

Lieu du stage : IRCICA – Villeneuve d'Ascq 59650

Tuteur de stage : Olivier Roux

Option disciplinaire : BIOSTIC – SANTE

Ozier-Lafontaine Anthony



Photographie du neurone artificiel

IRCICA Research Institute

CENTRALE NANTES

Table des matières

Remerciements

Je remercie tous les membres de l'IRCICA pour leur sympathie et leur altruisme et plus particulièrement Alain Cappy qui a rendu cette expérience possible.

I. Introduction

1. Cadre de travail

a. L'IRCICA

J'ai effectué mon stage sur les réseaux de neurones artificiels à l'Institut de Recherche en Composants logiciels et matériels pour l'Information et la Communication Avancée (IRCICA). L'IRCICA est une Unité de Service et de Recherche associant le CNRS et l'Université Lille1, sciences et technologies. Cet organisme fonctionne comme un « hôtel à projet », visant à initier des projets de recherche interdisciplinaires de grande envergure. Ces projets sont incubés très en amont et sont risqués mais ont des retombées potentielles importantes. Environ 120 enseignants, chercheurs, étudiants, ingénieurs et techniciens provenant de quatre laboratoires partenaires – CRIStAL, IEMN, Phlam et L2EP – y sont rassemblés. Les différents projets en cours traitent des fibres optiques, des interfaces tactiles haptiques et des architectures bioinspirées de traitement de l'information.

Une telle structure présente un avantage financier non négligeable. Les puces intégrant les neurones artificiels sont étudiées dans les laboratoires de nanotechnologie de l'Institut d'Électronique, de Microélectronique et de Nanoélectronique, l'IEMN. Je ne les ai jamais manipulées et je n'ai pu les voir qu'en tant que visiteur. L'IRCICA utilise des supercalculateurs pour certaines simulations, mais celles que j'ai effectuées ne nécessitaient pas une telle puissance de calcul.

J'ai rejoint les 25 membres du projet Bioinspiré qui s'articule autour de trois axes :

- L'équipe d'informaticiens développe un logiciel de simulation de réseaux de neurones à grande échelle.
- L'équipe FOX étudie numériquement le comportement des réseaux de neurones et la vision : apprentissage non-supervisé, réseaux à « spikes » et systèmes à couches et non-récurrents.
- L'équipe Hardware développe des neurones artificiels inspirés de modèles biologiques. J'ai effectué mon stage avec cette équipe, constituée de quatre membres permanents et de quelques membres temporaires.

b. L'équipe Hardware

J'ai exclusivement travaillé au sein de l'équipe Hardware. J'ai échangé avec les autres équipes du projet Bioinspiré lors des réunions d'avancement et parfois lorsqu'il me manquait des compétences techniques, notamment en informatique.

i. Objectifs et ambitions de l'équipe Hardware

L'objectif de l'équipe Hardware est de créer des neurones artificiels puis de les composer pour créer des systèmes à faible consommation d'énergie et possédant des capacités cognitives. Ces systèmes pourraient permettre de compenser les manques dus à certaines maladies s'ils peuvent être intégrés à des réseaux de neurones biologiques.

<u>Aspect énergétique</u>

Nous sommes dans l'ère du numérique, les ordinateurs, aussi appelés machines de Von Neumann sont de plus en plus nombreux et consomment entre 50 et 100 Watts heure. Le cerveau humain quant à lui consomme moins de 10 Watts heure, cette prouesse s'explique par la capacité des réseaux de neurones à

effectuer plusieurs tâches en parallèle. Une nouvelle génération de machines inspirées du vivant permettrait de répondre aux besoins de l'industrie du numérique sans impacter l'environnement par sa consommation énergétique.

Il serait même possible d'aller plus loin : un potentiel d'action ou spike de neurone biologique (cf. II.2.d.) mets en jeu le déplacement d'environ 100 millions d'ions, tandis que certains composants électroniques sont capables d'échanger les électrons un par un. L'économie d'énergie d'une telle technologie dans les neurones artificiels serait alors immense. Cette perspective est encore hypothétique mais le neurone artificiel consomme tout de même 100fois moins d'énergie qu'un neurone biologique.

Aspect cognitif

Avec une architecture adéquate, un réseau de neurones développe des capacités cognitives et effectue des tâches simples pour les êtres vivants dotés d'un cerveau. Ces réseaux peuvent par exemple raisonner et prendre des décisions, apprendre ou encore voir et reconnaître. Aujourd'hui, nous obtenons ces capacités cognitives chez des robots en simulant des réseaux de neurones dans leurs programmes. Mais le coût énergétique est important car les neurones sont simulés sur des machines de Von Neumann et sont donc incapables de fonctionner en parallèle.

L'équipe Hardware a développé deux versions du neurone artificiel. Ces deux versions fonctionnent à des vitesses complètement différentes. D'un côté le neurone Biologique fonctionne à la même vitesse que les neurones du vivant et il porte les espoirs des applications médicales du neurone artificiel. De l'autre, le neurone Fast est mille fois plus rapide qu'un neurone vivant, et serait utile pour des applications plus industrielles et robotiques.

Limites

Les applications liées aux neurones artificiels sont certes optimistes, mais les contraintes techniques actuelles imposent tout de même certaines limites en termes de superficie, de connectivité et de complexité.

Les neurones artificiels sont implémentés sur des circuits en deux dimensions et leur superficie environne les 200 micromètres carrés tandis que le neurone biologique est quasiment en une dimension et mesure 120micromètres de longueur.

L'architecture tridimensionnelle des neurones biologiques leurs permet de posséder en moyenne 7000 connexions synaptiques, tandis que l'architecture bidimensionnelle des circuits limite les neurones artificiels à quelques dizaines de connexions.

Enfin, il existe différentes échelles d'étude des neurones biologiques, depuis celle du gène à celle des populations. La fabrication d'un neurone artificiel nécessite un cadre restreint. Ainsi il n'existe aujourd'hui qu'un seul type de neurone artificiel, contre quelques dizaines de neurones biologiques. De plus le modèle du neurone artificiel ne rend pas compte de la complexité liée aux neurotransmetteurs, aux canaux ioniques et à la forme unique de chaque neurone.

ii. Etat actuel du projet et axes de réflexion

Les membres de l'équipe Hardware sont majoritairement spécialisés dans l'électronique et la nanoélectronique. Leurs efforts conjugués depuis l'initiation du projet il y a deux ans ont aboutis à la

création d'un circuit électronique pensé pour mimer le comportement électrique membranaire d'un neurone : le neurone artificiel.

La fabrication du neurone artificiel a montré des résultats satisfaisants. L'attention de toute l'équipe se porte maintenant sur sa caractérisation, son amélioration et la conception de réseaux composés de neurones artificiels. Les aspects suivants sont plus particulièrement étudiés :

- Le bruit est un élément indispensable au bon fonctionnement du cerveau, ses mécanismes stochastiques sont donc étudiés pour les implémenter et rendre le neurone artificiel plus réaliste.

- Il est nécessaire de caractériser les composants électroniques du neurone utilisés dans des conditions non prévues par leurs concepteurs.

- Les architectures de réseaux imaginées en 3D doivent être transposées en 2D par un ingénieur pour pouvoir être implémentées sur des puces électroniques. Il conçoit les réseaux en optimisant l'espace tout en permettant leur étude à l'aide de ports de branchements de dispositifs de mesures.

- Des photodiodes compatibles sont développées pour jouer le rôle de photorécepteurs dans le projet de rétine artificielle.

- Le travail administratif concernant la rédaction et la relecture des publications, le dépôt des brevets, le contact des partenaires et les réponses aux médias scientifiques intéressés occupe une partie du temps des têtes du projet.

Enfin, quelques mois séparent la fabrication des circuits de neurones artificiels, qu'on appelle des puces. Le rythme est donné par le laboratoire partenaire qui nous offre généreusement l'espace disponible dans les circuits qu'il fabrique. Ce laboratoire fait fondre ses puces en Asie et le millimètre carré de circuit nanométrique coûte 3500€, les économies réalisées par l'IRCICA grâce à cet accord sont immenses, mais en contrepartie, l'IRCICA doit s'adapter à l'emploi du temps de ce laboratoire. Afin d'optimiser, l'équipe essaie de profiter de chaque commande, même si l'étude de la commande précédente n'est pas terminée.

iii. Contexte de mon arrivée

Chaque lot de puces correspond à une étape du projet. Les deux lots de puces « chip1 » et « chip2 » étaient déjà fabriqués à mon arrivée et les plans de conceptions de la « chip 3 » étaient sur le point d'être envoyés pour la prochaine commande. Nous verrons par la suite que cette dernière commande a été reportée et que cela a eu un impact sur le déroulement de mon stage.

« Chip 1 » : Fabriquée et caractérisée

La « chip 1 » fut la première implémentation du neurone artificiel et a montré que les circuits Biologique et Fast fonctionnaient.

« Chip 2 » : Fabriquée et en cours de caractérisation

La « chip 2 » intègre un circuit de type « neurone – synapse – neurone » pour tester les connexions synaptiques. Elle permet aussi d'étudier séparément les composants du neurone artificiel afin de mieux comprendre son comportement.

« Chip 3 » : En cours de conception

La « chip 3 » intègrera le premier réseau de neurones artificiels. Ce réseau de type reservoir computing permettra de tester le modèle adopté pour les synapses plastiques artificielles (cf. II.3.c.). Son comportement sera comparé avec celui de ses simulations et permettra donc de juger la qualité des outils de simulation à disposition. La puce intègre aussi des prototypes de la photodiode en cours de développement. Elle sera fabriquée pendant le second semestre de 2017, et la date limite d'envoi des plans a été repoussée de juin 2017 à aout 2017, ainsi j'ai pu la simuler et l'étudier à la fin de mon stage.

2. Objectifs du stage

La durée du stage m'a permis de travailler en parallèle sur différentes problématiques. D'un côté j'ai développé un outil de simulation adapté aux réseaux de neurones artificiels, et en parallèle, j'ai utilisé cet outil afin de modéliser et étudier les réseaux à venir.

J'ai placé en annexe de ce rapport le document destiné à l'IRCICA dans lequel je présente précisément le logiciel sur lequel j'ai travaillé pour faciliter sa prise en main à mes éventuels successeurs. J'y présente aussi les réseaux de neurones que j'ai modélisés. Ce document constitue le rendu que j'ai présenté à mon responsable de stage et à pour objectif d'être le plus exhaustif possible. Les aspects les plus techniques de mon travail y sont présentés et les deux documents sont complémentaires.

a. Développement d'un outil de modélisation et de simulation

Les premières puces intégraient au maximum deux neurones connectés, des logiciels de simulation de circuits électroniques comme *Spice* étaient alors utilisés. Ils simulaient le comportement de chaque composant des neurones artificiels. Cependant, un réseau tel que le reservoir computing de la « chip 3 » contient une trentaine de neurones. Ces logiciels nécessitent plusieurs heures de calcul pour quelques dizaines de millisecondes simulées et les erreurs cumulées rendent compte d'un comportement non réaliste.

Les réseaux ne vont cesser d'augmenter en taille et cela impose une nouvelle méthode de simulation. Les neurones artificiels sont inspirés du modèle de Morris-Lecar [9], leur fonctionnement étant prouvé, un outil de simulation de réseaux de neurones intégrant ce modèle semble bien plus adapté et les temps de calcul redeviendraient raisonnables. Mon travail consiste donc à tester et améliorer un tel outil pour le rendre compatible avec le neurone artificiel. Cet outil sera mis à la disposition des membres de l'équipe Hardware et de mes successeurs, qui pourront reprendre sans difficultés le travail où je l'aurai laissé.

Le logiciel à partir duquel j'ai travaillé se nomme *Simbrain*, c'est un logiciel de simulation de réseaux de neurones proposant le modèle de Morris-Lecar.

b. Modélisation : rétine artificielle et reservoir computing

J'ai principalement étudié deux réseaux de neurones : le reservoir computing et la rétine artificielle. J'ai aussi développé des réseaux de test ou des réseaux de briques élémentaires en parallèle.

Le reservoir computing

On appelle reservoir computing un type de réseaux de neurones pensé pour l'apprentissage. Il tire son nom du réservoir de neurones placés entre l'entrée et la sortie du réseau. Son fonctionnement sera décrit dans la partie du développement correspondante.

L'équipe Hardware s'est lancée dans la fabrication de ce réseau relativement simple sans avoir pu étudier le paramétrage idéal, faute d'outil de simulation adapté. Il est donc prévu que ce réseau n'apporte pas forcément de résultats intéressants. Cela s'explique par la contrainte de rythme imposée par le laboratoire partenaire.

Les délais dans la fabrication de cette puce ont été les bienvenus puisque les quelques mois supplémentaires ont permis de simuler le réseau et d'éventuellement en améliorer les paramètres à l'aide de l'outil que j'avais développé. Finalement, ce réseau m'a beaucoup aidé à améliorer mon outil mais je n'ai pas eu assez de temps pour rechercher le paramétrage idéal à adopter.

La rétine artificielle

Objet d'étude initial de mon stage, la rétine artificielle est le véritable projet de réseau basé sur les neurones artificiels. Composée d'une couche de photodiodes puis de plusieurs couches de neurones, elle reproduit les performances de la rétine jusqu'à la première couche du cortex visuel. Elle devra donc reconnaitre des objets et les voir se déplacer.

Elle porte l'espoir d'une nouvelle génération de caméras intelligentes, et elle proposera une solution pour les personnes dont la vision est réduite. La particularité de la rétine artificielle par rapport à d'autres technologies similaires est qu'elle traite l'information de manière analogique et en temps réel, grâce au parallélisme des processus.

Le but de mon stage était d'entamer la modélisation de cette rétine artificielle à partir des idées de mon maître de stage Alain Cappy et de ma compréhension du fonctionnement de l'œil. La compréhension de ce réseau au stade embryonnaire à travers sa simulation devait ensuite me conduire à faire les bons choix, à l'améliorer et l'optimiser.

c. Démarche

L'outil de modélisation et de simulation a été développé en parallèle de la modélisation et de la simulation du reservoir computing et de la rétine artificielle. Ainsi, j'identifiais les modifications nécessaires de *Simbrain* tout en progressant sur la modélisation des réseaux. L'ensemble du processus nécessite des compétences en programmation, une bonne compréhension des mécanismes de la biologie et des contraintes techniques liées à la technologie du neurone artificiel.

Simbrain : outil de modélisation de réseaux de neurones artificiels

Le logiciel *Simbrain* constitue le socle de tout mon travail de stage. Il modélise les neurones à l'échelle cellulaire et rend compte de leur activité électrique membranaire moyenne. Chaque type de neurone est basé sur des équations mathématiques, celles-ci sont inspirées d'observations biologique en ce qui concerne le neurone de Morris-Lecar. J'ai été amené à modifier le logiciel car certains aspects de sa logique n'étaient pas compatibles avec le fonctionnement du neurone artificiel.

Modélisation de la rétine artificielle

Il existe deux façons d'aborder la modélisation d'un réseau de neurones inspiré du vivant tel que la rétine. Dans une optique de compréhension, on s'attache à reproduire fidèlement le réseau puis à vérifier si le comportement obtenu correspond. Et dans une optique technique, on cherche plutôt à discerner l'essentiel du réseau pour obtenir un comportement similaire sur un réseau plus simple. Cette seconde démarche est celle que j'ai suivie.

L'ensemble de la réflexion autour de la rétine artificielle s'inspire de connaissances en biologie sur les neurones et sur le fonctionnement de l'œil, mais aussi sur des travaux de vision et apprentissage d'intelligences artificielles et de traitement de l'information. Les contraintes techniques du neurone artificiel imposent au modèle d'optimiser le nombre de neurones utilisés. Des simulations à chaque étape permettent ensuite d'améliorer le réseau.

Modélisation du réseau de reservoir computing

Le réseau ayant déjà été conçu entièrement, je l'ai simplement reproduit dans le logiciel *Simbrain*. Dans un premier temps, ce réseau au fonctionnement bien plus simple que la rétine artificielle a révélé certains défauts subtils du logiciel *Simbrain*. Les simulations sur *Simbrain* ont apportées plusieurs éléments de compréhension du réseau là où les simulations électroniques avaient atteint leurs limites.

d. Conditions de travail

J'ai travaillé seul sur le logiciel *Simbrain* et sur la modélisation de la rétine artificielle. Monsieur Cappy, mon maître de stage, suivait quotidiennement mon travail et m'aiguillait par sa connaissance globale du projet, sa maîtrise du neurone artificielle, et sa compréhension des mécanismes de la biologie. Je n'ai jamais manipulé de neurone artificiel réel car ses dimensions nanométriques nécessitent des compétences que je ne possède pas.

J'ai été agréablement surpris par cette grande autonomie qui m'a permis de m'investir totalement dans le projet et de choisir mon rythme. J'ai ainsi pris certaines initiatives qui se sont avérées utiles, par exemple le fait d'apprendre les bases de la programmation JAVA de façon autonome a positivement impacté ma productivité pendant le stage.

L'IRCICA est un lieu convivial et tous les membres de l'équipe « bio inspirée » ont su se rendre disponibles et m'aider chaque fois que j'avais besoin de leurs compétences.

II. Développement d'un outil de modélisation et de simulation : Simbrain_IRCICA

1. Première approche de *Simbrain*

a. Présentation du logiciel

Le logiciel *Simbrain* dont le logo est ci-dessous est un logiciel en accès libre développé par Jeff Yoshimi, un enseignant chercheur et philosophe travaillant sur les sciences cognitives à l'université de Merced en Californie. Le logiciel *Simbrain* a pour ambition de permettre la modélisation de tous les types de réseaux de neurones, pour des applications variées. Le modèle de neurones utilisé dans un réseau détermine son comportement. C'est pourquoi le logiciel propose plusieurs modèles différents.

Logo *Simbrain*

Simbrain a été choisi parmi tous les logiciels de simulations de réseaux de neurones car il remplit plusieurs conditions indispensables :

- Il est l'un des rares logiciels de ce type à intégrer les équations de Morris-Lecar, et leur paramétrage est entièrement accessible
- La grandeur simulée par le neurone artificiel, le potentiel de membrane, est la variable d'intérêt du logiciel, il en donne un suivi en temps réel. De plus, il effectue la même approximation qui consiste à ne pas considérer les neurones comme des objets tridimensionnels dont le potentiel de membrane est hétérogène.
- Le logiciel propose d'ajouter du bruit neuronal, un élément essentiel des réseaux de neurones vivants. Cela sera utile pour l'avenir du projet.
- Les réseaux simulés peuvent atteindre des tailles conséquentes.
- Sa documentation explicite facilite la prise en main et l'interface graphique présente certains avantages.

b. Prise en main

J'ai tout d'abord créé des modèles avec le neurone de Morris-Lecar « standard » dont le paramétrage est celui de la publication de l'invention du modèle. Les modélisations des premières couches de la rétine artificielle ont immédiatement donné des résultats satisfaisants et encourageants pour la suite.

Au début du projet, nous croyions naïvement que le logiciel pouvait être utilisé tel quel et mon stage se focaliserait sur la rétine artificielle. J'ai rapidement rencontré les limites de *Simbrain*. Notamment pour certains aspects de modélisation incompatibles avec le neurone artificiel ; mais aussi par rapport à l'interface graphique qui entravait ma progression par manque de généralisation. J'ai alors entrepris d'apprendre la programmation JAVA afin de modifier directement le logiciel d'une part, et de me libérer de l'interface graphique d'autre part, puisque j'ai commencé à coder les réseaux de neurones, pour un gain de temps non négligeable.

Simbrain reste malgré tout une base de travail solide où presque tous les éléments de réseaux développés à l'IRCICA sont implémentés et presque compatibles : les neurones, les connexions synaptiques – plastiques ou non – et même les photorécepteurs avec un peu d'astuce. Seul le produit de courants synaptiques, inventé pour la reconnaissance de mouvement, n'était pas implémenté.

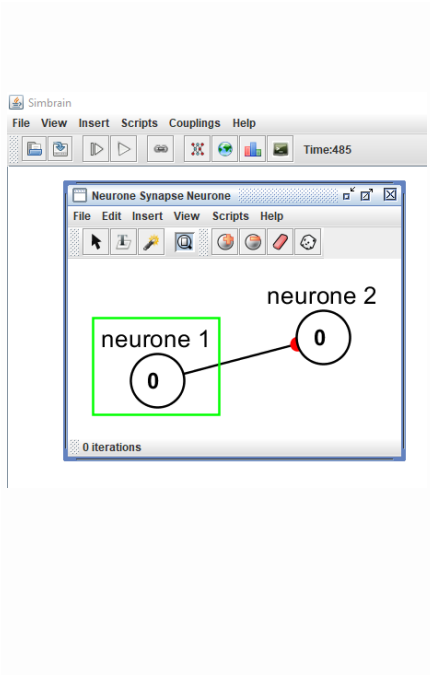
Durant ces quelques mois, il m'a fallu entrer au cœur du logiciel afin de m'imprégner de sa logique. Seule une compréhension complète de son programme me permettais d'identifier exactement les éléments à modifier et de ne pas apporter de solutions à des problèmes inexistants.

2. Modélisation du comportement d'un neurone

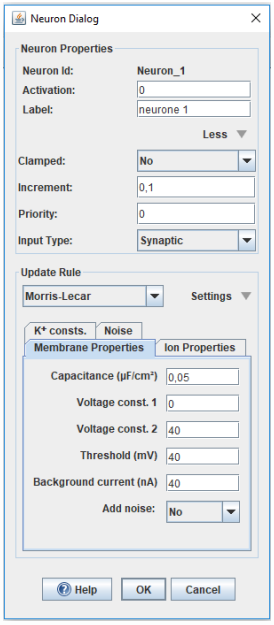
a. Les différents aspects du neurone dans Simbrain

i. Présentation générale du neurone Simbrain

L'interface graphique de *Simbrain,* même si elle est limitée, m'a tout de même permis de prendre en main le logiciel et d'en comprendre les bases. On y place des neurones dans un espace plan puis on les connecte sans difficultés. Les paramètres sont modifiables à travers des boites de dialogue. Les figures suivantes présentent l'interface et la boite de dialogue.



Interface Simbrain



Boite de dialogue neurone Simbrain

Le fait de coder les réseaux au lieu de passer par l'interface évite de modifier les paramètres un par un et offre donc un grand pouvoir de généralisation. De plus, cela permet de contourner certaines instabilités dues aux boites de dialogue. Il suffit d'un éditeur de texte pour écrire les scripts lus par *Simbrain.*

ii. « update rules » : équations de comportement du neurone

Le modèle mathématique qui régit le comportement de chaque neurone correspond à son « update rule ». Le logiciel va utiliser les équations de l' « update rule » pour mettre à jour l'état du neurone. *Simbrain* propose quinze modèles de neurones différents, chacun est issu de travaux scientifiques en biologie ou en informatique. Le modèle de Morris-Lecar fait partie des modèles dits à spikes, et j'ai aussi utilisé le modèle linéaire comme photorécepteurs ou sources de courants.

iii. « Activation »: variable fondamentale du neurone

La variable principale d'un neurone dans *Simbrain* est l' « activation ». Sa valeur est écrite sur le neurone dans l'interface graphique. Par exemple, elle vaut zéro pour les deux neurones de la figure précédente. L' « activation » d'un neurone Morris-Lecar correspond à son potentiel de membrane, alors que celle d'un neurone linéaire correspond au courant qu'il va envoyer dans ses synapses.

iv. « Input type » : type d'entrée reçue par le neurone

Avant de continuer, il faut savoir que les synapses sont orientées et ne délivrent le courant que dans un sens. Dans la biologie, la synapse est la connexion entre l'axone d'un neurone et la dendrite d'un autre, le courant ne peut aller que de l'axone vers la dendrite. Le logiciel *Simbrain* considère les neurones comme des éléments ponctuels, les axones et dendrites n'y existent donc pas. Dans la suite on appelle préneurone le neurone qui envoie un signal électrique et postneurone celui qui le reçoit.

L' « input type » détermine quel type de courant un postneurone va recevoir de ses synapses, il peut être « WEIGHTED » ou « SYNAPTIC ».

« SYNAPTIC »

Le cas « SYNAPTIC » correspond à la biologie car tant qu'il n'y a pas de potentiels d'action, les neurones sont complètement isolés électriquement. Nous l'utilisons dans la plupart des applications du neurone artificiel. Dans ce cas, le postneurone ne reçoit un courant synaptique que si le préneurone spike – atteint un pic de potentiel. Ce courant sera présenté explicitement dans la partie « spike responders ».

Le courant d'excitation I_{ex} du postneurone i est alors la somme des produits des poids synaptiques w_{ij} des neurones j vers i par les courants d'excitation générés par les potentiels d'action des préneurones j . Soit:

$$I_{ex}(i) = \sum_j w_{ij} * i_{SP}(j) \quad (1)$$

"WEIGHTED »

Dans le cas « WEIGHTED », le postneurone va simplement recevoir un courant d'excitation égal à la somme des produits des poids synaptiques w_{ij} par l'activation des préneurones j dont il reçoit le signal. Soit

$$I_{ex}(i) = \sum_j w_{ij} * activation(j) \quad (2)$$

Comme le type « WEIGHTED » n'avait aucune utilité dans les réseaux simulés, j'ai modifié cet « input type » pour créer le produit de courants synaptiques nécessaire à la reconnaissance de mouvements de la rétine artificielle. En effet, il était plus simple pour moi de modifier un morceau de code que d'en insérer un nouveau et en gérer toutes les dépendances. Cette modification est présentée dans la partie sur la rétine artificielle.

v. « Noise » : Elément indispensable au bon fonctionnement des réseaux

Simbrain offre la possibilité d'ajouter du bruit dans les réseaux et d'en choisir la distribution et les paramètres. Les propriétés stochastiques sont essentielles au bon fonctionnement des réseaux de neurones dans le vivant, c'est donc légitimement l'un des sujets d'étude de l'équipe. .

Ses sources sont multiples dans le vivant, depuis l'ouverture aléatoire des canaux ioniques jusqu'à la répartition tridimensionnelle des neurones en passant par la grande variété des neurorécepteurs. Je n'ai malheureusement pas eu l'occasion de m'y intéresser pendant mon stage car mes sujets occupaient déjà

b. Les équations de Morris-Lecar

i. Présentation des équations

Les équations de Morris-Lecar sont la base de tout le projet de l'équipe « Hardware » du projet Bioinspiré. Leur histoire commence en 1952, lorsque deux éminents chercheurs en physiologie Alan Lloyd Hodgkin et Andrew Huxley ont publié leurs travaux sur les potentiels d'actions étudiés à partir de l'axone géant du calmar, pour lesquels ils ont reçu un prix Nobel en 1963. Dans ces travaux ont été formulées les équations les plus efficaces pour rendre compte du comportement électrique des neurones. Ces équations ont ensuite été simplifiées par Catherine Morris et Harold Lecar en 1981 dans le cadre de leurs travaux sur les muscles.

Les équations de Hodgkin-Huxley et celles de Morris-Lecar formalisent le comportement du neurone biologique avec des équations, il devient ainsi possible de reproduire ce comportement à l'aide d'un circuit électrique. Certes, les équations de Morris-Lecar sont moins réalistes que les équations de Hodgkin-Huxley, mais elles sont plus simples à reproduire, c'est donc celles-ci qui ont inspiré le neurone artificiel illustré sur la figure. Ce lien direct entre le circuit et le modèle nous autorise à faire l'hypothèse d'une correspondance formelle entre les équations de Morris-Lecar et *a fortiori* des neurones de *Simbrain* avec les neurones artificiels. Ainsi, l'utilisation du logiciel *Simbrain* comme outil de simulation des réseaux de neurones artificiels est licite.

Voici un schéma du circuit du neurone artificiel de l'IRCICA [28] suivi des équations de Morris-Lecar :

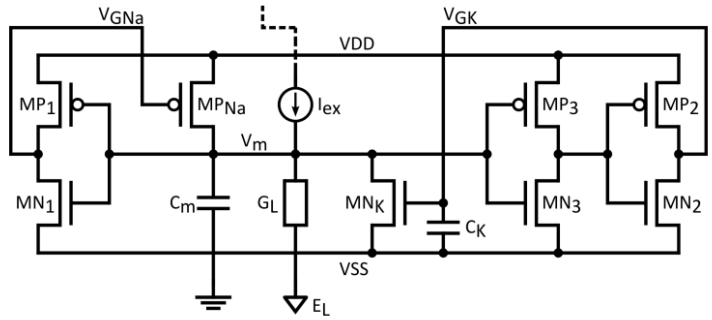


Schéma du circuit du neurone artificiel

$$C_m \frac{dV_m}{dt} = Iex - G_{Ca}.m_{SS}(V_m).(V_m - E_{Ca}) - G_K.n.(V_m - E_k) - G_L(V_m - E_L) \quad (3)$$

$$\frac{dn}{dt} = \lambda(V_m).(n_{SS}(V_m) - n) \quad (4)$$

$$m_{SS}(V_m) = \frac{1}{2}.\left[1 + Tanh\left(\frac{V_m - V_1}{V_2}\right)\right] \quad (5)$$

$$n_{SS}(V_m) = \frac{1}{2}.\left[1 + Tanh\left(\frac{V_m - V_3}{V_4}\right)\right] \quad (6)$$

$$\lambda(V_m) = \lambda_0.Cosh\left(\frac{V_m - V_3}{2V_4}\right) \quad (7)$$

Bien qu'étant une simplification, les équations de Morris-Lecar possèdent tout de même 12 paramètres. C_m représente la capacité de la membrane du neurone. Iex est le courant d'excitation du neurone reçu par ses différentes dendrites. E_{Ca}, E_k et E_L sont les potentiels d'équilibre respectivement des ions Calcium, des ions potassium et de fuite. G_{Ca},G_K et G_L sont les conductances correspondantes. n_{SS} et m_{SS} représentent les équilibres des canaux ioniques. λ_0 est la fréquence de référence. EtV_1,V_2, V_3 et V_4 sont des potentiels ajustés selon les propriétés dynamiques souhaitées.

Le modèle mathématique de Morris-Lecar est un système de deux équations différentielles non linéaires du premier ordre couplées (1) et (2). Les deux variables couplées sont le potentiel de membrane V_m qui correspond à l' « activation » dans *Simbrain*. La fraction de canaux de potassium ouverts sur le neurone n est la seconde variable et elle est calculée en interne dans *Simbrain* et nous n'y avons pas accès.

A première vue, le circuit et les équations semblent impressionnants, mais les équations rendent seulement compte d'un jeu d'équilibre des différents flux ioniques entre le neurone et son milieu.

ii. Hypothèse de correspondance formelle

L'hypothèse de correspondance formelle entre le neurone artificiel et les équations de Morris-Lecar est forte et est au cœur des problématiques sur lesquelles travaille l'équipe Hardware. Un tel circuit est un système complexe régit par des équations non linéaires et chaque composant électrique apporte son lot de perturbations. La miniaturisation du système, ainsi que l'utilisation de composants hors des limites recommandées par le fabriquant impliquent forcément un comportement neuronal qui n'est pas exactement conforme aux équations. Ainsi mon travail de modélisation s'attache à approcher au mieux le comportement du circuit sans pour autant chercher à le mimer parfaitement. De leur côté, les membres de l'équipe étudient minutieusement chaque composant afin de maîtriser au mieux le comportement du neurone artificiel et l'approcher un maximum de celui des équations.

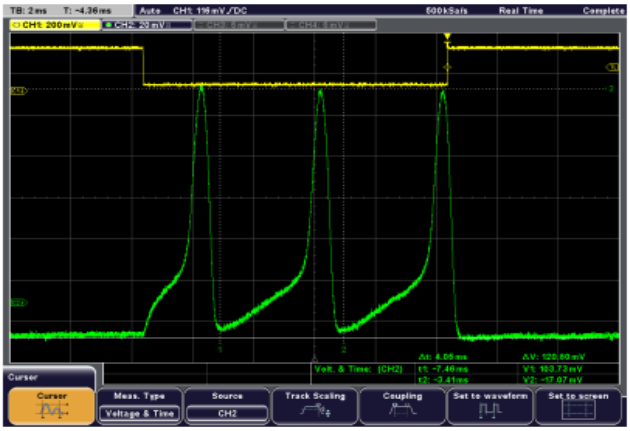
c. Calibrage du modèle

i. Objectif comportemental du neurone de Simbrain

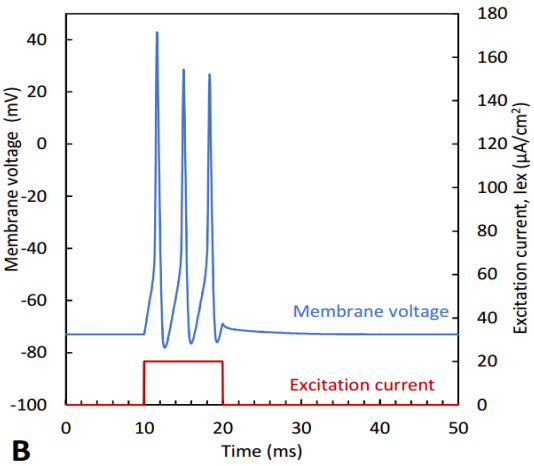
Le neurone de Morris-Lecar modélisé dans *Simbrain* doit plus ou moins se comporter comme le neurone artificiel. Pour ce travail de calibrage, j'ai principalement travaillé sur le neurone Biologique. En effet les exigences concernant le neurone Biologique sont les mêmes que pour le neurone Fast et la différence entre les deux se situe au niveau des paramètres.

Nous avons déjà parlé de la particularité du neurone artificiel et du modèle de neurone dans *Simbrain* d'être considérés comme des objets ponctuels plutôt que des objets tridimensionnels. Ainsi, là où un neurone vivant est excité par un courant mesuré en ampères, nous utilisons un courant par unité de surface pour exciter nos neurones ponctuels.

Dans un premier temps, mon repère de calibrage grossier était l'une des mesures expérimentales de la « chip 1 ». Ce calibrage est simple à vérifier : un neurone excité pendant 10 millisecondes par un courant égal à 20 microampères par centimètre carré spike trois fois. Et le potentiel membranaire au repos est situé entre -60 millivolts et -80 millivolts.
Cela revient finalement à obtenir une courbe similaire aux courbes suivantes, ou nous avons à gauche la mesure expérimentale et à droite la mesure simulée à l'aide d'un logiciel de simulation de circuits électriques.



Mesure expérimentale du neurone artificiel soumis à un courant d'excitation



Mesure simulée du neurone artificiel soumis à un courant d'excitation

ii. Résolution numérique des équations dans Simbrain

Comme présenté plus tôt, le modèle de Morris-Lecar possède 12 paramètres. Dans le logiciel *Simbrain,* il est possible de les configurer indépendamment. De plus, il est possible de choisir le pas de temps de la simulation qui entre en compte dans la résolution numérique.

La résolution numérique standard de *Simbrain* consiste en une méthode des différences finies d'Euler avec correction pour les deux variables d'intérêtV_m et n. J'ai été amené à remettre cette résolution en question, mais il se trouve que les différences de comportement entre cette méthode et des méthodes plus puissantes telles que la méthode de Runge-Kutta [8] sont négligeables.

iii. Jeu de paramètres par défaut des équations

Le paramétrage par défaut du modèle de Morris-Lecar dans *Simbrain* correspond à celui proposé dans la publication relative de 1981. Ce paramétrage est adapté au fonctionnement des cellules nerveuses dans les muscles, et est très éloigné de celui des deux types de neurone artificiel : un potentiel d'action du neurone standard dure environ 20 millisecondes, contre 3 potentiels d'action en 10 millisecondes pour le neurone artificiel.

Les cellules nerveuses des muscles fonctionnent de la même façon que les neurones, mais avec des constantes de temps plus longues. Cela s'explique par la différence de constitution de ces deux types de cellules. Dans le cerveau, les ions en jeu dans les échanges ioniques à l'origine de l'activité électrique des neurones sont les ions potassium (K+) et les ions sodium (Na+). Le sodium (Na+) est remplacé par des ions calcium (Ca+) dans les cellules du système nerveux, et ces ions ne peuvent traverser les parois neuronales aux mêmes vitesses.

Cette différence n'a pas d'influence sur les équations. Un simple changement dans les valeurs des paramètres permet de passer d'un type de cellules à un autre, de la même façon qu'on peut passer du neurone Biologique au neurone Fast.

L'outil *Simbrain* adapté à l'IRCICA a pour but principal d'étudier des architectures de réseaux. Le comportement d'un réseau est indifférent des temps caractéristiques de ses neurones tant qu'ils présentent une cohérence interne. Ainsi, dans l'attente de trouver le paramétrage correspondant aux neurones artificiels, il est possible de modéliser des réseaux avec le paramétrage par défaut des équations.

iv. Calibrage du neurone

Dans un premier temps, j'ai simplement récupéré les paramètres du neurone artificiel accessibles à partir des éléments du circuit, que j'ai implémentés après avoir effectué la conversion adéquate des unités. Cependant, comme dit précédemment, il est difficile de faire correspondre exactement le circuit aux équations. Les paramètres que j'avais n'étaient donc pas fonctionnels et une grande partie de mon travail s'est focalisée sur la recherche du paramétrage idéal.

Il existe deux approches pour déterminer les valeurs des paramètres du modèle. La méthode analytique consiste à comprendre les équations et à suivre un raisonnement afin de trouver le bon paramétrage. La méthode empirique consiste à tester un grand nombre de paramétrages jusqu'à trouver le bon. La non-linéarité des équations rend l'approche analytique difficile, et les 12 paramètres du modèle rendent l'approche empirique incroyablement longue.

Ma démarche a donc fait intervenir les deux approches, monsieur Cappy m'a aidé sur la partie analytique par sa maitrise des équations. Et j'ai reproduit les équations du modèle sur un logiciel de calcul formel afin d'accélérer l'approche empirique, car *Simbrain* ne s'y prêtait absolument pas. En effet les options concernant le tracé de valeurs y sont limitées et le temps de tracé des courbes est ralenti par la mise à jour des éléments graphiques du logiciel.

J'ai tracé à l'aide du langage Python les grandeurs nécessaires à la compréhension du modèle. L'avantage de sortir de *Simbrain* pour ce type d'étude est la possibilité de tester automatiquement des dizaines de paramétrages rapidement, de superposer les courbes si besoin et surtout de tracer les grandeurs d'intérêt.

Voici une présentation succincte des courbes tracées lors de l'étude de calibrage. Les scripts permettant de tracer ces différentes courbes sur python sont fournies en annexe.

<u>Les isoclines en zéro</u>

Les isoclines en zéro permettent d'étudier la stabilité de ce système de deux équations différentielles d'ordre 1 couplées. V_m y est représenté en abscisse et n en ordonnée. Pour rappel, les deux équations différentielles du modèle sont les suivantes :

$$C_m \frac{dV_m}{dt} = Iex - G_{Ca}.m_{SS}(V_m).(V_m - E_{Ca}) - G_K.n.(V_m - E_k) - G_L(V_m - E_L) \quad (3)$$

$$\frac{dn}{dt} = \lambda(V_m).(n_{SS}(V_m) - n) \quad (4)$$

Nous avons une isocline en zéro par équation, elles représentent l'ensemble des points d'équilibre de leur équation, c'est-à-dire les points (V_m, n) solutions des équations suivantes :

$$0 = Iex - G_{Ca}.m_{SS}(V_m).(V_m - E_{Ca}) - G_K.n.(V_m - E_k) - G_L(V_m - E_L) \quad (8)$$

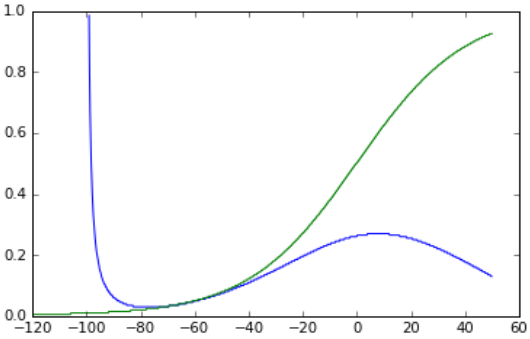
$$0 = \lambda(V_m).(n_{SS}(V_m) - n) \quad (9)$$

Les points d'intersection des courbes sont donc les points d'équilibre du système couplé. Si les pentes des deux courbes sont de même signe, alors le point d'équilibre du système est stable, et si les pentes sont de signes opposés, alors le point d'équilibre est instable.

Les isoclines en zéro permettent donc de savoir si un paramétrage est stable ou pas. Si pour un paramétrage nous n'avons que des points d'équilibre instable, alors le neurone en question va sans cesse osciller entre ses états d'équilibre instables. Si au contraire le paramétrage présente un équilibre stable, alors le neurone

va pouvoir se stabiliser à ce point d'équilibre lorsqu'il ne sera pas excité et cela correspondra à son état de repos.

La figure suivante représente les isoclines en zéro du paramétrage que j'ai utilisé pendant un temps avant de réaliser qu'il faussait les simulations car le point d'équilibre stable était trop proche de l'instabilité :

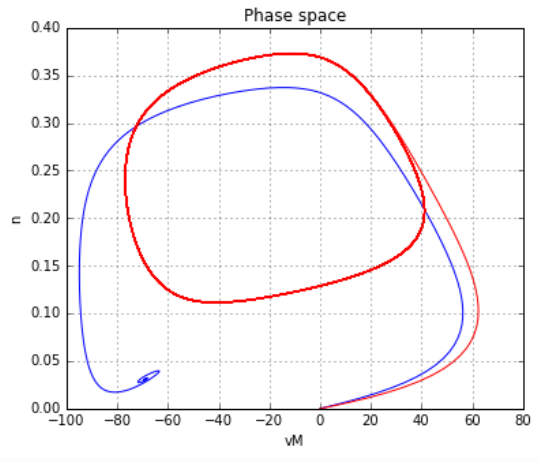


Isoclines en zéro des équations (1) et (2) pour le paramétrage déterminé en début de stage

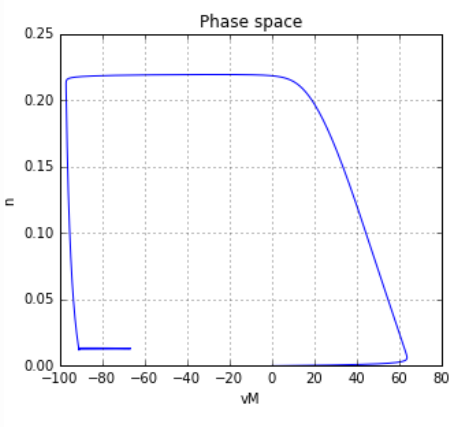
Le plan de phases

Ces sont aussi tracées sur un plan ayant pour abscisse $V_m$et pour ordonnée n. Elles représentent le comportement de ces deux variables au cours du temps. Chaque point (V_m, n) de la courbe correspond aux valeurs prises par ces variables lors de la résolution numérique des deux équations différentielles. Nous pouvons ainsi voir le point d'équilibre vers lequel le neurone converge lorsqu'il n'est pas excité. Ce point correspond à l'intersection des deux isoclines présentées précédemment.

La figure suivante illustre en bleu le comportement d'un neurone non excité dont le potentiel membranaire et l'ouverture des canaux ioniques convergent vers le point d'équilibre stable pointé par l'intersection des isoclines en zéro de la figure précédente. Le cycle en rouge représente les deux variables du même neurone lorsque celui-ci reçoit un courant d'excitation et spike :



Courbes de phases des variables *Vm* et *n* pour le paramétrage standard de Morris-Lecar



Courbes de phases des variables *Vm* et *n* pour le paramétrage obtenu en début de stage et erroné

Les valeurs d'intérêt

J'ai tracé différentes grandeurs d'intérêt liées au comportement électrique membranaire du neurone de Morris-Lecar, en commençant par l'évolution de V_m (A) et de n (C) au cours du temps, mais j'ai aussi étudié les courants correspondants à chaque flux ionique (D) ainsi que le courant total I_{tot} (B) à l'origine de toutes les variations de V_m :

$$I_{tot} = I_{ex} - G_{Ca}.m_{SS}(V_m).(V_m - E_{Ca}) - G_K.n.(V_m - E_k) - G_L(V_m - E_L) \quad (10)$$

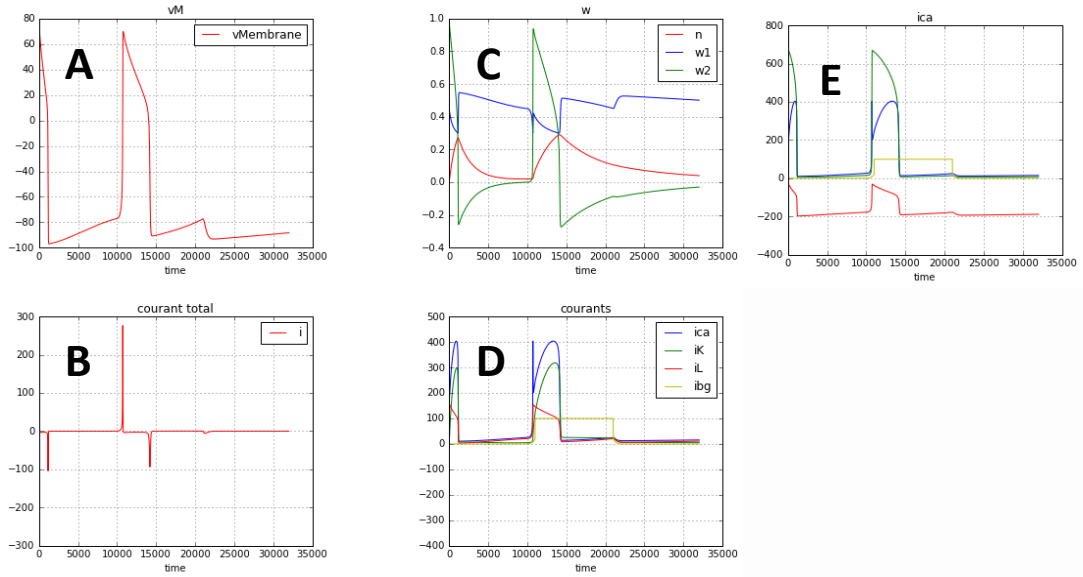
Ainsi, les variations de V_m sont proportionnelles à ce courant, l'équation (1) mène à :

$$\frac{dV_m}{dt} = \frac{I_{tot}}{C_m} \quad (11)$$

Enfin, j'ai tracé les facteurs de l'équation (2) (C) ainsi que ceux associés au courant du flux de l'ion calcium dans l'équation (1) :

$$I_{Ca} = G_{Ca}.m_{SS}(V_m).(V_m - E_{Ca}) \quad (12)$$

J'ai tracé ces courbes pour chaque jeu de paramètres testé afin de mieux cerner l'influence de chaque paramètre, les suivantes correspondent au paramétrage utilisé pour les courbes des isoclines et du plan de phases :

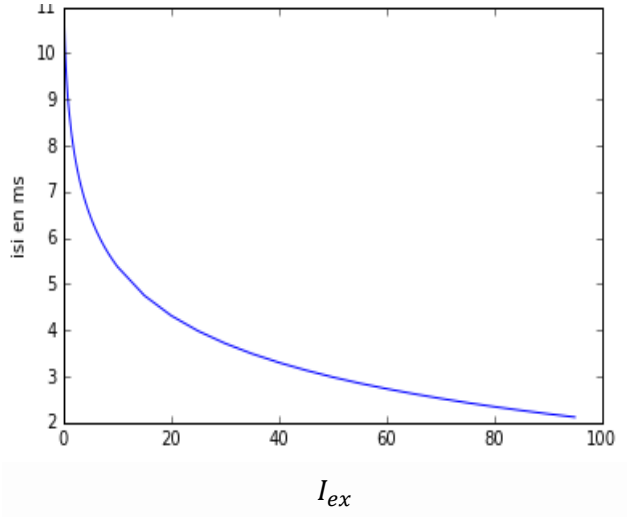


Courbes d'intérêt de la simulation du neurone artificiel

La courbe d'ISI

L'ISI', ou 'inter spike intervalle' correspond au temps qui sépare deux potentiels d'action. Pour un paramétrage donné, l'ISI' dépend uniquement de la valeur du courant d'excitation du neurone. Un courant d'excitation élevé implique une fréquence de spike du postneurone élevée aussi. La courbe d'ISI' représente donc l'ISI' en fonction du courant d'excitation reçu par le neurone.

Ainsi, la plage de courant de courant sur laquelle il convient d'exciter le neurone pour avoir différents niveaux d'excitation est déterminée par la zone de la courbe d'ISI' où la pente est la plus prononcée. La figure suivante représente la courbe d' 'ISI' obtenue pour le même paramétrage que celui des paragraphes précédents.



Courbe d'ISI obtenue pour un paramétrage donné

v. *Difficultés rencontrées*

Stabilité

Un modèle à douze paramètres interdépendants n'est pas facile à maîtriser, et la non-linéarité des équations le rend très sensible. Une très grande majorité des paramétrages possibles sont instables et le neurone spike sans cesse ; ou non viables et le neurone ne spike jamais car il ne peut pas quitter son état de repos. La fenêtre d'équilibre entre les deux, qui était mon objectif, est sensible à plusieurs paramètres et donc difficile à atteindre.

Le premier paramétrage testé, calqué sur le circuit de neurone artificiel, était instable. Cela s'explique par une approximation faite lors de la fabrication du neurone artificiel : aucun composant n'a été ajouté au circuit pour modéliser les pertes d'énergie I_L qui existent dans les équations de Morris-Lecar :

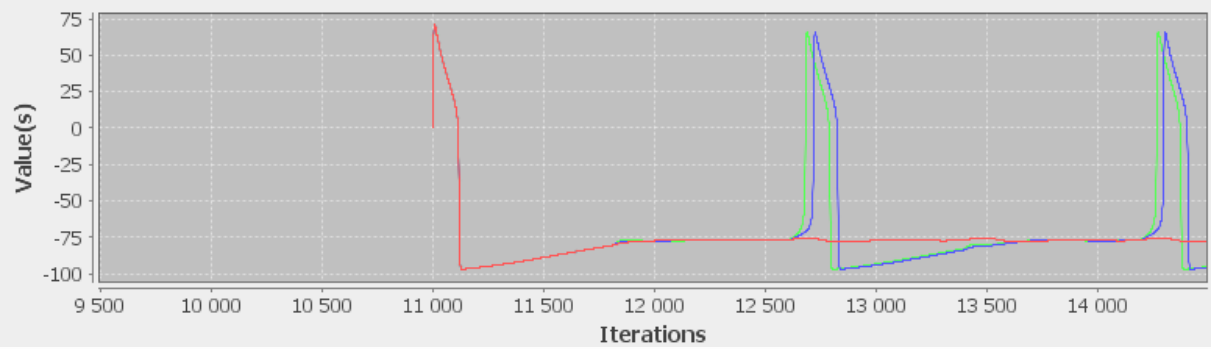
$$I_L = G_L(V_m - E_L) \quad (13)$$

En effet, une dissipation d'énergie existe naturellement dans une portion de circuit traversée par un courant. Il m'a donc fallu la modéliser. Or la modification des paramètres de perte G_L et E_L implique un nouvel équilibre de tous les paramètres. C'est cet équilibre que je n'ai pas su déterminer tout en atteignant l'objectif de calibrage *(cf. II.2.c.i)*.

Seuil de spike trop bas

J'ai tout de même trouvé pendant un temps un paramétrage qui semblait convenir à l'objectif de calibrage, celui présenté dans la partie dédiée aux courbes. Mes travaux sur le reservoir computing m'ont permis de réaliser que ce paramétrage était trop proche de l'instabilité. Sur la figure des isoclines, on pouvait observer que la stabilité du système n'est pas très prononcée graphiquement. Cette proximité avec l'instabilité s'est fait ressentir dans la sensibilité de ce neurone, dont le seuil de spike était situé à moins de 1 microampère par centimètre carré contre 20 microampères par centimètre carré idéalement. Cela faussait complètement le comportement des réseaux, car un préneurone inhibiteur pouvait faire spiker son postneurone là où il était censé l'en empêcher. Et plus généralement, un seuil trop bas implique que les neurones spikent pour la moindre excitation, alors que dans la biologie, plusieurs préneurones sont nécessaires pour faire générer un potentiel d'action de postneurone.

Les trois courbes de la figure suivante représentent la variable V_m de trois neurone possédant ce paramétrage au cours du temps, le rouge est excité par un courant égal à 0.2 microampères par centimètre carré, le bleu par 0.4 microampères par centimètre carré et le vert par 0.6 microampères par centimètre carré. On observe que le seuil de spike est très bas.



vM en fonction du temps pour trois neurones excités différemment

vi. Conclusion du travail de paramétrage

L'étude des paramètres du neurone de Morris-Lecar pour la résolution numérique m'a grandement familiarisée avec le modèle. Les paramètres que j'ai trouvés n'ont jamais été totalement satisfaisants, et la détermination du paramétrage adéquat des modèles a été remise à un autre membre du projet, afin que tout mon stage ne soit pas centré sur cette investigation imprévue. J'ai utilisé le paramétrage standard pour le reste des simulations.

Heureusement, les applications concernant l'outil *Simbrain* sont principalement des problématiques de topologie et d'architecture de réseaux, on peut les étudier indépendamment du paramétrage choisi, tant que les réseaux présentent une cohérence interne.

d. Les potentiels d'action

i. Intérêt du potentiel d'action dans la communication neuronale

Les neurones sont isolés électriquement les uns aux autres grâce à des tubes de myéline dans la biologie. Le potentiel d'action, constitue le seul moyen de communication entre des neurones partageants une synapse, un courant d'excitation chez les postneurones résulte de la cascade de réactions qu'il provoque.

Les mécanismes relatifs au potentiel d'action sont nombreux. Il existe des dizaines de types de neurones différents et chacun possède plusieurs manières de spiker. Les canaux ioniques, les neurotransmetteurs, la taille et la forme des neurones présentent une grande diversité et jouent aussi un rôle dans le phénomène de potentiel d'action.

Le modèle du neurone artificiel est simplifié et ne prend pas en compte tous ces aspects pour l'instant. Les réseaux ne sont constitués que d'un type de neurone artificiel et les potentiels d'action sont tous semblables à l'ISI près.

ii. Gestion numérique de l'aspect discret du potentiels d'action

La modélisation des neurones à spike tels que le modèle de Morris-Lecar possède deux facettes. Le potentiel membranaire V_m est une variable continue et concerne uniquement le neurone puisqu'il est isolé

électriquement. Les équations différentielles du modèle régissent son évolution. Et le potentiel d'action est un évènement discret et ponctuel qui correspond à un pic de potentiel membranaire transmettant l'information d'un neurone à l'autre.

Dans *Simbrain*, ces deux facettes sont traitées séparément. Le potentiel membranaire correspond à l'activation comme nous l'avons vu, et est mis à jour à chaque itération. Le potentiel d'action au contraire est binaire et géré par une fonction de Dirac. Ainsi le neurone artificiel génère un unique spike lorsque son potentiel effectue un pic de potentiel positif, et le modèle doit se comporter de la même façon.

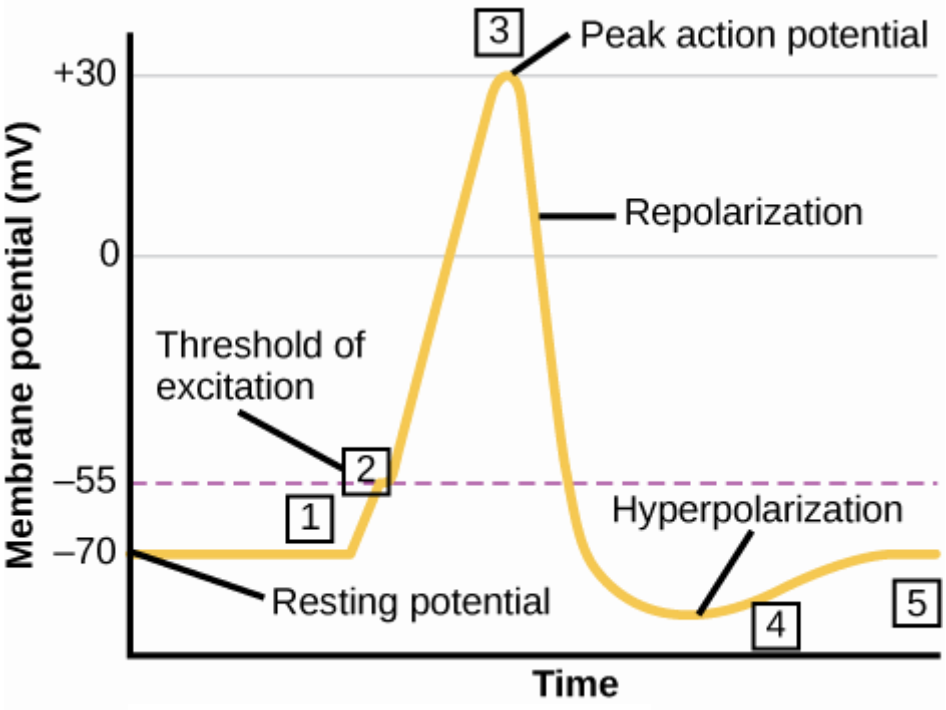
iii. Définition du potentiel d'action

Un potentiel d'action est un pic du potentiel positif de la membrane composé de trois phases consécutives. Le spike s'enclenche lorsque les sources d'excitation d'un neurone l'ont chargé jusqu'à son potentiel de seuil.

- Une fois le seuil atteint, le potentiel augmente violement jusqu'à atteindre un maximum pendant la phase de dépolarisation.
- Puis il diminue tout aussi violement lors de la phase de repolarisation.
- Après cette phase, le potentiel est plus bas que le potentiel d'équilibre du neurone au repos. Celui-ci se recharge alors pour retourner à son potentiel d'équilibre pendant la phase d'hyperpolarisation. Le neurone ne peut pas « spiker » durant ce temps de recharge. Cela permet aux systèmes de rester stables et de ne pas s'emballer.

Si le neurone reçoit toujours un courant d'excitation une fois qu'il atteint son potentiel de repos, alors le potentiel de membrane va continuer à augmenter pour atteindre la tension de seuil et spiker à nouveau. Et ce jusqu'à ce que le neurone ne soit plus excité.

Ces trois étapes correspondent à un unique potentiel d'action. Un potentiel d'action d'un neurone de la biologie dure environ 1milliseconde, et un potentiel d'action de cellule nerveuse dure environ 20millisecondes. Il est représenté sur la courbe suivante.



Etapes successives d'un potentiel d'action

iv. Modification du code Simbrain

J'ai réalisé en utilisant le logiciel que contrairement aux autres modèles de neurones accompagnés d'exemples d'utilisation, le neurone de Morris-Lecar n'avait pas, ou du moins presque pas été testé et utilisé par ses concepteurs, car aucun exemple d'utilisation n'était fourni. De plus, il présentait un défaut de conception important le rendant inutilisable dans des applications de neurone à spikes : il produisait plus d'un potentiel d'action par pic de potentiel positif, et cela faussait quasiment toutes ses applications.

Avant modification, le neurone spikait à chaque itération où son potentiel était supérieur à un seuil choisi. Il communiquait donc plus d'un potentiel d'action par pic de potentiel positif. Cela faussait particulièrement l'apprentissage plastique où le poids synaptique évolue à chaque potentiel d'action du préneurone.

Donc j'ai ajouté des conditions dans le programme pour contraindre le neurone à communiquer un seul potentiel d'action. L'introduction de variables intermédiaire a été nécessaire. Les nouvelles conditions sont les suivantes :

- Le potentiel de membrane doit être supérieur au seuil choisi. Je choisis de placer ce seuil en zéro car un potentiel d'action est un pic de potentiel positif. (Condition 1)
- Le potentiel de membrane doit présenter un maximum local, cela permet de faire spiker le neurone exactement au point maximal du pic de potentiel. Le choix du moment exact du potentiel d'action est arbitraire. (Condition 2)
- Enfin le neurone ne doit pas encore avoir spiké depuis la dernière fois que le potentiel a été négatif. Cela permet d'éviter que le neurone spike plusieurs fois lorsque des perturbations ou du bruit impliquent plusieurs maximums locaux pendant que le neurone spike. (Condition 3)

Les deux figures suivantes illustrent les conditions présentées ci-dessus :

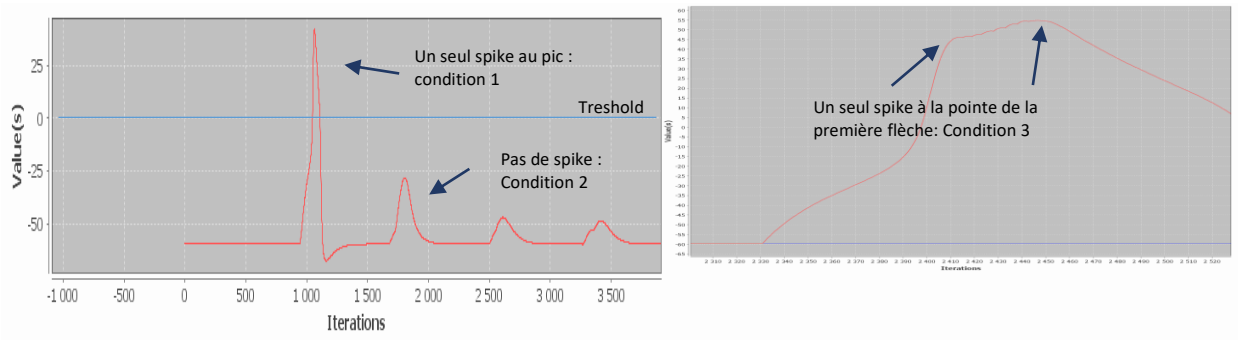


Illustration des nouvelles conditions de spike de Simbrain

3. Connexion synaptique

La synapse est un élément aussi important que le neurone dans les réseaux. Dans la biologie, la synapse correspond à la zone d'échange entre un axone et une dendrite. Pour préserver l'isolation électrique des neurones, ces deux éléments ne sont jamais en contact direct. Si l'axone et la dendrite sont très proches l'un de l'autre, alors le transfert de neurotransmetteurs est facilité et les gradients de concentration liés à l'activité du premier neurone sont mieux ressentis par le second. Le transfert d'information entre les deux neurones est donc plus efficace.

Comme pour tous les phénomènes neurologiques, les mécanismes en jeu sont complexes et font intervenir plusieurs éléments, dont les canaux ioniques et les neurotransmetteurs, le modèle utilisé pour le neurone artificiel est très simplifié.

a. La synapse *Simbrain*

Dans *Simbrain,* la synapse est représentée par un trait noir entre les deux neurones. Un disque dont la taille est proportionnelle au poids synaptique est accolée au postneurone, rouge si la synapse est excitatrice, bleu si elle est inhibitrice et gris si elle est de poids nul. La figure suivante illustre ce code couleur.

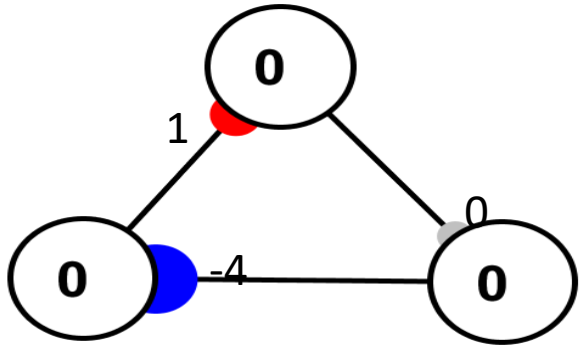


Figure permettant d'observer le code couleur des synapses dans Simbrain

Comme nous l'avons déjà dit plus haut, la physique liée aux échanges qui ont lieu dans cette zone est complexe et nous la simplifions dans nos modèles. Nous modélisons donc cette efficacité de connexion par un coefficient appelé « poids synaptique ». Un poids synaptique élevé et un seul potentiel d'action du préneurone est nécessaire pour exciter le post neurone. Au contraire, un poids synaptique faible et les potentiels d'action de plusieurs préneurones sur un court intervalle de temps seront nécessaires pour faire spiker le postneurone. Lorsque le poids synaptique est négatif, la synapse est inhibitrice.

b. Spike Responder

Comme nous l'avons dit dans le paragraphe concernant l' « input type » (*II.2.a.iv*), l' « Input type » dit « SYNAPTIC » met en jeu des courants synaptiques générés par les potentiels d'action du préneurone. En biologie, ce courant résulte des flux de neurotransmetteurs et des courants ioniques induits dans la zone d'échange de la synapse par le potentiel d'action. Dans la modélisation, nous ignorons complètement ces effets microscopiques et nous choisissons une fonction de courant d'excitation disponible parmi les « spike responders ». Pour les besoins de l'IRCICA, je ne présenterai que deux d'entre eux : « Rise and Decay » et « Jump and Decay ».

Rappel : le courant d'excitation que reçoit un neurone d' « input type » « SYNAPTIC » est à chaque instant la somme des produits entre la valeur du courant d'excitation de chacune de ses synapses et le coefficient « poids synaptique » correspondant.

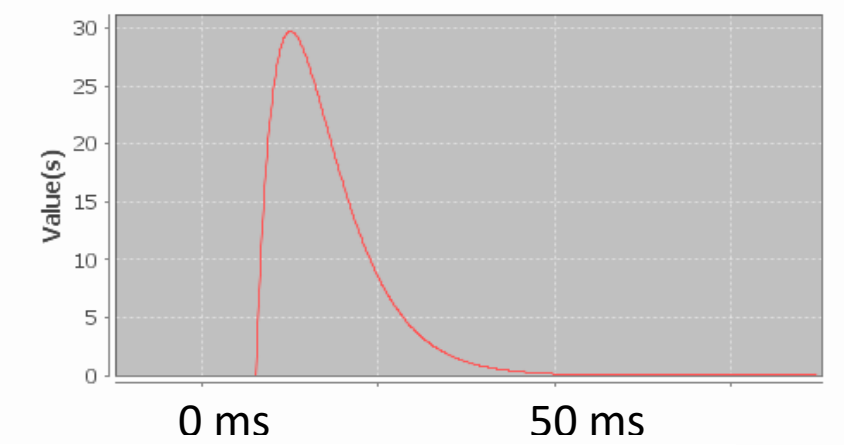
i. "Rise and Decay"

Le "spike responder" nommé "Rise and Decay" est le courant synaptique généré par un potentiel d'action similaire à ce qu'on peut observer dans la biologie. Par conséquent, le neurone artificiel et sa modélisation génèrent ce type de courant synaptique.

Ce courant correspond à la fonction suivante :

$$f(x) = \frac{bxe^{\left(-\frac{x}{a}\right)}}{a} \quad (14)$$

Tel que f atteint son maximum b pour x = a. On peut ainsi jouer sur les deux paramètres a et b pour faire varier la constante de temps et la valeur du maximum de courant. La figure suivant représente la courbe du courant « Rise and Decay » pour a = 5 ms et b = 30mV



Courbe de courant « Rise and Decay » au cours du temps

La constante de temps

Selon les observations biologiques, cette fonction s'étale sur un temps environ 10 fois supérieur à la durée du potentiel d'action qui l'a générée. Mais j'ai d'abord adapté mon modèle à ce que le neurone artificiel était capable de faire, et initialement, il s'étalait sur la durée d'un potentiel d'action. Ce n'est que plus tard qu'il a été possible de modifier cette durée sur le neurone artificiel et il reviendra donc aux prochaines personnes qui utiliseront l'outil *Simbrain* d'étudier les réseaux en intégrant cette nouvelle durée.

Le courant d'excitation maximal

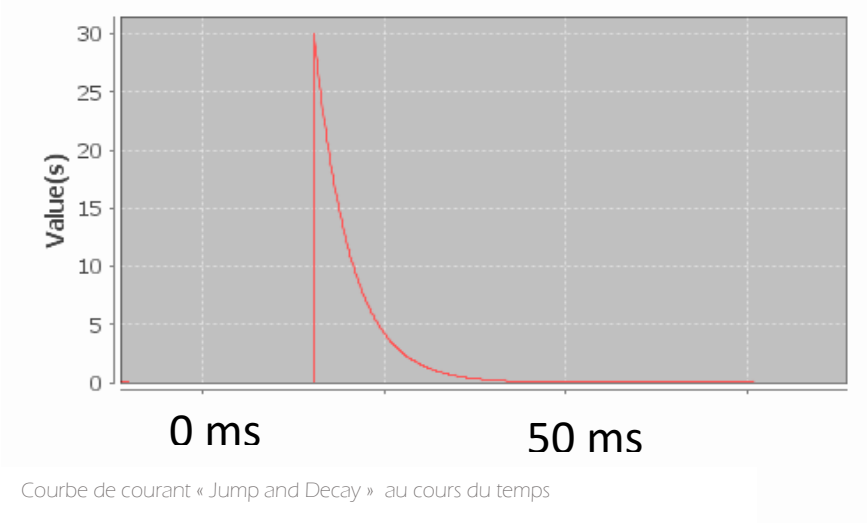
La fonction du « Rise and Decay » a une forme de cloche et elle atteint donc un maximum de courant d'excitation. Il est possible de choisir la valeur de ce maximum. Il faut veiller à ce que ce maximum ne soit pas trop grand afin qu'un seul potentiel d'action ne soit pas capable d'exciter un postneurone à lui seul. Il ne faut pas non plus qu'il soit trop faible sinon il est nécessaire qu'un trop grand nombre de neurones spikent presque simultanément pour provoquer un potentiel d'action chez le postneurone.

Une faute de frappe dans la fonction de *Simbrain* empêchait le courant d'atteindre son maximum. Cette correction fait partie des modifications que j'ai apportées à l'outil *Simbrain*. Elle était nécessaire car la valeur exacte de ce maximum est liée au nombre de potentiels d'action quasi-simultanés nécessaires pour exciter un postneurone.

ii. "Jump and Decay"

Le « spike responder » nommé « Jump and Decay » délivre au postneurone un courant constitué d'un « step », puis d'une décroissance exponentielle. J'ai utilisé ce type de « spike respondeur » afin d'effectuer de l'échantillonnage de potentiels d'action dans le cadre de la modélisation de la rétine artificielle. Cet échantillonnage permet à la rétine de percevoir des effets de mouvement dans l'image observée et est présenté plus en détail dans la partie correspondante.

Il est possible de jouer ici aussi sur la valeur maximale atteinte, soit la hauteur du « step » et sur la constante de temps de la décroissance exponentielle. La figure suivante représente la courbe de courant « Jump and Decay » pour une constante de temps égale à 5ms et un saut égal à 30mV.



Courbe de courant « Jump and Decay » au cours du temps

c. Update Rule : STDP

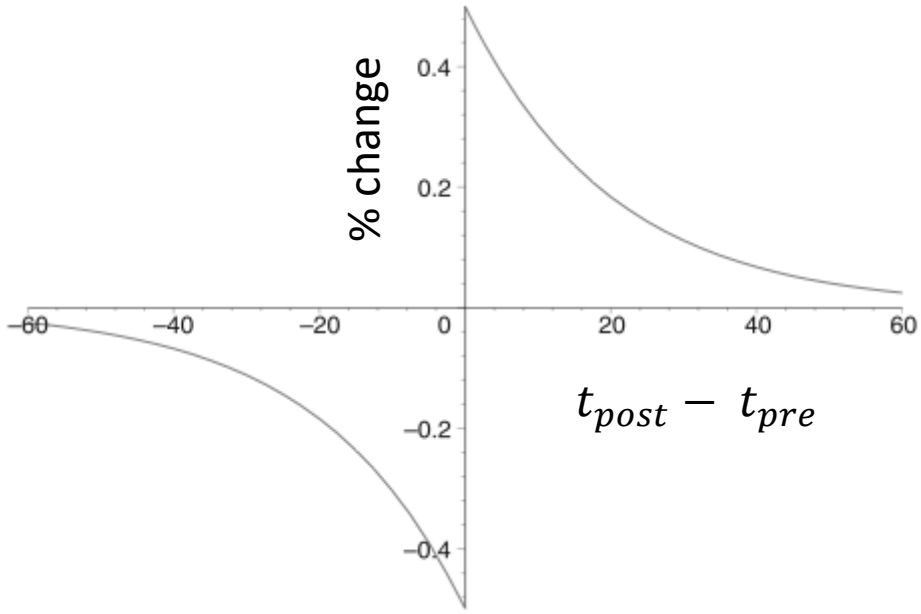
Les synapses peuvent être plastiques ou statiques dans la modélisation, dans le vivant, cette distinction est plus floue. Une synapse statique a un poids constant. Une synapse plastique peut modifier son poids et c'est la base des fonctions d'apprentissage. Il existe différentes façons de modéliser la plasticité synaptique. Celles accessibles dans *Simbrain* le sont sous l'onglet « update rule » de la synapse. Nous ne présenterons pas les différents apprentissages synaptiques possibles mais nous nous intéresserons seulement à la « Spike Timing Dependant Plasticity » (STDP). C'est la plasticité synaptique la plus connue et c'est aussi celle utilisée pour le neurone artificiel.

i. Principe

Le principe est le suivant : le poids synaptique varie lorsque les neurones en jeu spikent tous les deux sur un court intervalle de temps. Si le postneurone spike peu de temps après le préneurone, alors il y a un lien de causalité et le poids synaptique augmente. Si au contraire le postneurone spike peu de temps avant le préneurone, alors il n'y a pas de lien de causalité et le poids synaptique diminue. Plus le temps séparant les deux potentiels d'action est court, plus la modification du poids sera importante.

Ce principe mime la biologie où les événements impliquant une augmentation de poids synaptique vont pousser les dendrites à créer des excroissances au niveau de leur membrane en direction de l'axone qui les aura excitées. Et ce jusqu'à ce que axone et dendrite soient quasiment en contact – poids synaptique maximal – pour que l'information se propage plus rapidement.

Dans *Simbrain*, on peut choisir la valeur de chacun des paramètres et ainsi influer sur les temps caractéristiques et les taux de modification du poids. La figure suivante présente la courbe de la variation du poids synaptique en fonction du temps $\Delta t = t_{post} - t_{pre}$ où t_{pre} et t_{post} sont les instants respectifs de spike du préneurone et du postneurone.



Pourcentage de variation du poids synaptique selon Δt

ii. Contexte Neurone Artificiel

Concernant le neurone artificiel, il est difficile de créer un poids synaptique variable à l'aide de composants électroniques. La solution adoptée par l'équipe « hardware » est de séparer l'apprentissage en deux phases distinctes, l'apprentissage à court terme et l'apprentissage à long terme. En effet, si l'information du poids synaptique est mise en mémoire de manière analogique, c'est-à-dire sous forme de potentiel dans un condensateur, alors ce condensateur imparfait se déchargera naturellement et l'information sera perdue au bout de quelques secondes. Cette solution n'est pas envisageable pour un apprentissage à long terme. Si au contraire l'information est stockée de manière numérique, alors il sera plus difficile de la faire varier en temps réel.

Pour l'apprentissage à court terme, le poids synaptique est gardé en mémoire dans la charge d'un condensateur à décharge lente. Il va se charger ou se décharger en fonction de l'ordre des potentiels d'action du préneurone et du postneurone, selon le principe de la STDP. Le condensateur est totalement chargé dans le sens positif ou négatif seulement au bout de plusieurs potentiels d'action. Ce type d'apprentissage est à court terme car lorsqu'aucun événement n'a lieu, le condensateur se décharge naturellement.

C'est là que l'apprentissage à long terme entre en jeu. Celui-ci correspond à un interrupteur à trois positions qu'on notera « position 1 », « position 0 » et « position -1 ». Une synapse neutre est en « position 0 ». Lorsque le condensateur atteint sa charge maximale dans le sens positif, l'interrupteur passe en « position 1 » pour une synapse excitatrice. Il atteint la « position -1 » dans l'autre sens, le sens de l'inhibition. Contrairement à l'apprentissage à court terme, une fois que ces états sont atteints ils sont stables, et plusieurs événements de charge du condensateur dans un sens possible sont nécessaires afin de changer la valeur de l'interrupteur.

Cet apprentissage similaire à la STDP, nous ne cherchons pas à le modéliser sur *Simbrain*. En effet cette solution se veut être un modèle de STDP convenable. La comparaison entre la simulation et l'expérimentation réelle permettra de déterminer à quel point ce modèle est proche de celui de la STDP théorique. Cette façon de tester le modèle physique à partir des résultats de la simulation se nomme « rétro

simulation ». C'est une composante essentielle à tout processus de création scientifique, et elle est aussi utile que la simulation pour faire avancer la connaissance et comprendre les mécanismes.

iii. Modification du script

Une fois de plus, le code du logiciel *Simbrain* n'était pas satisfaisant. Initialement, la STDP différenciait les synapses inhibitrices et excitatrices pour la modification de poids, et le renforcement d'une synapse inhibitrice correspondait donc à une diminution de poids. Ce n'est pas le comportement voulu, mais une simple modification de signe dans le programme *Simbrain* a arrangé la situation.

4. Conclusions du développement de l'outil *Simbrain*

L'outil de modélisation et de simulation adapté à l'étude de réseaux de neurones artificiels et basé sur *Simbrain* est aujourd'hui satisfaisant pour l'équipe Hardware. Tous les aspects développés pour le neurone artificiel ont leur correspondance dans le logiciel. Le bruit neuronal, une fois qu'il sera maîtrisé pour le neurone artificiel, pourra être implémenté dans les modèles à travers l'onglet « noise ».

L'aspect d'optimisation des calculs que mes compétences limitées en informatique ne m'ont pas permis d'aborder pendant ce stage pourrait permettre au logiciel d'atteindre un niveau encore supérieur dans la taille des réseaux de neurones simulés en des temps abordables.

Pendant mon stage, aucun réseau de neurones artificiels n'avait été fabriqué, je n'ai donc pas eu l'occasion de comparer les réseaux réels avec les réseaux simulés.

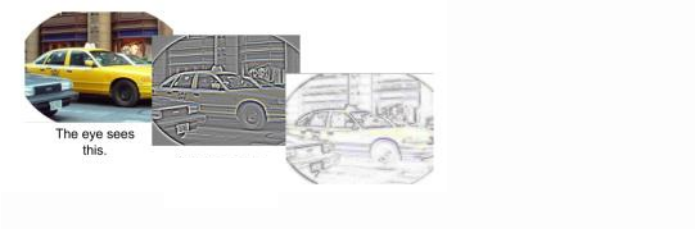
Lorsque le paramétrage adéquat sera obtenu, il sera possible de calibrer les constantes de temps associées à tous les autres éléments en fonction de celles du neurone artificiel, ainsi les applications de l'outil ne seront plus limitées à l'architecture des réseaux. L'outil *Simbrain* sera alors une véritable plateforme de test des réseaux avant leur fabrication.

Cependant, je n'ai pas réussi à extraire un exécutable du logiciel *Simbrain* depuis mon éditeur de programme. Le rendu que j'ai proposé à l'IRCICA en attendant de trouver une solution est une présentation exacte de chacune des modifications effectuées accompagnant le document de présentation du logiciel *Simbrain* et des réseaux simulés.

III. Modélisation et simulation

1. Introduction aux mécanismes biologiques de la vision

L'œil est le fruit d'une évolution de plusieurs milliards d'années, c'est une extension du cerveau. Chaque être vivant possède un œil différent dont les caractéristiques dépendent de son mode de vie. Nous nous intéressons particulièrement à l'œil humain ici. L'œil reçoit l'information lumineuse sur la rétine, l'image reçue traverse alors plusieurs couches de neurones où elle est « traitée » pour être « vue ». La figure suivante illustre ces différents traitements que nous allons présenter puis tenter de reproduire à l'aide de la rétine artificielle



Exemple d'une image traitée par le cerveau

a. La réception de l'image

Le globe oculaire correspond à un dispositif optique permettant de concentrer les rayons lumineux sur la rétine. Sa description ne nous intéresse pas car le but de ce projet est de modéliser uniquement la rétine et les parties du cerveau en jeu dans la vision.

La rétine reçoit et convertit l'information lumineuse en influx électriques pour la faire parvenir au cerveau. La rétine contient principalement trois types de cellules, les photorécepteurs, les interneurones et les ganglions.

i. Les photorécepteurs

La rétine est tapissée de photorécepteurs captant la lumière, ces cellules photosensibles enclenchent des mécanismes électriques selon la quantité de photons reçus et leurs longueurs d'ondes. Les réactions en chaînes initiées vont activer plusieurs couches de neurones ; l'image sera ainsi interprétée et reconnue par le cerveau. Les photorécepteurs sont des cônes ou des bâtonnets, illustrés sur la figure suivante.

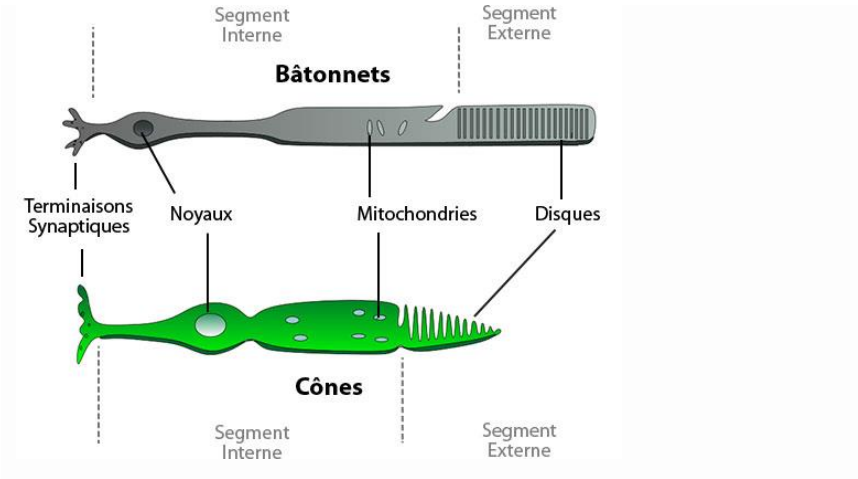


Schéma d'un cône et d'un bâtonnet

Les bâtonnets sont les plus sensibles à la luminosité, ils servent principalement à la vision nocturne. Les cônes sont plus petits et ont besoin de plus de lumière. Cependant, ils distinguent les couleurs et voient avec précision, sans eux nous serions incapables de lire.

ii. Les interneurones et cellules ganglionnaires

Les interneurones lient les photorécepteurs aux neurones ganglions. Ils ne spikent pas et ne peuvent donc pas transmettre d'information sur de longues distances. Ils ont tout de même une activité électrique et leur potentiel de membrane varie en fonction de l'activité des photorécepteurs. Ces variations enclenchent la transmission de neurotransmetteurs entre interneurones et vers les cellules ganglionnaires. Les cellules ganglionnaires transmettent ensuite l'information à certaines structures du cerveau par l'intermédiaire de potentiels d'action. L'ensemble des cellules ganglionnaires forme le nerf optique relié directement au cerveau.

Les connexions synaptiques liant les photorécepteurs aux interneurones sont d'une grande complexité. Il existe dans la rétine plusieurs types d'interneurones :

- Les cellules bipolaires (B) et les cellules horizontales (H) sont liées à un ou plusieurs photorécepteurs.
- Il existe des cellules amacrines (A) reliant plusieurs interneurones et dont la fonction est encore mal comprise aujourd'hui.

Toutes ces cellules remplissent différentes fonctions de traitement de l'image, et nous ne les comprenons pas encore totalement. Par exemple, un mécanisme de rétroaction ajuste l'excitation des cellules selon la luminosité pour s'adapter au soleil ou à la pénombre.

Enfin les cellules ganglionnaires (G) sont au bout de la chaine de neurones constituant la rétine et elles s'occupent d'acheminer l'information jusqu'au cerveau. La figure suivante est un schéma de tous les types d'interneurones de la rétine.

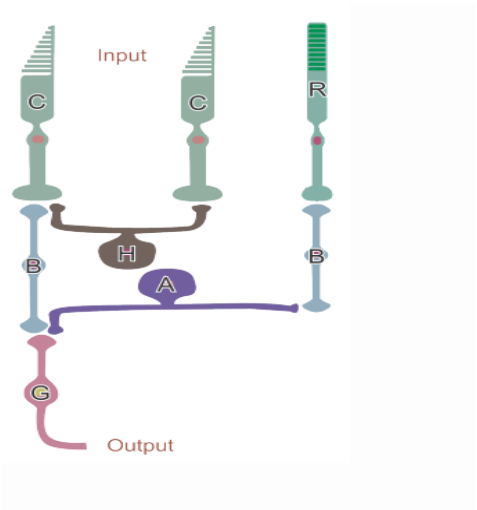


Schéma d'organisation des interneurones

La complexité des arrangements entre les différents types d'interneurones rend difficile leur implémentation lors de la fabrication des puces. Ces pourquoi dans notre étude nous commencerons par simplifier grandement cette partie de la vision pour qu'elle ne constitue qu'une couche homogène, que nous appellerons la « couche 1 ».

b. La vision de l'image

i. Le LGN

Le nerf optique est directement relié au corps géniculé latéral (LGN) ainsi qu'à d'autres structures. Ces structures sont ensuite reliées à d'autres zones du cerveau et permettent par exemple grâce aux fonctions motrices d'orienter le regard en direction d'un objet attirant notre attention. Le LGN reçoit l'image observée en contraste. A partir de l'activité de la rétine il sait quelles zones sont lumineuses et quelles zones sont sombres. Il va ainsi reconstituer deux images des contours de contrastes opposés à l'aide de cellules appelées cellules On et cellules Off. L'image suivante est une représentation simplifiée du nerf optique et des parties du cerveau liées à la vision.

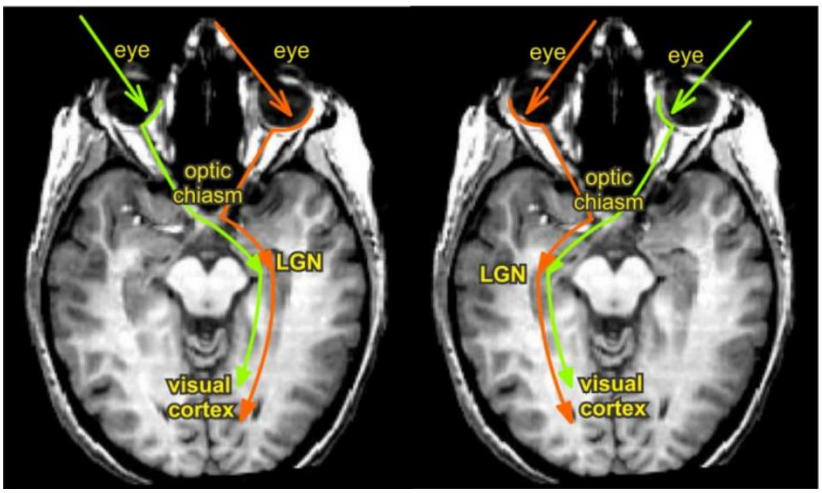


Schéma du nerf optique et des zones du cerveau liées à la vision

Les cellules On et Off synthétisent l'information issue des photorécepteurs. Elles sont reliées à une zone des photorécepteurs qu'on appelle champ de réception. Cette zone est divisée en deux parties : une partie centrale et un anneau périphérique. La figure suivante illustre le fonctionnement des neurones On et Off, les traits verts sont les potentiels d'action au cours du temps.

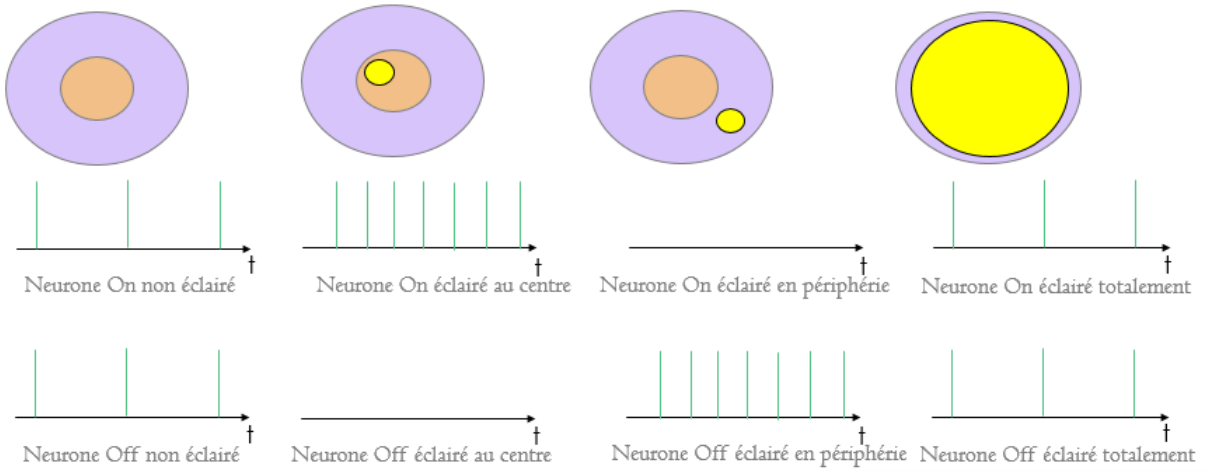


Schéma du fonctionnement des neurones On et Off

Une cellule On sera excitée lorsque la partie centrale de son champ de réception sera plus éclairée que la partie périphérique. La cellule Off sera excitée lorsque la partie périphérique du champ de réception sera plus éclairée que la partie centrale. Le LGN reconstitue les images du contraste transmises aux parties suivantes du cerveau grâce à la juxtaposition de toutes ces cellules.

En réalité les interneurones de la rétine fonctionnent aussi avec des cellules On et Off. Mais le fonctionnement exact du cerveau nous échappe encore et on ne sait pas déterminer la fonction exacte de chaque partie.

ii. Le Cortex V1

Le cortex V1 correspond à la première des six couches du cortex visuel [15]. Il est organisé en colonnes corticales [16]. Une colonne corticale est une colonne de neurones dont tous les éléments sont sensibles à la même excitation. Le cortex V1 possède donc deux couches de colonnes corticales.

Colonnes corticales simples

La première couche est composée de « cellules simples ». Ces cellules simples sont connectées aux neurones On et Off du LGN qui transmettent une information ponctuelle et localisée du contraste de l'image observée. Chaque colonne de cellules simples est sensible à une orientation particulière et à une localisation donnée, cette couche va donc recréer une image des contours de l'image observée. Dans le cerveau, les colonnes corticales sont sensibles à toutes les orientations séparées de 5 degrés, donc à 180/5 = 36 orientations différentes, une orientation est reconnue par 3 neurones On ou Off activés en même temps et juxtaposés. Les contraintes techniques du neurone artificiel nous limitent à 4 orientations différentes pour la première version de la rétine artificielle.

L'organisation des colonnes dans le cortex V1 correspond spatialement à l'image observée : les colonnes situées en haut traitent la partie haute de l'image et les colonnes situées en bas la partie basse de l'image. Nous avons donc une image rémanente formée dans notre cortex.

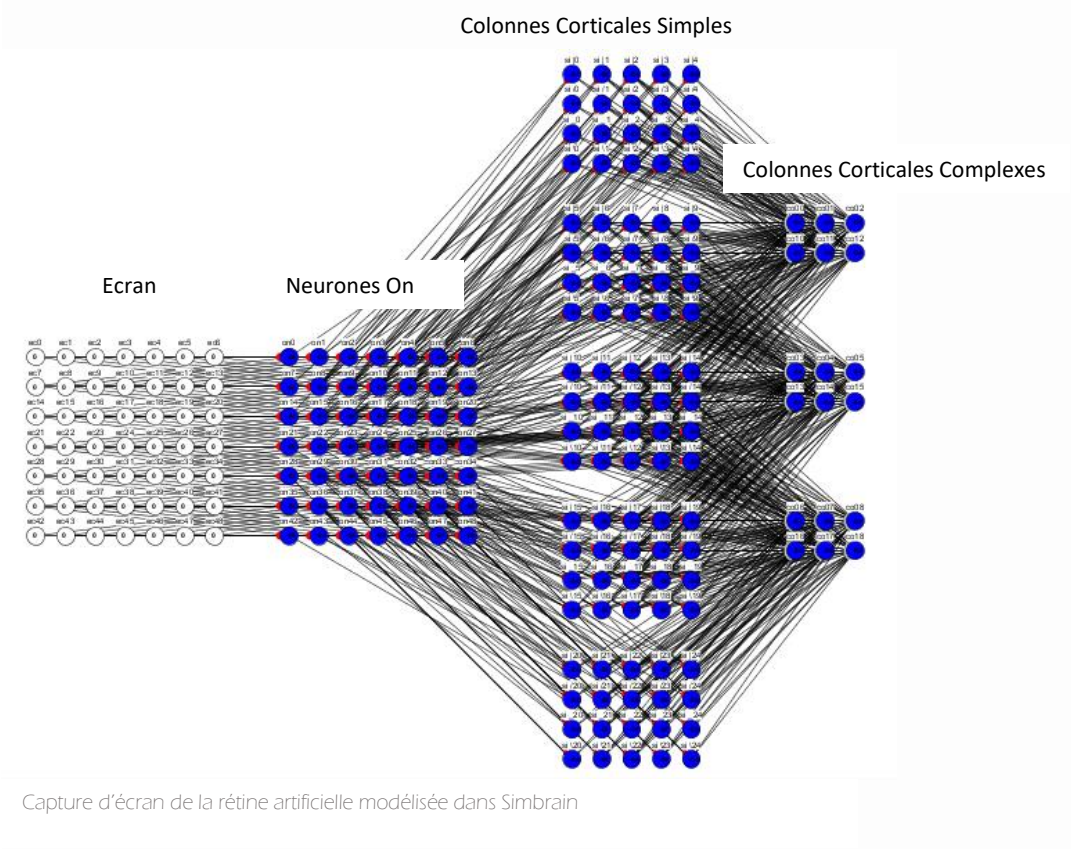
Colonnes corticales complexes

Les colonnes de cellules complexes, sont directement connectées aux colonnes de cellules simples. A partir de plusieurs orientations, elles sont capables de reconnaitre des formes. On imagine aisément qu'il est possible de reconnaître un grand nombre de formes avec seulement 4 orientations différentes.

Le cortex V1 est ensuite relié à d'autres zones du cerveau dédiées à la reconnaissance de formes, l'apprentissage ainsi qu'à la prédiction de formes. Notre rétine artificielle sera limitée à ces zones et n'ira pas plus loin. La complexité du cerveau à partir du cortex est telle que nous préférerons des astuces techniques à la modélisation du vivant pour cette première rétine artificielle.

2. Rétine artificielle

Le projet de modélisation de rétine artificielle dans le but de la créer à partir de neurones artificiels constitue le fil conducteur de mon stage. Le modèle de rétine développé est un premier essai et ne prétend pas égaler la biologie. L'objectif principal est plutôt de comprendre les mécanismes fondamentaux de la vision et de les reproduire en s'émancipant des contraintes liées à la biologie. Voici le réseau final obtenu à la fin de mon stage :



Capture d'écran de la rétine artificielle modélisée dans Simbrain

L'ensemble du système modélisé, depuis les photorécepteurs jusqu'au cortex V1 a été divisé en trois couches. Chaque couche joue le rôle d'un ou plusieurs types de cellules et remplit une fonction dans le processus de la vision.

Comme pour la rétine biologique, la rétine artificielle génèrera d'abord l'image observée en contraste avant d'en déduire des orientations et des contours, ainsi que des mouvements et des vitesses pour des images dynamiques. Ces fonctions seront effectuées de façon systématique sans apprentissage, c'est-à-dire sans plasticité synaptique. A partir de cela, les couches les plus profondes apprendront – à l'aide de la plasticité synaptique – à reconnaître certaines formes et à prédire le mouvement d'un objet à partir de ses déplacements.

Les réseaux de neurones dans le cerveau sont constitués d'un grand nombre de boucles de rétrocontrôle et d'inhibition latérale. Notre modèle de rétine artificielle ne possède pas ces caractéristiques les mécanismes sous-jacents sont assez mal compris et donc difficilement reproductibles, ceci constitue la différence fondamentale avec la biologie, et donc la principale limite de la rétine artificielle.

a. Couche 0 : Les photorécepteurs

Pour modéliser les photorécepteurs, la rétine artificielle disposera d'un maillage de photodiodes éclairées par fibre optique. J'ai reproduit ce maillage dans *Simbrain* avec des neurones de type « linéaire » se comportant comme des sources de courant continu de puissance réglable. Ce maillage constitue la « couche 0 » de la rétine artificielle nommée « écran » dans la modélisation et ne contient aucun neurone artificiel, on appellera ses éléments des pixels dans la suite et cet. Cette couche ne reproduit pas les spécificités des cônes et des bâtonnets, mais elle suffit à recréer les mécanismes fondamentaux de la vision.

Chaque pixel délivre un courant proportionnel à la puissance lumineuse qu'il reçoit aux neurones de la « couche 1 » qui l'ont dans leur champ de réception. Cela permet de présenter des images nuancées à la rétine. La courbe d'ISI (cf II.2.c.iv) des neurones de la « couche 1 » permet de déterminer la plage de courant délivrée par les pixels.

Le maillage choisi est carré, car c'est le maillage le plus simple à réaliser et il permet de discerner les quatre orientations. Dans la modélisation, je choisis la taille de l'écran et les couches suivantes s'adaptent à ses dimensions.

b. Couche 1 : Les neurones On et Off – Contraste

La « couche 1 » constitue la première couche de la rétine artificielle constituée de neurones. Elle synthétise l'action des interneurones, des cellules ganglion du nerf optique et les neurones du LGN, ou du moins ce qu'on en comprend. Elle est composée d'une couche de neurones On parallèle à une couche de neurones Off. Ces couches génèrent les images de contraste, l'une étant le négatif de l'autre. Elles n'interagissent pas entre elles. Pour économiser du temps de calcul, les couches suivantes ont été modélisées et simulées à partir d'une seule de ces deux couches, la deuxième n'étant pas nécessaire aux applications simples envisagées pour cette rétine artificielle.

Pour cette couche, les éléments importants à définir sont le contraste, le calibrage des neurones On et Off, les champs de réception et les poids des connexions synaptiques.

- J'ai introduit la notion de contraste pour la rétine artificielle en m'inspirant du contraste défini en optique. Nous avons donc le contraste perçu par un neurone On définit à partir de son champ de réception :

$$Contraste = \frac{kI_{central} - (\sum_k I_k)}{kI_{central} + (\sum_k I_k)} \quad (15)$$

  Où $I_{central}$ est le courant délivré par le pixel central et les I_k sont les courants délivrés par les k pixels périphériques du champ de réception du neurone On.
- Le calibrage de cette couche correspond à la définition du contraste minimal entre deux pixels capable d'exciter un neurone On ou Off.
- Le champ de réception des cellules On et Off sur la « couche 0 » doit comporter une zone centrale et une zone périphérique. Le choix définitif s'est porté sur l'architecture « en petites croix » présentée sur la figure suivante car elle s'étend sur peu de neurones et est performante. Elle permet de gérer simplement les effets de bord. Le neurone C correspond à la zone centrale et les neurones A, B, D et E correspondent à la zone périphérique.

Capture d'écran de l'architecture du champ de réception d'un neurone On modélisé dans Simbrain

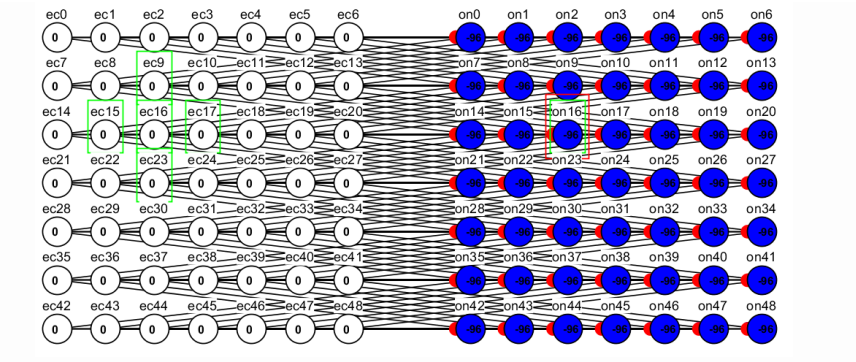
- Les poids des connexions synaptiques reliant le champ de réception au neurone sont définis à partir de ce contraste. Un éclairement homogène sur les cinq pixels du champ de réception correspond à un contraste nul donc cela n'active pas le neurone.
  Dès que le contraste atteint la valeur minimale définie au calibrage, le courant d'excitation reçu par le neurone devient assez grand pour l'exciter. Son 'ISI' est alors le plus grand possible.

Les cellules On s'activent lorsque la zone centrale est la plus éclairée et les cellules Off s'activent lorsque la zone périphérique est la plus éclairée.

On peut émettre certaines réserves concernant ce choix d'architecture de champ de réception :

- Elle ne possède aucune redondance de l'information car chaque pixel appartient exactement à une zone centrale d'un champ de réception de neurone On ou Off.
- Les images possédant beaucoup de variations locales d'éclairage peuvent générer une image de contraste incompréhensible car trop détaillée.
- Le contraste entre deux pixels voisins sur une même diagonale ne provoque aucune réaction car la zone périphérique du champ de réception est limitée aux directions horizontales et verticales.

Le niveau de simplification atteint avec cette couche correspond à l'émancipation des contraintes biologiques évoquées plus haut. Les simulations effectuées jusqu'à maintenant montraient que ce modèle était satisfaisant. Voici un zoom sur les couches 1 et 2 avec un neurone On encadré en rouge et son champ de réception encadré en vert.



Capture d'écran des couches 0 et 1 de la rétine artificielle modélisée dans Simbrain

c. Couche 2 : Les colonnes corticales simples du cortex – Orientations

La « couche 2 » modélise les colonnes corticales simples du cortex V1. Dans un premier temps, la modélisation de la colonne corticale s'est inspirée du travail d'Olivier Faugeras [17], puis nous avons élaborée des solutions plus adaptées au fonctionnement du neurone artificiel. Cette couche est directement connectée à la « couche 1 » et possède des cellules sensibles à des barres orientées dans les quatre directions et des cellules sensibles au déplacement de ces barres. On définit ici aussi le champ de réception de chaque colonne : un carré de 3 neurones On ou Off de côté et par transitivité un carré de 3 pixels de côté sur l'écran. Le nombre de colonnes de la couche dépend donc du niveau de superposition des colonnes, la figure suivante illustre les trois types d'agencement possibles.

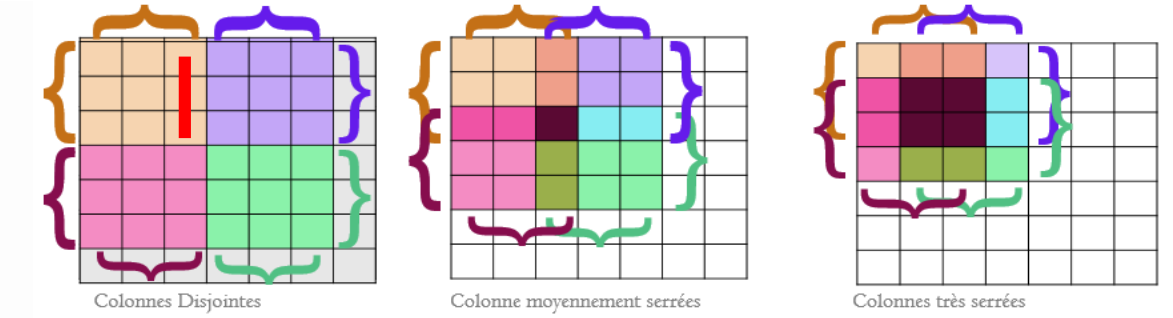


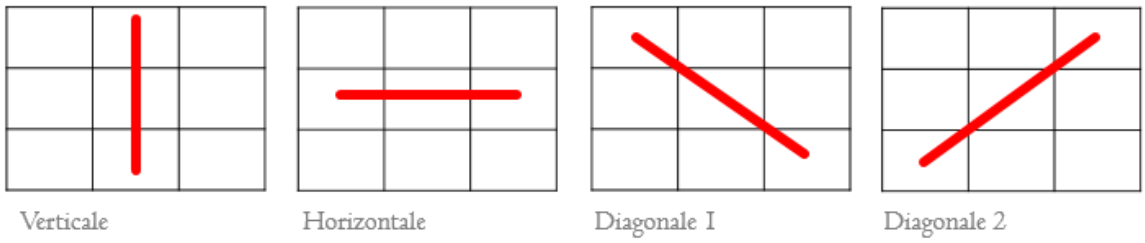
Schéma de l'agencement des champs de réception des colonnes corticales les uns par rapport aux autres

Afin de bien comprendre la problématique d'agencement, imaginons qu'on parcoure l'écran en partant du coin en haut et à gauche. On repère alors chaque champ de réception par son pixel le plus en haut à gauche.

Pour l'agencement le moins serré, on doit se déplacer de trois neurones vers la droite ou de trois neurones vers le bas pour trouver le champ de réception suivant. Dans le cas intermédiaire, il suffit de se déplacer de seulement deux pixels pour rencontrer le champ de réception suivant, on a alors une superposition des champs de réception. Enfin, dans le cas le plus serré, un seul pixel suffit et la superposition des champs de réception est plus grande. J'ai recommandé le maillage le plus serré car les autres maillages rendent impossible la détection de certaines orientations, par exemple l'orientation rouge sur le premier maillage est impossible à détecter avec les deux premiers maillages car elle n'est pas placée sur la verticale centrale du maillage.

i. Les orientations

Chaque colonne corticale modélisée possède quatre neurones d'orientation et est connectée à une zone carrée de l'écran de 3 pixels de côté. Ce champ de réception de la colonne permet ainsi de voir les deux orientations diagonales, l'orientation verticale et l'orientation horizontale. Ce sont les neurones On et Off qui vont exciter les neurones des colonnes corticales. Les poids synaptiques sont choisis de telle sorte que l'activation des trois neurones On ou Off de l'orientation vont pouvoir activer le neurone sensible à cette orientation dans la colonne. La figure suivante affiche les 4 orientations détectables respectivement par les 4 neurones de la colonne liée à ce champ de réception :



Représentation des orientations associées à un champ de réception de colonne corticale

Les simulations de cette couche furent satisfaisantes et les colonnes pouvaient reconnaître les orientations avec une très grande efficacité. Certaines réserves ont d'abord été émises pour les cas où les trois neurones de l'orientation ne spikeraient pas simultanément mais ce problème disparait dès l'instant ou l'on ajuste la constante de temps des courants synaptiques – « spikes responders » – comme dans la biologie (cf II.3.b.)

ii. Les mouvements

Le mécanisme proposé par les ingénieurs de l'équipe bioinspirée pour la sensibilité au mouvement repose sur le même principe que celui adopté pour la STDP : la charge d'un condensateur à décharge lente va garder en mémoire l'information du passage de l'objet sur un temps court. S'il a lieu avant que le condensateur se soit trop déchargé, le potentiel d'action généré par le passage de l'objet à la position suivante va se multiplier avec l'énergie restante dans le condensateur pour exciter le neurone témoin du mouvement.

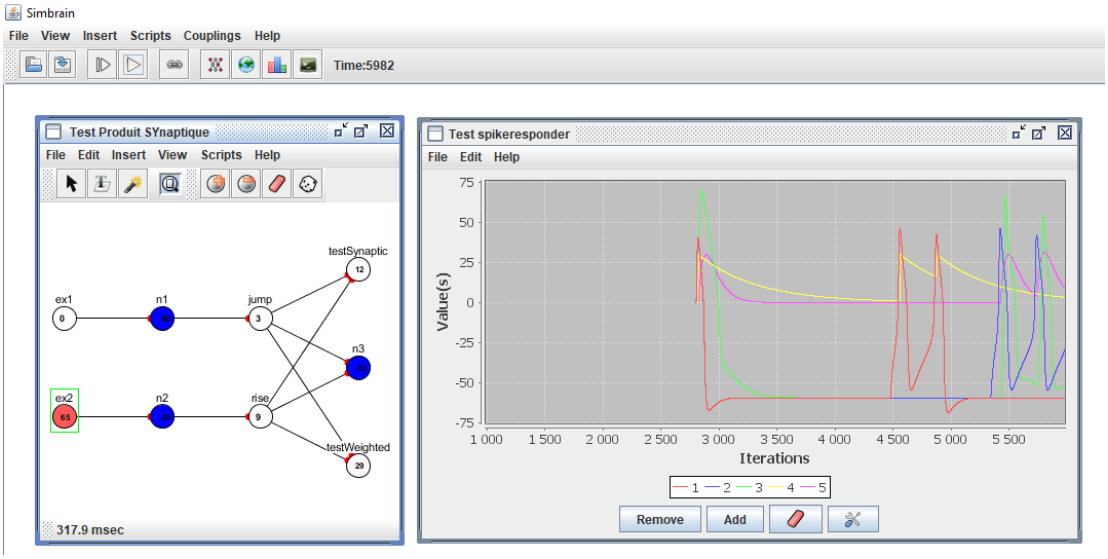
La modélisation de cette solution a nécessité la modification de l' « input type » « WEIGHTED » afin de permettre le produit de courants d'excitation dans *Simbrain (cf.* II.2.a.iv). J'ai ensuite utilisé un neurone linéaire chargé par le courant « SYNAPTIC » « Jump and Decay » (cf. II.3.b.ii) en guise de condensateur.

La décharge du condensateur est calibrée pour être bien plus longue que la durée d'existence d'un courant d'excitation provoqué par un potentiel d'action. Ainsi, le neurone témoin du mouvement vers la droite ne

peut théoriquement pas être témoin d'un mouvement vers la gauche, à moins que la vitesse de déplacement de l'objet soit très grande. Et dans ce cas, on pourrait parler d'effet d'optique pour la rétine artificielle dont les neurones des sensibles au déplacement dans les deux sens s'activeraient et elle « verrait » l'objet se déplacer en même temps dans les deux sens.

L'ingéniosité de cette solution réside dans le fait qu'elle peut déterminer la vitesse de déplacement de l'objet, car produit des courant reçu par le neurone sensible au déplacement est proportionnel à la vitesse de l'objet. En effet, si l'objet se déplace rapidement, alors le condensateur n'aura pas eu le temps de se décharger et le courant d'excitation sera élevé, si au contraire l'objet se déplace lentement, le condensateur sera relativement déchargé et le courant d'excitation sera plus bas. Ainsi, l'ISI du neurone dépendra de la vitesse et permettra donc de remonter à cette information.

La solution technique n'a été proposée que quelques semaines avant la fin de mon stage et j'ai pu la modéliser seule mais je n'ai pas pu l'implémenter sur la modélisation de rétine artificielle. La figure suivante est une capture d'écran du modèle non implémenté :



Capture d'écran du mécanisme de détection de mouvements modélisé sur Simbrain

Le neurone ***n1*** (rouge) correspond au premier neurone à voir l'objet, il charge le neurone transitoire ***jump*** (jaune) par un courant synaptique « Jump and Decay » avec une longue constante de temps. Le neurone ***n2*** (bleur) spike lorsqu'il voit l'objet passer et charge le neurone transitoire ***rise*** (rose) avec un courant « Rise and Decay » de courte constante de temps. Les neurones ***jump*** et ***rise*** sont des neurones linéaires et leur « activation » est égale au courant synaptique qu'ils reçoivent à chaque instant. Le neurone ***n3*** (vert) reçoit comme courant d'excitation la valeur du produit des « activations » des deux neurones intermédiaires. Les neurones ex1 et ex2 sont des sources de courant, et les neurones testWeighted et testSynaptic sont des témoins pour comprendre comment fonctionne l' « input type » « Weighted » modifié.

iii. Résultats

Une grande partie des tests effectués sur cette couche, principalement pour ce qui concerne les orientations, utilisaient le paramétrage erroné présenté dans la partie II.2.c.vi. avant que je ne réalise ses failles en utilisant le reservoir computing. C'est pourquoi des simulations de la couche 2 pourront plus tard remettre en question certains des choix que j'ai effectués.

d. Couche 3 : Colonnes corticales complexes – Reconnaissance de formes

Cette couche correspond à la couche des colonnes corticales complexes, capables de discerner des formes à partir des orientations de leur champ de réception. Cette couche est la première couche de la rétine non systématique car elle fait intervenir de la plasticité synaptique. Ces synapses plastiques entre la couche 2 et la couche 3 permettent un apprentissage puis une reconnaissance des formes.

Les axes de réflexion liés à cette couche s'éloignent quelque peu de la biologie dont les connaissances lacunaires de la communauté scientifique ne peuvent apporter de solutions concrètes bioinspirées.

Le champ de réception de la couche

Ici le champ de réception va énormément influencer la taille des formes reconnues. Après tout un carré peut aussi bien être sur une zone de l'écran de 3x3 pixels que de 9x9 pixels. Si le champ de réception est trop petit, alors certaines formes ne seront pas « vues » par la rétine. C'est pourquoi il est envisageable de devoir développer un mécanisme de généralisation des formes sans considérer leurs tailles.

La méthode d'apprentissage

Il existe plusieurs méthodes d'apprentissage dans les réseaux de neurones. Par exemple, le reservoir computing en est une. Dans la littérature on différentie l'apprentissage supervisé [10] et l'apprentissage non supervisé [11].

Lorsqu'il est supervisé, l'apprentissage ne peut se faire que si un expérimentateur supervise le réseau pendant sa phase d'apprentissage. C'est-à-dire qu'il montre chaque objet d'une banque d'images au réseau et lui indique la sortie attendue. L'apprentissage est terminé lorsque les poids des synapses plastiques en jeu ont convergé vers leurs valeurs finales et que le réseau est capable de reconnaître des objets similaires à ceux de l'apprentissage. Cet apprentissage est le plus simple à mettre en place, cependant il présente quelques limites :

- Le nombre d'objets présentés lors de la phase d'apprentissage peut parfois être immense. Certains réseaux doivent avoir appris sur plusieurs millions d'objets pour être capable d'effectuer une reconnaissance.
- Les réseaux d'apprentissage supervisé classiques ne sont pas capables de s'adapter, un réseau entrainé à reconnaitre des chats et des chiens sera incapable de reconnaitre autre chose qu'un chat ou un chien.
- Il semble que les contrairement à ces réseaux, les cerveaux biologiques sont capables d'apprendre de façon non supervisée et de faire preuve d'abstraction et de généralisation.

L'apprentissage non supervisé est cependant plus difficile à mettre en place et ne fonctionne pas systématiquement. Il consiste à présenter des objets au réseau pendant la phase d'apprentissage sans lui indiquer la sortie attendue, puis le laisser déterminer ses sorties et faire converger ses poids synaptiques.

Je me suis intéressé aux neurones a cliques ([12], [13] et [14]) pour l'apprentissage de la « couche 3 » mais je n'ai pas trouvé de façon de les implémenter dans la rétine artificielle.

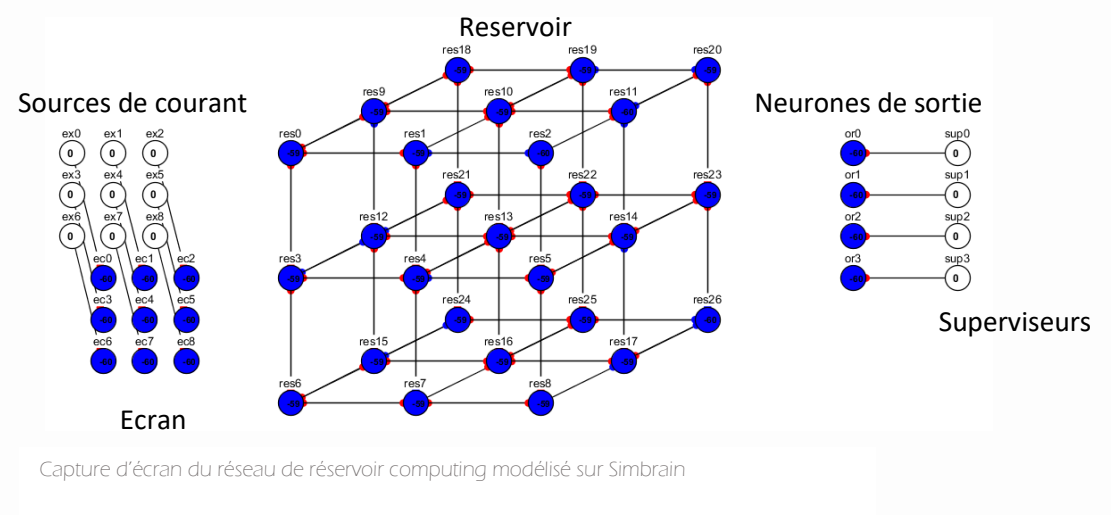
La généralisation

Il existe des dizaines de façons de dessiner la même forme géométrique sur un écran de pixels. Les neurones de la « couche 3 » doivent être capable de généraliser et d'abstraire une forme observée pour la reconnaître, indépendamment de sa taille ou de la précision du trait.

Tous ces problèmes sont liés les uns aux autres et cela nous laisse penser qu'il sera difficile de créer une « couche 3 » fonctionnelle et aussi économe que les couches précédentes. Les mécanismes de reconnaissance sont très dépendants d'autres zones du cerveau en lien avec la mémoire, le langage et les conceptions abstraites d'objets. Ainsi, je n'ai pas eu le temps d'étudier en profondeur les problématiques liées à cette couche de la rétine artificielle.

3. Reservoir computing

Le réseau nommé reservoir computing sera le premier réseau de neurones artificiels fabriqué par l'IRCICA. Les reservoir computing sont une famille de réseaux de neurones non bioinspirés pensés pour l'apprentissage. Un signal d'entrée est transmis à un réseau aléatoire dynamique de neurone appelé réservoir qui aura pour fonction d'augmenter la dimension de ce signal. Les neurones de sortie sont ensuite entrainés à lire les signaux de plus haute dimension sortant du réservoir. Voici une capture d'écran du modèle de réservoir que j'ai développé, sans les connexions entre les différentes couches pour plus de clarté :



Capture d'écran du réseau de réservoir computing modélisé sur Simbrain

Ce réseau au fonctionnement basique est fabriqué dans le but de tester le fonctionnement des neurones artificiels en réseau. Les membres de l'équipe Hardware ayant développé ce réseau savent qu'ils ont peu de chances d'obtenir un apprentissage satisfaisant pour plusieurs raisons. D'une part le réservoir de 27 neurones est relativement petit et cela n'assure pas d'augmenter assez la dimension du signal d'entrée pour l'apprentissage. De plus, les paramètres du réseau présentés dans le paragraphe suivant n'ont pas été optimisés, en effet, les logiciels de simulation de circuits électriques étaient incapables de simuler le comportement d'un tel réseau. Mais les attentes concernant ce réseau ne sont pas de cet ordre, il aura pour principal objectif de tester l'outil *Simbrain* en comparant les comportements réels et simulés du même réseau. Ainsi, les délais dans sa fabrication m'ont permis de chercher à optimiser les paramètres 'a' et 'b' par sa modélisation et sa simulation dans *Simbrain*, et cette modélisation sera déjà prête lorsqu'il sera temps de la comparer avec le réseau réel.

a. Présentation du réseau de reservoir computing de l'IRCICA

Le reservoir computing développé par l'IRCICA est un réseau simple pensé pour apprendre à reconnaitre les quatre orientations, horizontale, verticale, et les deux diagonales. Tous les neurones utilisés sont des neurones de Morris-Lecar délivrant des courants d'excitations de type « Rise and Decay » et trois potentiels d'action de préneurones sont nécessaires pour provoquer un potentiel d'action de postneurone.

Le signal d'entrée est constitué de neuf neurones de Morris-Lecar composant un écran de dimensions '3 x 3' sur lequel on peut projeter les quatre orientations. Les neurones de cet écran possèdent des connexions synaptiques réparties aléatoirement dans le réservoir selon un ratio qu'on notera 'a' dont la valeur initiale vaut 20%. Cela signifie qu'un neurone de l'écran et un neurone du réservoir ont une probabilité de partager une connexion synaptique égale à 'a'.

Le réservoir est composé de 27 neurones de Morris-Lecar représentant un cube de trois neurones de côté. Chaque neurone de ce cube est connecté à ses plus proches voisins, ces connexions peuvent être excitatrices ou inhibitrice. Le pourcentage de connexions inhibitrice est noté 'b' est a pour valeur initiale 20%. Ainsi, le nombre de voisins d'un neurone du réservoir dépend de sa position dans le cube, cette diversité asymétrique ajoute de l'efficacité au réseau.

Chaque neurone du réservoir est connecté aux quatre neurones de Morris-Lecar de sortie par des synapses plastiques de poids initialement nul. L'apprentissage aura pour but de faire converger leurs poids vers les valeurs 1 pour l'excitation et -1 pour l'inhibition. Dans la pratique, les poids synaptiques adoptent des valeurs intermédiaires pendant l'apprentissage avant de converger vers -1 et 1.

b. Résultats concernant le réseau

Déroulement de la simulation d'apprentissage

Après l'avoir modélisé, j'ai simulé le comportement du reservoir computing. J'ai observé le comportement du réservoir et programmé un apprentissage automatique dans le logiciel *Simbrain.* Le fonctionnement du réseau est le suivant :

L'expérimentateur excite les neurones de l'écran pour y faire apparaître une des quatre orientations. Les courants synaptiques générés par les potentiels d'action des neurones de l'écran enclenchent une réaction en chaine dans le réservoir et ses neurones se mettent alors à spiker dans un ordre dépendant des paramètres 'a' et 'b' et du tirage aléatoire des connexions synaptiques et de leur état excitateur ou inhibiteur. Les courants synaptiques des potentiels d'action du réservoir sont tous perçus par les neurones de sortie.
L'expérimentateur excite le neurone destiné à être sensible à l'orientation apparue à l'écran quelques instants après avoir excité l'écran. Cette excitation est brève et ne provoque qu'un potentiel d'action du neurone de sortie, cela enclenche le mécanisme d'apprentissage nommé STDP (cf. II.3.c) sur les synapses reliant le réservoir à ce neurone. De ce fait, les neurones du réservoir ayant spiké avant ce potentiel d'action du neurone de sortie vont voir le poids de leur connexion synaptique avec celui-ci augmenter. Et dans le cas où la réaction en chaîne n'est pas terminée, les neurones qui spikent après le neurone de sortie vont voir le poids de leur connexion synaptique avec celui-ci diminuer.

Valeurs des paramètres

Selon les valeurs des paramètres 'a' et 'b', la réaction en chaine du réservoir est plus ou moins longue. Si le taux de connexion entre l'entrée et le réservoir est trop élevé, c'est-à-dire si la valeur de 'a' est trop élevée, alors les signaux d'entrée vont surexciter le réservoir et ses neurones vont spiker très longtemps, voir

indéfiniment. Cet effet peut être en partie compensé par un fort taux d'inhibition dans le réservoir, obtenu avec une haute valeur de 'b'. Si au contraire le taux de connexion entre l'entrée et le réservoir est trop bas, alors les signaux d'entrée ne vont pas assez exciter le réservoir et peu de ses neurones vont spiker, parfois aucun. Cet effet sera relativement compensé par un fort taux d'excitation dans le réservoir, et donc une valeur basse de 'b'. Il est donc possible de trouver un équilibre optimal entre les valeurs de 'a' et 'b'.

Si 'a' ou 'b' est trop proche de 0% ou de 100%, le réservoir est dysfonctionnel, c'est pourquoi les valeurs optimales sont situées dans une zone intermédiaire. Le taux d'inhibition ne doit pas dépasser 50% sous peine de 'taire' totalement le réservoir. J'ai trouvé ces valeurs optimisées pour l'apprentissage: 'a' = 33% et 'b'= 20%. La valeur de 'a' égale à un tiers correspond aux trois potentiels d'action simultanés nécessaires pour exciter un neurone de l'écran. Elle assure qu'un certain nombre de neurones du réservoir enclencheront la réaction en chaîne. La valeur de 'b' correspond aux observations de la biologie, il y a généralement 20% d'inhibition dans les réseaux de neurones qu'on observe dans le vivant.

Ces valeurs des paramètres 'a' et 'b' n'assurent pas un apprentissage systématique. En effet les connexions liées à ces paramètres sont tirés aléatoirement à chaque simulation et il arrive que le réseau ne soit pas adapté pour l'apprentissage. Je n'ai pas pu tester tous les paramétrages possibles, en effet, la convergence des poids synaptiques est relativement longue et les simulations nécessitaient environ une demi-heure par paramétrage. Les autres paramétrages fonctionnels que j'ai trouvés étaient proches des valeurs proposées ci-dessus.

Autres résultats

La réaction en chaîne excite souvent une grande partie du réservoir. Il a été nécessaire de modifier les paramètres des neurones de sortie pour qu'ils soient deux fois moins sensibles, car ceux-ci recevaient de trop grands courants d'excitation au terme de l'apprentissage.

J'ai tenté de rendre le réseau symétrique en créant un réservoir sphérique, c'est-à-dire en connectant chaque face à son opposée. Ainsi, tous les neurones avaient le même nombre de voisins directs. Cette modification a totalement empêché le réseau d'apprendre à reconnaître les orientations. Cela a confirmé l'utilité de l'asymétrie générée par la forme cubique.

Améliorations apportées au logiciel *Simbrain*

L'étude du reservoir computing sur les dernières semaines de mon stage a eu un impact positif sur le développement de l'outil *Simbrain*. Ce réseau plus simple que la rétine artificielle, et dont on connait déjà le comportement théorique a été l'occasion de remettre en question et d'améliorer ma compréhension du logiciel. La simulation du reservoir computing m'a d'ailleurs permise de réaliser certaines limites de ma reprogrammation. J'ai pris consciences des limites du paramétrage du neurone de Morris-Lecar que j'avais déterminé, j'ai également redéfinit les conditions de spike à partir du comportement attendu du réseau.

IV. Conclusions

Ce stage de cinq mois constitue ma première expérience en lien avec mon projet professionnel. Il fut enrichissant sur bien des aspects et j'en garderai un souvenir positif. J'ai la sensation d'avoir contribué à un projet qui me tient à cœur et je souhaite continuer d'en suivre l'évolution.

1. Contribution au projet

Mes contributions principales au sein de l'équipe Hardware du projet bioinspiré concernent le logiciel et la rétine artificielle. D'une part, j'ai adapté le logiciel *Simbrain* au neurone artificiel pour qu'il devienne l'outil de modélisation et de simulation de prédilection de l'IRCICA, d'autre part, j'ai entamé la modélisation de la rétine artificielle et ainsi contribué à la réflexion autour de sa création.

Le logiciel *Simbrain*

Il n'était pas prévu initialement que je fasse de la programmation et ma formation en Biologie et sciences du numérique ne m'a pas apporté les compétences nécessaires pour mener à bien ce travail. J'ai appris sur le tas et je n'ai d'ailleurs même pas été capable de fournir un logiciel exécutable à l'IRCICA une fois qu'il était terminé.

Mon travail s'est focalisé sur la simulation et la modélisation et je n'ai pas l'optimisation de ses performances. J'ai cherché à ce que le neurone artificiel et sa simulation fonctionnent de manière similaire sur tous les aspects, et le résultat est prometteur, il ne reste plus qu'à le confirmer avec les essais sur le réseau de reservoir computing.

Les informations détaillées sur les modifications du logiciel sont dans le document en annexe destiné à l'IRCICA.

Base de travail de la rétine artificielle

L'objectif initial de mon stage était de modéliser la rétine artificielle. Le travail de bibliographie et de réflexion nécessaire pour mener ce projet à terme est bien plus important qu'un stage de quelques mois, et la diversification de mes objectifs au cours du stage a contribué à retarder mon avancement.

J'ai tout de même établi les bases de la modélisation de la rétine artificielle sur *Simbrain* et celles-ci faciliteront non seulement la prise en main du logiciel pour mes successeurs mais elles seront aussi réutilisables pour la suite du projet.

La simulation des couches modélisées de la rétine a fourni des résultats encourageants pour la suite, et ceux-ci le seront encore plus maintenant que tous les aspects des réseaux artificiels sont programmés et insérables dans les modélisations. En outre, les prochaines étapes de la modélisation, comme la détection des mouvements, sont prêtes à être insérées sans efforts particuliers.

Les informations détaillées sur ce réseau sont dans le document destiné à l'IRCICA en annexe, le script du réseau est aussi en annexe.

Reservoir computing

L'apport de ce réseau sur la reprogrammation de l'outil *Simbrain* a été remarquable et non négligeable, et il me semble que sa fabrication apportera d'autres éléments de compréhension des neurones artificiels

réels et numériques. C'est pourquoi je conseille fortement à l'équipe Hardware de continuer à développer des réseaux dont le comportement est connu en parallèle des projets importants comme la rétine artificielle.

Tout ce qui concerne ce réseau se trouve aussi en annexe.

Satisfaction de l'équipe Hardware

Les retours à chaud concernant ma contributions au projet bioinspiré de l'équipe Hardware ont été positifs. En tant que 'spécialiste' du logiciel *Simbrain* que j'ai développé j'envisage de me tenir à disposition si certains de mes choix son mal compris ou remis en question.

2. Apport personnel

D'un point de vue personnel, ce stage a été enrichissant et a rempli la plupart de mes attentes. Le sujet était intéressant et l'ambiance de travail agréable, l'autonomie qu'on m'a laissée et la confiance qu'on a placée en moi ont accru ma motivation.

Le projet auquel j'ai pris part s'est avéré passionnant et pluridisciplinaire. J'ai été immergé dans un véritable environnement de recherche et j'en ai découvert les contraintes et la culture, autant sur les aspects techniques qu'administratifs. L'équipe Hardware avec son projet se différencie grandement du reste du projet bioinspiré. La publication sur le neurone artificiel a eu un certain retentissement dans le milieu des neurosciences et j'ai observé l'équipe gérer le dépôt de brevets et les journalistes, et s'émanciper du projet global en créant son identité propre avec un nom, un site et un logo, tout en se projetant dans l'avenir à long terme et à court terme. J'ai apprécié la dynamique positive de l'équipe alimentée par l'ambition d'atteindre ses objectifs.

Ce stage a été l'occasion d'approfondir mes connaissances sur l'œil, le cerveau, les neurones biologiques, l'intelligence artificielle et la nanoélectronique. J'ai aussi acquis des connaissances en programmation JAVA et en modélisation de réseaux de neurones. Je ne retiens qu'un seul aspect négatif, concernant l'aspect humain. Il était agréable de travailler seul sur mon sujet et d'organiser mon temps comme je l'entendais, mais j'espère qu'à l'avenir cette expérience me permettra d'avoir plus responsabilités et de travailler à plusieurs, éventuellement de manager une équipe.

La démarche scientifique est parfois infructueuse et certaines petites avancées sont le produit de plusieurs échecs, il est donc nécessaire dans ce milieu professionnel de faire preuve d'optimisme et de ne jamais abandonner. Je sais maintenant que j'apprécie énormément le milieu de la recherche, et l'idée d'effectuer une thèse dans un domaine lié aux technologies bioinspirées a germée et pris un peu plus racine dans mon esprit.

Bibliographie

Contexte scientifique large :

[1] R. Yuste – "From the neuron doctrine to neural networks" – *Nature Reviews Neuroscience – 2015 – volume 16, pages 1 à 11.*

[2] G. Cauwenberghs – "Reverse engineering the cognitive brain" – *Proceedings of the National Academy of Sciences of the United States of America – 2013 – vol. 110, no. 39.*

[3]W. Maass – "Searching for principles of brain computation" – *sciencedirect.com, Current Opinion in Behavioral Sciences – 2016 – pages 81 à 92.*

[4} W. Maass – "To spike or not to spike, that is the question" – *Proceedings of the IEEE – vol.103, no.12, pages 2219 à 2224.*

[5] M.R. Ahmed et B.K. Sujatha – "A review on methods, issues and challenges in neuromorphic engineering" – *International Conference on Communication and Signal Processing – 2015 – pages 899 à 903.*

[6] T.P. Vogels et al. – "Neural network dynamics" – *Annual Review of Neuroscience – 2005 – vol.28, pages 357 à 376.*

[7] Y. Katayama – "Wave-Based Neuromorphic Computing Framework for Brain-Like Energy Efficiency and Integration" – *IEEE Transactions on Nanotechnology – 2016 – vol.15, issue15, pages 762 à 769.*

[8] "Runge-Kutta methods" – *Wikipedia.*

Modèles mathématiques de neurones:

[9] E.M. Izhikevich – "Which model to use for cortical spiking neurons?" – *IEEE transactions on neural networks – 2004 – vol.15, no.5, pages 1063 à 1070.*

Apprentissage de réseau de neurones :

[10] T. Laukkarinen –"Pattern recognition with spiking neural networks a simple training method" - *Proceedings of the 14th Symposium on Programming Languages and Software Tools – 2015 – pages 296 à 308.*

[11] S.J. Thorpe et T. Masquelier – "Unsupervised learning of visual features through spike timing dependant plasticity" – *PLOS Computational Biology – vol.3, issue2, 11pages.*

[12] C. Berrou et V. Gripon – "Sparse neural networks with large learning diversity" – *IEEE Transactions on neural networks – vol.22, issue7, pages 1087 à 1096.*

[13] J.Z. Tsien – "Organizing principles of real-time memory encoding: neural clique assemblies and universal neural codes" – *Trends in neuroscience – 2006 – vol.29, no.1, pages 48 à 57.*

[14] B. Larras – "Integration CMOS analogique de réseaux de neurones à cliques" – *Présentation de soutenance de thèse – 2015 – 127 slides.*

Colonnes corticales biologie et modèles :

[15] R.B. Wells – "Cortical Neurons and Circuits: A Tutorial Introduction" – *Livre - 2005*

[16] V.B. Mountcastle – "The columnar organization of the neocortex" – *Brain – 1997 – vol. 120, issue4, pages 701 à 722.*

[17] F. Grimber et O. Faugeras – "Analysis of Jansen's model of a single cortical column" – *2006 – RR-5597, INRIA – 34 pages.*

[18] B.H. Jansen et V.G. Rit – "Electroencephalogram and visual evoked potential generation in a mathematical model of coupled cortical columns" – *Biological Cybernetics – 1995 – pages 357 à 366.*

[19] N. Wagatsuma et al. – "Layer-dependent attentional processing by top-down signals in a visual cortical microcircuit model" – *Frontiers in Computational Neuroscience – 2011 – vol.5, article 31.*

Fonctionnement oeil :

[20] Berne and Levy – "The special senses" – *Principles of physiology – 2005 – chap. 8.*

[21] Auteur inconnu – "The physiology of the senses, lecture 1: the eye" – *Livre – Revision de 2016.*

[22] Auteur inconnu – "The physiology of the senses, lecture 2: the primary visual cortex" – *Livre – Revision de 2016.*

[23] Auteur inconnu – "The physiology of the senses, lecture 3: visual perception of objects" – *Livre – Revision de 2016.*

[24] Auteur inconnu – "The physiology of the senses, lecture 4: the visual sense of motion" – *Livre – Revision de 2014.*

[25] Auteur inconnu – "The physiology of the senses, lecture 5:the cerebral association cortex" – *Livre – Revision de 2015.*

[26] Auteur inconnu – "The physiology of the senses, lecture 6: visually guided actions" – *Livre – Revision de 2016.*

[27] “Receptieve field” – *page Wikipedia.*

Article de l’IRCICA concernant le neurone artificiel

[28] A. Cappy et al. – ”Spike artificial neuron in 65nm CMOS technology” – *Frontiers in neuroscience – 2017 – vol.11, pages 124 à 148.*

Annexes

Annexe 1 : Document de présentation des scripts et de l'outil Simbrain à destination de l'IRCICA

Présentation pratique du travail sur *Simbrain*

1. Introduction

J'ai effectué mon stage à l'IRCICA entre avril et aout 2017. L'objectif principal de mon stage était de mettre à disposition de l'équipe bioinspirée un outil de modélisation et de simulation de réseaux de neurones compatibles avec le neurone artificiel.

Ce document doit être considéré comme une annexe de mon rapport de stage. Cette présentation pratique de mon travail sur le logiciel *Simbrain* est censée faciliter sa prise en main et la reprise des modélisations et simulations où je les aurai laissées. Toutes les informations théoriques absentes de ce document sont disponibles sur mon rapport de stage.

2. Simbrain

a. Présentation

Le logiciel *Simbrain* a été conçu dans le but de proposer une interface instinctive et accessible pour la modélisation de réseaux de neurones. Il a été choisi par l'équipe Hardware car il est l'un des rares logiciels à simuler le modèle de Morris-Lecar sur lequel s'appuie le neurone artificiel, et il permet de suivre au cours du temps la variable d'intérêt qu'est le potentiel membranaire. De plus, sa prise en main est instinctive et très documentée.

Pour que l'outil *Simbrain* devienne la base d'étude des réseaux de neurones artificiels avant de les fabriquer, il faut l'adapter un peu. Ma démarche a été d'étudier les réseaux de neurones biologiques et de comprendre les solutions techniques proposées par l'équipe Hardware. J'ai modélisé quelques réseaux dont le « reservoir computing » et la rétine artificielle pour tester et adapter le logiciel au fur et à mesure des nécessités rencontrées.

Le programme de *Simbrain* ainsi que les scripts des modèles sont codés en JAVA.

b. Conseils d'utilisation

Les paragraphes suivants ne prétendent pas servir de guide d'utilisation de *Simbrain* et il est nécessaire d'expérimenter et de prendre en main le logiciel de façon autonome. Ce sont simplement des astuces contournant des défauts de programmation que j'ai remarqués ; ainsi que des conseils relatifs à l'utilisation que j'ai faite du logiciel.

i. Optimiser la construction des réseaux

Dans un premier temps, il est à mon sens plus simple de passer par l'interface graphique de *Simbrain* pour le découvrir et le comprendre. Toutefois, cette interface montre rapidement ses limites et le script est plus approprié pour vraiment modéliser des réseaux.

Les limites de l'interface

Les défauts de l'interface soulevés sont liés à des défauts de programmation que je n'ai pas su modifier. Même si une solution est trouvée, le script reste plus adapté pour la généralisation des réseaux, leur clarté et la liberté.

- Les boites de dialogues des paramètres sont peu fiables. Certains paramètres ne prennent pas la valeur qu'on leur impose, ou sont liés à d'autres paramètres, c'est le cas du potentiel de perte dans le neurone Morris-Lecar. De plus, les nombres à virgule ont tendance à être tronqués à l'unité près lorsqu'on les a écrit avec un '.' plutôt que ',' ou qu'on a rouvert la boite de dialogue.

- Pour les réseaux de taille importante, le copier-coller dans l'interface peut faire gagner un temps important. Cependant, il apporte des erreurs, comme le fait de tripler les neurones copiés plutôt que de les doubler, et il gère mal les connexions synaptiques.

- Il est contraignant de créer un réseau de grande taille depuis l'interface, en effet, le point de vue se calibre sur tout le réseau à chaque mouvement.

Astuces concernant le script

- Pour modéliser des réseaux, l'éditeur de script proposé par *Simbrain* est imparfait, à chaque réouverture du script dans cet éditeur, un saut de ligne a été inséré entre chaque ligne. Il est donc préférable d'utiliser un éditeur de texte à part. La difficulté principale concernant le développement de réseaux par script est la recherche d'erreur. Il faut tester le réseau à chaque étape de la programmation pour éviter d'avoir à trouver une erreur parmi tout le code à la fin.

- Le script permet de définir tous les types d'objets une fois pour toutes au début du programme. Mais des conflits peuvent apparaître entre les mêmes objets.

- On peut aussi y définir tous les paramètres qu'on voudra modifier en début de programme pour qu'ils soient accessibles et modifiables (exemple : taille des réseaux, paramètres des fonctions de courant).

- Seul le script permet d'ajouter une part d'aléatoire dans la construction des réseaux. Plus généralement, il permet de programmer tout ce que le langage JAVA permet de programmer.

- Le fait de pouvoir placer exactement les neurones où on le souhaite rend l'aspect des réseaux plus soigné.

ii. Optimiser les simulations sans modifier le logiciel

Le temps de simulation sera grandement amélioré si l'on met à jour certains neurones en parallèle. Cela nécessite des compétences en informatique plus poussées que les miennes. Il est tout de même possible d'optimiser le temps de simulation des réseaux de façon non négligeable assez simplement.

- Pour chaque simulation, un temps de calcul est nécessaire pour mettre à jour les interfaces graphiques à chaque itération. Cela prend beaucoup de temps. Pour empêcher ces mises à jour parfois inutiles de se faire, il suffit de réduire les fenêtres à l'intérieur de *Simbrain.*

- Dans le même état d'esprit, chaque courbe tracée utilise un temps de calcul non négligeable. Il ne faut donc pas surcharger en tentant de tracer toutes les courbes, et bien choisir les courbes qu'on veut voir.

- Les simulations ralentissent après un grand nombre d'itérations. Je ne sais pas l'expliquer, mais il est préférable de réinitialiser *Simbrain* pour faire une nouvelle simulation plutôt que de simuler dans la continuité de la précédente.

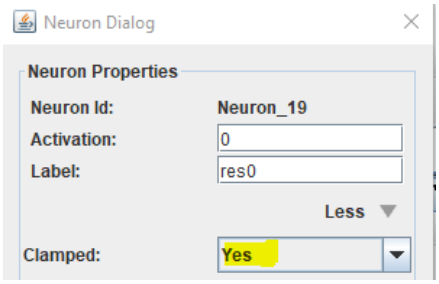
iii. Gestion des valeurs des sources de courant

J'ai appelé « pixels » les neurones basés sur le modèle ***linear rule***, ils fonctionnent comme des sources de courant continu pour les neurones de Morris-Lecar. Il existe plusieurs manières de les utiliser. On peut gérer le courant délivré à la main depuis les boites de dialogue, ou avec les tableaux ***input table*** et ***dataworld***.

Un script permettant de créer les tableaux compatibles, d'extension '.csv' sur python, ainsi qu'un tableau généré sont donnés en annexe. Les tableaux doivent être enregistrés dans le dossier simulations/tables de *Simbrain* pour être accessibles plus rapidement.

A la main

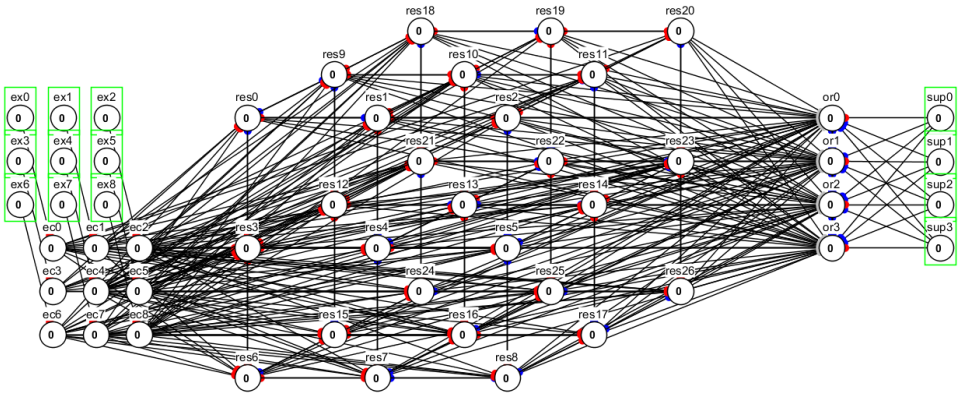
Lorsqu'on modifie leur ***activation*** à la main, il faut que le paramètre ***clamped*** soit ***vrai*** pour que l'activation ne retombe pas à zéro à l'itération suivante. Cette méthode permet de se familiariser avec le comportement d'un réseau dans un premier temps avant d'utiliser les tableaux.



Paramètre clamped

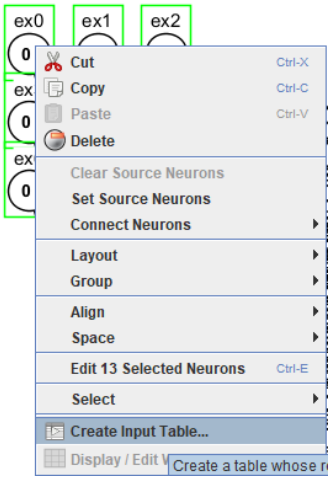
Avec une 'input table'

- Les « pixels » peuvent être indifféremment ***clamped*** ou non pour cet usage
- sélectionner les « pixels » d'intérêt

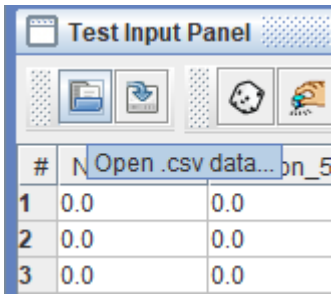


Neurones d'intérêt encadrés en vert

- clic droit sur la sélection -> 'Create input table …' -> 'Open .csv data…' -> choix du tableau dans les fichiers de l'ordinateur

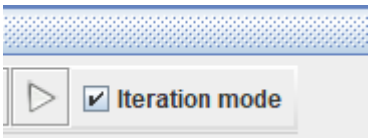


create input table



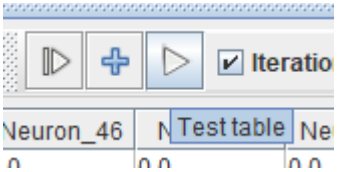
Open .csv data

- cocher 'iteration mode'



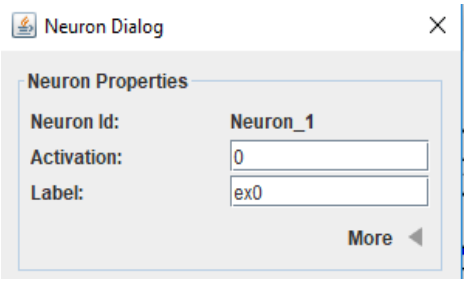
Iteration mode

- 'test table' pour lancer les itérations du réseau avec les pixels prenant les valeurs indiquées dans le tableau



Test table (play)

A chaque neurone du réseau est assigné un numéro identifiant. Le tableau ‘input table’ ne prend pas en compte l’ordre dans lequel a été faite la sélection des pixels pour leur assigner des colonnes du tableau mais seulement l’ordre des identifiant. Ainsi la colonne la plus à gauche correspondra au neurone dont l’identifiant est le plus bas.



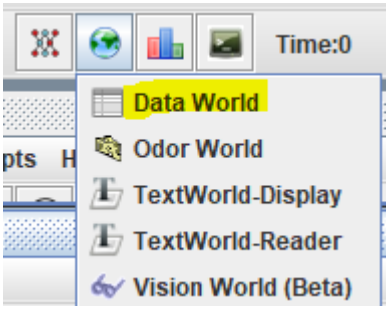
Identifiant de neurone

Simbrain ne montre pas le déroulement des itérations avec l’***input table*** et ne trace aucun graphe. On ne peut voir que l’état final du réseau après un parcourt complet du tableau. Cette méthode est la plus rapide.

Avec un ***data world***

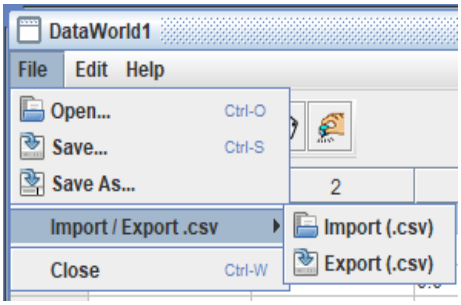
Cette méthode utilise le même type de tableau géré différemment par le logiciel.

- Les « pixels » doivent ne pas être ‘clamped’ pour cet usage
- Sélectionner ***dataworld*** dans le bouton avec une petite planète



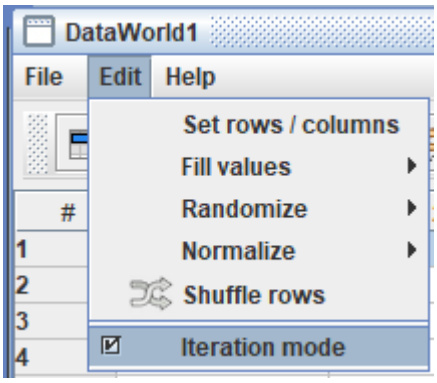
Dataworld

- Dans le ‘data world’ : ‘file’ -> ‘import / export .csv’ sélectionner le tableau



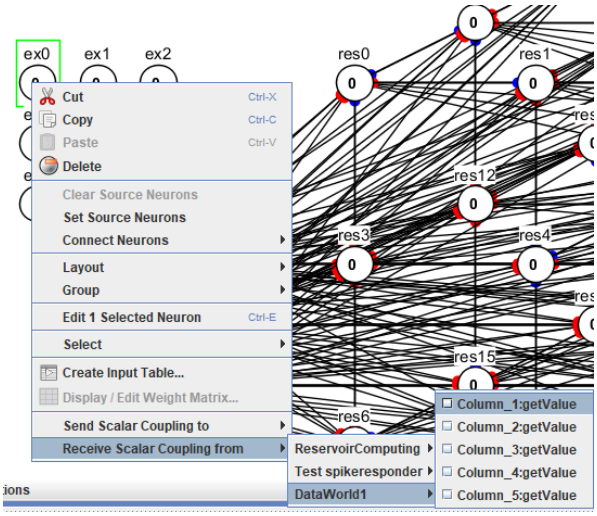
File, import/export .csv

- Dans le ‘data world’ : ‘edit’ -> ‘iteration mode’



Edit, iteration mode

- clic droit sur chacun des « pixels » à connecter au tableau et sélectionner ‘receive scalar coupling from’-> ‘dataworld’ -> ‘colonne correspondante’



Receive scalar coupling, dataworld, column_ :getvalue

Avec cette méthode, chaque itération du réseau prend en compte le tableau. On peut donc faire évoluer le réseau normalement et les entrées du tableau seront prises en compte. Ainsi, l’évolution du réseau sera visible et les courbes seront tracées en temps réel.

c. Améliorations apportées

Avant d’effectuer toute modification, j’ai passé beaucoup de temps à lire le code source du logiciel disponible sur Github pour m’en imprégner et être sûr que je n’apportais pas de solutions à des problèmes qui n’existaient pas.

L’adaptation de l’outil *Simbrain* s’est faite au fur et à mesure des besoins spécifiques rencontrés lors de la modélisation et la simulation de la rétine artificielle et du « reservoir computing ». Le logiciel

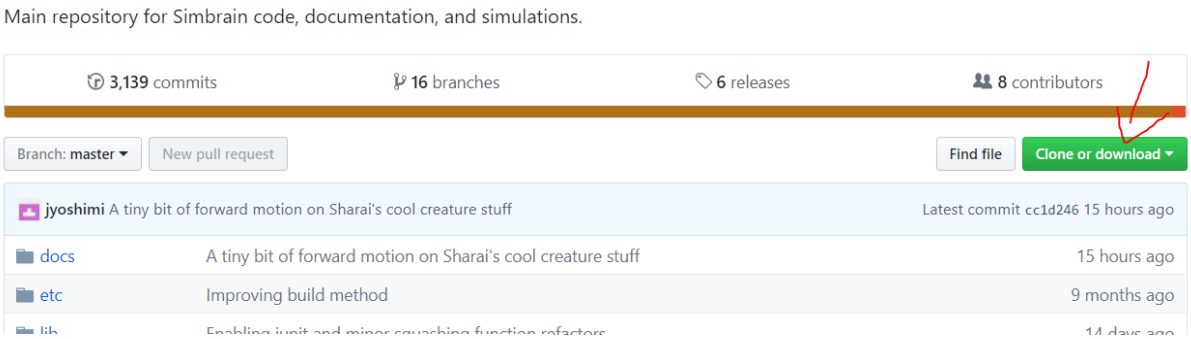
est donc maintenant capable de répondre aux besoins des réseaux actuels mais nécessitera surement de nouvelles modifications pour modéliser les réseaux suivants.

Le logiciel *Simbrain* modifié pour les besoins de l'IRCICA peut sembler satisfaisant à première vue mais seule la confrontation entre la simulation et la réalité permettra de dire s'il convient à la simulation des réseaux de neurones artificiels.

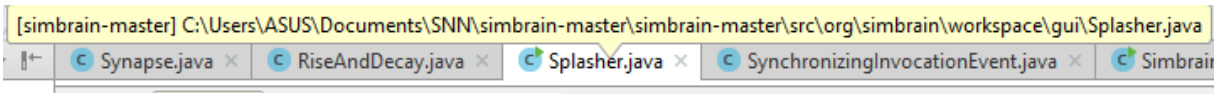
Comment effectuer une modification sur le logiciel *Simbrain*

Pour effectuer des modifications sur un logiciel en accès libre tel que *Simbrain*, il faut dans un premier temps se munir d'un 'environnement de développement' (IDE). J'ai principalement utilisé l'environnement de développement Intellij IDEA.

Ensuite il suffit d'importer le code source de Simbrain depuis https://github.com/simbrain/simbrain en cliquant sur 'clone or download', de faire les modifications nécessaires sur le projet, puis de définir le main ***splasher*** avant d'exécuter le nouveau logiciel. Je n'ai pas réussi à extraire un exécutable des IDE que j'ai utilisés. Les modifications que j'ai effectuées sur *Simbrain* doivent donc être reproduites sur un nouveau projet.



Clone or download



Main : splasher

i. Modification 1 – Spike du neurone de Morris-Lecar

Adresse dans Simbrain : /src/org/simbrain/network/neuron_update_rules/MorrisLecarRule.java

Présentation de la modification

Le modèle de Morris-Lecar rend compte du comportement du potentiel de membrane d'un neurone, et ne considère pas le spike. Ces variations de courant impliquent des effets sur le circuit du neurone artificiel mais la simulation informatique utilise une information binaire de potentiel d'action pour induire ces effets. Le neurone simulé doit transmettre l'information binaire relative à un spike une

seule fois pour chaque pic de potentiel. Pour valider cette condition il a été nécessaire de modifier le programme concernant la mise à jour des neurones Morris-Lecar.

Portion de code modifiée

En rouge les éléments supprimés et en vert les éléments ajoutés:

```
   /** Boolean giving the current variation of vMembrane **/
   private boolean isRising = true;

   /** Boolean giving the previous variation of vMembrane **/
   private boolean wasRising = true;

   /**Boolean that allow the neuron to spike once when its potential gets positive **/
   private boolean hasNotSpikedYet = true;

public void update(Neuron neuron) {
   double dt = neuron.getNetwork().getTimeStep();
   double i_syn = inputType.getInput(neuron);
   // Under normal circumstances this will cause no change.
   double vMembrane = neuron.getActivation();

   double dVdt = dVdt(vMembrane, i_syn);
   double dWdt = dWdt(vMembrane, w_K);

   double vmFut = vMembrane + dt * dVdt;
   double wKFut = w_K + dt * dWdt;
   double vM1 = vMembrane;
   vMembrane = vmFut;
   vMembrane = vMembrane + (dt/2) * ((dVdt) + dVdt(vmFut, i_syn));
```

//Cette modification de la mise à jour de vMembrane permet seulement de gagner un peu de temps lors des simulations.
//Jusqu'à maintenant tous les tests ne montraient aucune différence entre les deux résolutions numériques.

```
   double vM2 = vMembrane;
   double nDer = vM2 - vM1;
   wasRising = isRising;
   if (nDer >= 0 ) {
      isRising = true;
   }else {
      isRising = false;
   }
   if (vM1 > treshold && vM2 < treshold){
      hasNotSpikedYet = true;
   }

   w_K = wKFut;
   w_K = w_K + (dt/2) * ((dWdt) + dWdt(vMembrane, wKFut));

   neuron.setSpkBuffer(topSpike(isRising, wasRising, vMembrane, hasNotSpikedYet));
   setHasSpiked(topSpike(isRising, wasRising, vMembrane, hasNotSpikedYet), neuron);

  neuron.setSpkBuffer(vMembrane > threshold);

  setHasSpiked(vMembrane > threshold, neuron);

   if (topSpike(isRising, wasRising, vMembrane, hasNotSpikedYet)) {
      hasNotSpikedYet = false;
```

```
    }
    neuron.setBuffer(vMembrane);
}

private boolean topSpike(boolean isRising, boolean wasRising, double vMembrane, boolean
hasNotSpikedYet ){
    if (!isRising && wasRising && vMembrane > threshold && hasNotSpikedYet) {
        return true;
    }else{
        return false;
    }
}
```

Extrait de mon rapport de stage consacré aux conditions de spike

i. *Modification du code Simbrain*

J'ai réalisé en utilisant le logiciel que contrairement aux autres modèles de neurones accompagnés d'exemples d'utilisation, le neurone de Morris-Lecar n'avait pas, ou du moins presque pas été testé et utilisé par ses concepteurs, car aucun exemple d'utilisation n'était fourni. De plus, il présentait un défaut de conception important le rendant inutilisable dans des applications de neurone à spikes : il produisait plus d'un potentiel d'action par pic de potentiel positif, et cela faussait quasiment toutes ses applications.

Avant modification, le neurone spikait à chaque itération où son potentiel était supérieur à un seuil choisi. Il communiquait donc plus d'un potentiel d'action par pic de potentiel positif. Cela faussait particulièrement l'apprentissage plastique où le poids synaptique évolue à chaque potentiel d'action du préneurone.

Donc j'ai ajouté des conditions dans le programme pour contraindre le neurone à communiquer un seul potentiel d'action. L'introduction de variables intermédiaire a été nécessaire. Les nouvelles conditions sont les suivantes :

- *Le potentiel de membrane doit être supérieur au seuil choisi. Je choisis de placer ce seuil en zéro car un potentiel d'action est un pic de potentiel positif. (Condition 1)*
- *Le potentiel de membrane doit présenter un maximum local, cela permet de faire spiker le neurone exactement au point maximal du pic de potentiel. Le choix du moment exact du potentiel d'action est arbitraire. (Condition 2)*
- *Enfin le neurone ne doit pas encore avoir spiké depuis la dernière fois que le potentiel a été négatif. Cela permet d'éviter que le neurone spike plusieurs fois lorsque des perturbations ou du bruit impliquent plusieurs maximums locaux pendant que le neurone spike. (Condition 3)*

Les deux figures suivantes illustrent les conditions présentées ci-dessus :

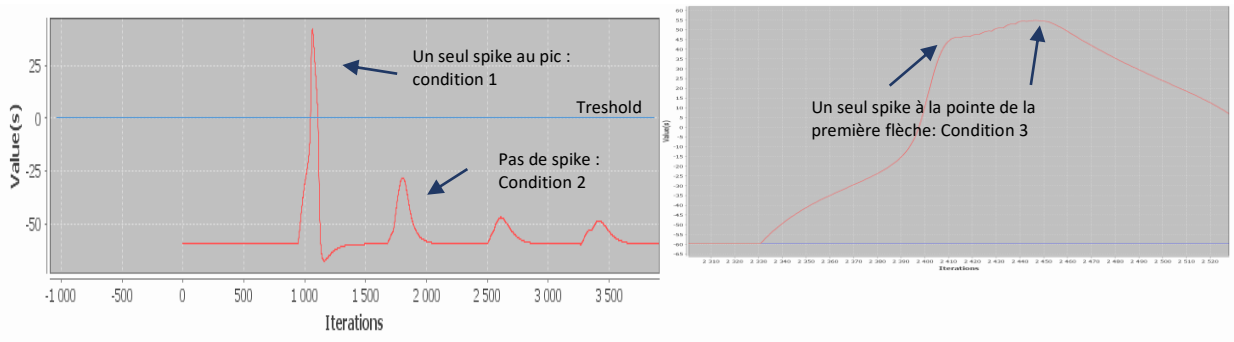


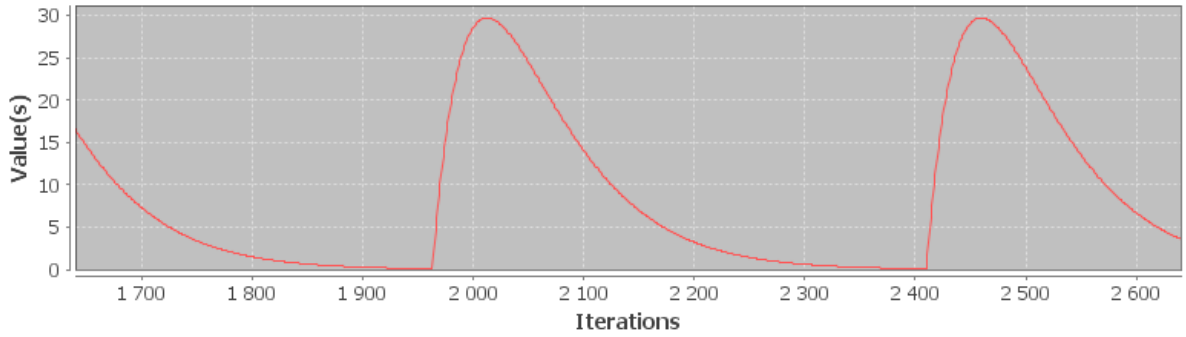
Illustration des nouvelles conditions de spike de Simbrain

ii. Modification 2 – Spike responder: Rise and Decay

Adresse: /src/org/simbrain/network/synapse_update_rules/spikeresponders/RiseAndDecay.java

Présentation de la modification

L'excitation en courant d'une synapse vers un postneurone qui correspond le mieux à l'excitation qu'on peut trouver en biologie est le spike responder de *Simbrain* nommé « Rise and Decay ».



Tracé courant rise and decay

Portion de code modifiée

Une faute de frappe dans la fonction l'empêchait d'atteindre le maximum définit:

```
public void update(Synapse s) {
     double timeStep = s.getParentNetwork().getTimeStep();
     if (s.getSource().isSpike()) {
         recovery = 1;
     }


     recovery += ((timeStep / timeConstant) * (-recovery));
     value += ((timeStep / timeConstant) * ((Math.E * maximumResponse
             * recovery * (1 - value)) - value));
     s.setPsr(value * s.getStrength());
 }
```

Extrait de mon rapport de stage consacré aux « spike responders »

b. *Spike Responder*

Comme nous l'avons dit dans le paragraphe concernant l' « input type » (II.2.a.iv), l' « Input type » dit « SYNAPTIC » met en jeu des courants synaptiques générés par les potentiels d'action du préneurone. En biologie, ce courant résulte des flux de neurotransmetteurs et des courants ioniques induits dans la zone d'échange de la synapse par le potentiel d'action. Dans la modélisation, nous ignorons complètement ces effets microscopiques et nous choisissons une fonction de courant d'excitation disponible parmi les « spike responders ». Pour les besoins de l'IRCICA, je ne présenterai que deux d'entre eux : « Rise and Decay » et « Jump and Decay ».

Rappel : le courant d'excitation que reçoit un neurone d' « input type » « SYNAPTIC » est à chaque instant la somme des produits entre la valeur du courant d'excitation de chacune de ses synapses et le coefficient « poids synaptique » correspondant.

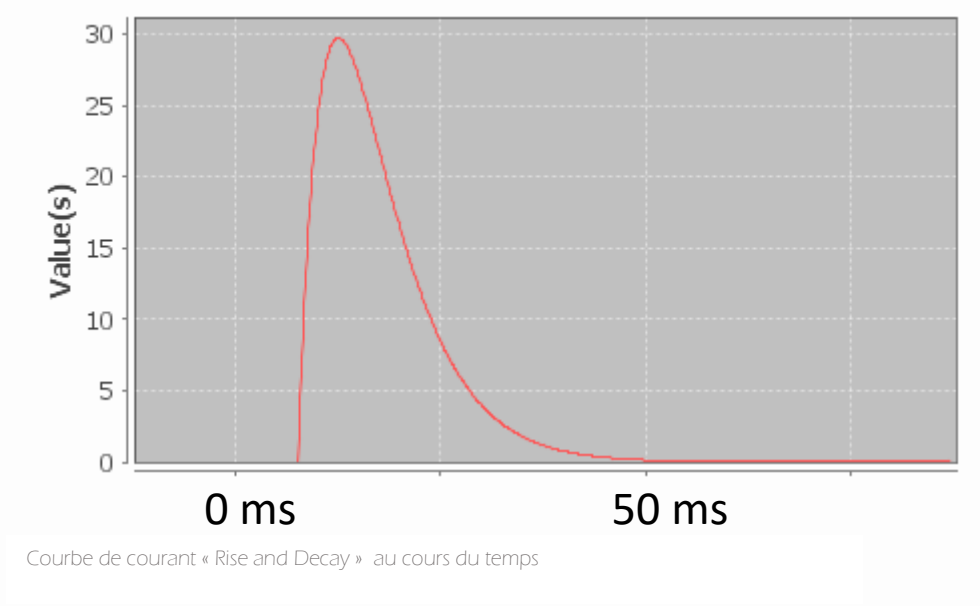
i. "Rise and Decay"

Le "spike responder" nommé "Rise and Decay" est le courant synaptique généré par un potentiel d'action similaire à ce qu'on peut observer dans la biologie. Par conséquent, le neurone artificiel et sa modélisation génèrent ce type de courant synaptique.

Ce courant correspond à la fonction suivante :

$$f(x) = \frac{bxe^{\left(-\frac{x}{a}\right)}}{a} \qquad (14)$$

Tel que f atteint son maximum b pour x = a. On peut ainsi jouer sur les deux paramètres a et b pour faire varier la constante de temps et la valeur du maximum de courant. La figure suivant représente la courbe du courant « Rise and Decay » pour a = 5 ms et b = 30mV



Courbe de courant « Rise and Decay » au cours du temps

<u>*La constante de temps*</u>

Selon les observations biologiques, cette fonction s'étale sur un temps environ 10 fois supérieur à la durée du potentiel d'action qui l'a générée. Mais j'ai d'abord adapté mon modèle à ce que le neurone artificiel était capable de faire, et initialement, il s'étalait sur la durée d'un potentiel d'action. Ce n'est que plus tard qu'il a été possible de modifier cette durée sur le neurone artificiel et il reviendra donc aux prochaines personnes qui utiliseront l'outil Simbrain d'étudier les réseaux en intégrant cette nouvelle durée.

<u>*Le courant d'excitation maximal*</u>

La fonction du « Rise and Decay » a une forme de cloche et elle atteint donc un maximum de courant d'excitation. Il est possible de choisir la valeur de ce maximum. Il faut veiller à ce que ce maximum ne soit pas trop grand afin qu'un seul potentiel d'action ne soit pas capable d'exciter un postneurone à lui seul. Il

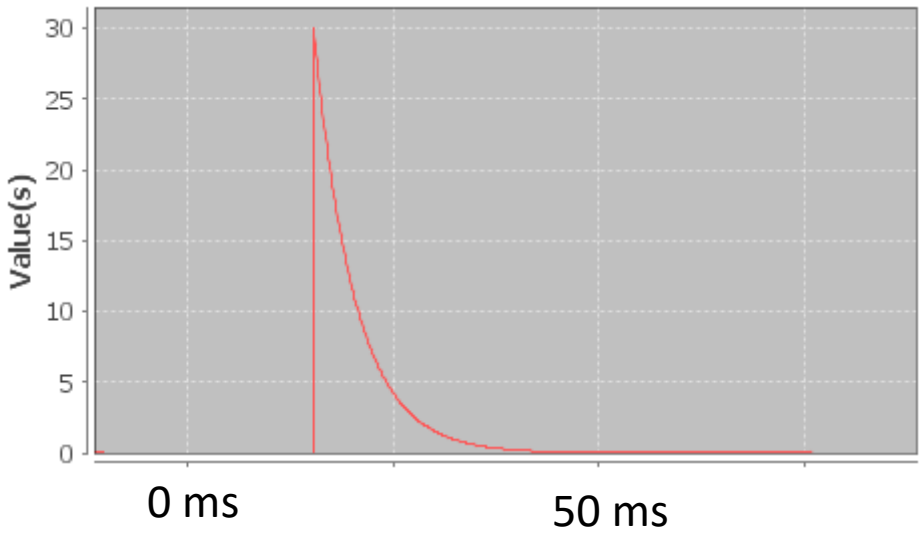
ne faut pas non plus qu'il soit trop faible sinon il est nécessaire qu'un trop grand nombre de neurones spikent presque simultanément pour provoquer un potentiel d'action chez le postneurone.

Une faute de frappe dans la fonction de Simbrain empêchait le courant d'atteindre son maximum. Cette correction fait partie des modifications que j'ai apportées à l'outil Simbrain. Elle était nécessaire car la valeur exacte de ce maximum est liée au nombre de potentiels d'action quasi-simultanés nécessaires pour exciter un postneurone.

ii. *"Jump and Decay"*

Le « spike responder » nommé « Jump and Decay » délivre au postneurone un courant constitué d'un « step », puis d'une décroissance exponentielle. J'ai utilisé ce type de « spike respondeur » afin d'effectuer de l'échantillonnage de potentiels d'action dans le cadre de la modélisation de la rétine artificielle. Cet échantillonnage permet à la rétine de percevoir des effets de mouvement dans l'image observée et est présenté plus en détail dans la partie correspondante.

Il est possible de jouer ici aussi sur la valeur maximale atteinte, soit la hauteur du « step » et sur la constante de temps de la décroissance exponentielle. La figure suivante représente la courbe de courant « Jump and Decay » pour une constante de temps égale à 5ms et un saut égal à 30mV.



Courbe de courant « Jump and Decay » au cours du temps

iii. Modification 3 – Synapse Update Rule: STDP

Adresse : /src/org/simbrain/network/synapse_update_rules/STDPRule.java

Présentation de la modification

La STDP est le mécanisme des synapses plastiques.

Avant modification, la STDP diminuait le poids d'une synapse inhibitrice pour la renforcer et la rendre plus inhibitrice. La STDP utilisée pour le neurone artificiel ne différencie pas les synapses inhibitrices et excitatrices, et toute synapse renforcée doit voir son poids augmenter.

Portion de code modifiée

Un simple changement de signe était nécessaire dans le programme :

```
public void update(Synapse synapse) {
    if (synapse.getSource().isSpike() || synapse.getTarget().isSpike()) {
        try {
            final double str = synapse.getStrength();
            final double delta_t = ((((SpikingNeuronUpdateRule) synapse
                    .getSource().getUpdateRule()).getLastSpikeTime())
                    - ((SpikingNeuronUpdateRule) synapse
                    .getTarget().getUpdateRule()).getLastSpikeTime())
                    * (hebbian ? 1 : -1);    // Reverse time window for
                                             // anti-hebbian
            if (delta_t < 0) {
                delta_w = W_plus * Math.exp(delta_t / tau_plus)
                        * learningRate;
            } else if (delta_t > 0) {
                delta_w = -W_minus * Math.exp(-delta_t / tau_minus)
                        * learningRate;
            }
              if(Math.signum(str) == -1) {
                  synapse.setStrength(str - delta_w);
              } else {
                  synapse.setStrength(str + delta_w);
               }
            synapse.setStrength(str + delta_w);
        } catch (ClassCastException cce) {
            cce.printStackTrace();
            System.out.println("Don't use non-spiking neurons with STDP!");
        }
    }
}
```

Extrait de mon rapport de stage consacré à la STDP

a. Update Rule : STDP

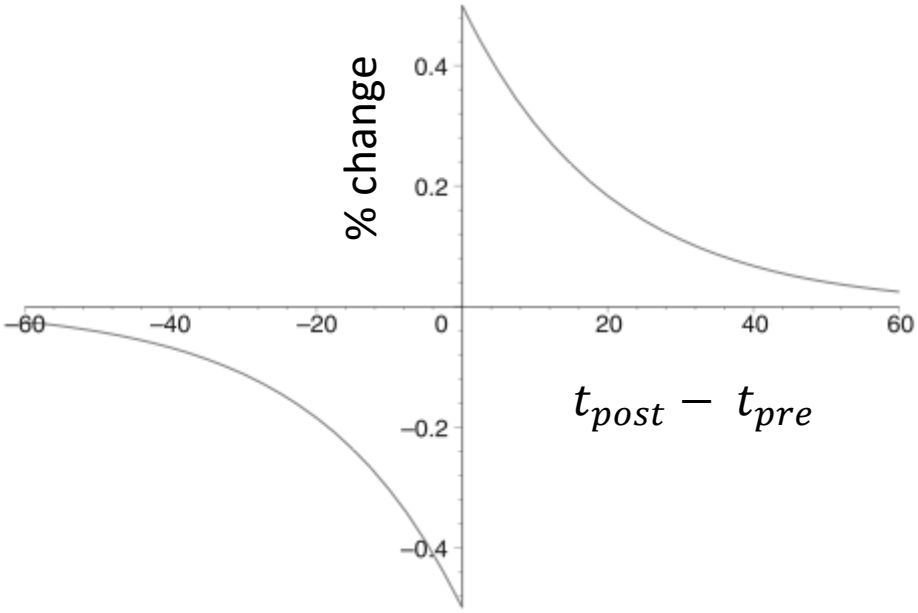
Les synapses peuvent être plastiques ou statiques dans la modélisation, dans le vivant, cette distinction est plus floue. Une synapse statique a un poids constant. Une synapse plastique peut modifier son poids et c'est la base des fonctions d'apprentissage. Il existe différentes façons de modéliser la plasticité synaptique. Celles accessibles dans Simbrain le sont sous l'onglet « update rule » de la synapse. Nous ne présenterons pas les différents apprentissages synaptiques possibles mais nous nous intéresserons seulement à la « Spike Timing Dependant Plasticity » (STDP). C'est la plasticité synaptique la plus connue et c'est aussi celle utilisée pour le neurone artificiel.

i. Principe

Le principe est le suivant : le poids synaptique varie lorsque les neurones en jeu spikent tous les deux sur un court intervalle de temps. Si le postneurone spike peu de temps après le préneurone, alors il y a un lien de causalité et le poids synaptique augmente. Si au contraire le postneurone spike peu de temps avant le préneurone, alors il n'y a pas de lien de causalité et le poids synaptique diminue. Plus le temps séparant les deux potentiels d'action est court, plus la modification du poids sera importante.

Ce principe mime la biologie où les événements impliquant une augmentation de poids synaptique vont pousser les dendrites à créer des excroissances au niveau de leur membrane en direction de l'axone qui les aura excitées. Et ce jusqu'à ce que axone et dendrite soient quasiment en contact – poids synaptique maximal – pour que l'information se propage plus rapidement.

Dans Simbrain, on peut choisir la valeur de chacun des paramètres et ainsi influer sur les temps caractéristiques et les taux de modification du poids. La figure suivante présente la courbe de la variation du poids synaptique en fonction du temps $\Delta t = t_{post} - t_{pre}$ où t_{pre} et t_{post} sont les instants respectifs de spike du préneurone et du postneurone.



Pourcentage de variation du poids synaptique selon Δt

ii. Contexte Neurone Artificiel

Concernant le neurone artificiel, il est difficile de créer un poids synaptique variable à l'aide de composants électroniques. La solution adoptée par l'équipe « hardware » est de séparer l'apprentissage en deux phases distinctes, l'apprentissage à court terme et l'apprentissage à long terme. En effet, si l'information du poids synaptique est mise en mémoire de manière analogique, c'est-à-dire sous forme de potentiel dans un condensateur, alors ce condensateur imparfait se déchargera naturellement et l'information sera perdue au bout de quelques secondes. Cette solution n'est pas envisageable pour un apprentissage à long terme. Si au contraire l'information est stockée de manière numérique, alors il sera plus difficile de la faire varier en temps réel.

Pour l'apprentissage à court terme, le poids synaptique est gardé en mémoire dans la charge d'un condensateur à décharge lente. Il va se charger ou se décharger en fonction de l'ordre des potentiels d'action du préneurone et du postneurone, selon le principe de la STDP. Le condensateur est totalement chargé dans le sens positif ou négatif seulement au bout de plusieurs potentiels d'action. Ce type d'apprentissage est à court terme car lorsqu'aucun événement n'a lieu, le condensateur se décharge naturellement.

C'est là que l'apprentissage à long terme entre en jeu. Celui-ci correspond à un interrupteur à trois positions qu'on notera « position 1 », « position 0 » et « position -1 ». Une synapse neutre est en « position 0 ». Lorsque le condensateur atteint sa charge maximale dans le sens positif, l'interrupteur passe en « position 1 » pour une synapse excitatrice. Il atteint la « position -1 » dans l'autre sens, le sens de l'inhibition. Contrairement à l'apprentissage à court terme, une fois que ces états sont atteints ils sont stables, et plusieurs événements de charge du condensateur dans un sens possible sont nécessaires afin de changer la valeur de l'interrupteur.

Cet apprentissage similaire à la STDP, nous ne cherchons pas à le modéliser sur Simbrain. En effet cette solution se veut être un modèle de STDP convenable. La comparaison entre la simulation et l'expérimentation réelle permettra de déterminer à quel point ce modèle est proche de celui de la STDP théorique. Cette façon de tester le modèle physique à partir des résultats de la simulation se nomme « rétro simulation ». C'est une composante essentielle à tout processus de création scientifique, et elle est aussi utile que la simulation pour faire avancer la connaissance et comprendre les mécanismes.

iii. Modification du script

Une fois de plus, le code du logiciel Simbrain n'était pas satisfaisant. Initialement, la STDP différenciait les synapses inhibitrices et excitatrices pour la modification de poids, et le renforcement d'une synapse inhibitrice correspondait donc à une diminution de poids. Ce n'est pas le comportement voulu, mais une simple modification de signe dans le programme Simbrain a arrangé la situation.

iv. Modification 4 – Synapse : confort de creation de synapses

Adresse : /src/org/simbrain/network/core/Synapse.java

Présentation de la modification

Cet ajout de fonction est un exemple de toutes les créations d'objets *Simbrain* possibles pour gagner du temps dans la modélisation des réseaux. Ici j'avais besoin d'entrer directement ces paramètres lors de la création d'une synapse pour ne pas avoir à la modifier ensuite.

Portion de code modifiée

Cette modification est une simple modification de confort de programmation, non nécessaire.

```
/**
 * Construct a synapse using a source and target neuron, and a specified
 * learning rule. Assumes the parent network is the same as the parent
 * network of the provided source neuron.
 *
 * @param newParent new parent network for this synapse. Used when copying
 *            and pasting to new network.
 * @param source source neuron
 * @param target target neuron
 * @param learningRule update rule for this synapse
 * @param templateSynapse synapse with parameters to copy
 * @param initialStrength initial strength for synapse
 */
public Synapse(Network newParent, Neuron source, Neuron target,
                SynapseUpdateRule learningRule, Synapse templateSynapse,
double initialStrength ) {
    this(templateSynapse); // invoke the copy constructor
    setSourceAndTarget(source, target);
    initSpikeResponder();
    setLearningRule(learningRule);
    parentNetwork = newParent;
    this.forceSetStrength(initialStrength);
```

Mon rapport de stage d'évoque pas cette modification.

v. Modification 5 – Weighted

Adresse : /src/org/simbrain/network/core/Neuron.java

Présentation de la modification

La solution technique proposée concernant la détection de mouvement dans la rétine artificielle implique un produit des courants d'entrée d'un neurone. La simulation devait pouvoir reproduire ce produit. A défaut de savoir créer un nouvel ***input type*** sans dérégler tout le logiciel, j'ai choisi de modifier l' ***input type WEIGHTED*** dont je n'ai jamais eu besoin car les réseaux de l'IRCICA sont essentiellement des réseaux à spike.

Portion de code modifiée

```
    public double getWeightedInputs() {
//          double wtdSum = inputValue;
//          for (int i = 0, n = fanIn.size(); i < n; i++) {
//           wtdSum += fanIn.get(i).calcWeightedSum();
//          }

          double wtdSum = inputValue;
          double a = 1;
          for (int i = 0, n = fanIn.size(); i < n; i++) {
             a *= fanIn.get(i).calcWeightedSum();
          }
          wtdSum += a;
          return wtdSum;
    }
```

Extrait de mon rapport de stage consacré aux ***input types***

i. *« Input type » : type d'entrée reçue par le neurone*

Avant de continuer, il faut savoir que les synapses sont orientées et ne délivrent le courant que dans un sens. Dans la biologie, la synapse est la connexion entre l'axone d'un neurone et la dendrite d'un autre, le courant ne peut aller que de l'axone vers la dendrite. Le logiciel *Simbrain* considère les neurones comme des éléments ponctuels, les axones et dendrites n'y existent donc pas. Dans la suite on appelle préneurone le neurone qui envoie un signal électrique et postneurone celui qui le reçoit.

L' « input type » détermine quel type de courant un postneurone va recevoir de ses synapses, il peut être « WEIGHTED » ou « SYNAPTIC ».

« SYNAPTIC »

Le cas « SYNAPTIC » correspond à la biologie car tant qu'il n'y a pas de potentiels d'action, les neurones sont complètement isolés électriquement. Nous l'utilisons dans la plupart des applications du neurone

artificiel. Dans ce cas, le postneurone ne reçoit un courant synaptique que si le préneurone spike – atteint un pic de potentiel. Ce courant sera présenté explicitement dans la partie « spike responders ».

Le courant d'excitation I_{ex} du postneurone i est alors la somme des produits des poids synaptiques w_{ij} des neurones j vers i par les courants d'excitation générés par les potentiels d'action des préneurones. Soit:

$$I_{ex}(i) = \sum_j w_{ij} * i_{SP}(j) \quad (1)$$

"WEIGHTED »

Dans le cas « WEIGHTED », le postneurone va simplement recevoir un courant d'excitation égal à la somme des produits des poids synaptiques w_{ij} par l'activation des préneurones j dont il reçoit le signal. Soit

$$I_{ex}(i) = \sum_j w_{ij} * activation(j) \quad (2)$$

Comme le type « WEIGHTED » n'avait aucune utilité dans les réseaux simulés, j'ai modifié cet « input type » pour créer le produit de courants synaptiques nécessaire à la reconnaissance de mouvements de la rétine artificielle. En effet, il était plus simple pour moi de modifier un morceau de code que d'en insérer un nouveau et en gérer toutes les dépendances. Cette modification est présentée dans la partie sur la rétine artificielle.

vi. Limites et améliorations en attente

Les améliorations que j'ai effectuées concernent principalement la mise à jour des neurones et les fonctions mathématiques sous-jacentes. Il reste beaucoup de possibilités, en voici quelques-unes :

- La simulation des réseaux de neurones est bien plus rapide que celle des circuits électroniques. Un ajout de parallélisme dans les calculs pourrait encore augmenter sa rapidité. Le créateur de *Simbrain* a déjà implémenté un outil de gestion de la mise à jour des réseaux mais je n'ai pas su l'exploiter.

- Une fois que les paramètres idéaux des simulations numériques auront été découverts pour le neurone Biologique et le neurone Fast, il faudra modifier les paramètres par défaut du neurone de Morris-Lecar. De même, les paramètres par défaut des fonctions ***Rise and Decay***, ***Jump and Decay*** ou encore ***STDP*** peuvent aussi être ajustés par rapport au neurone Simbrain.

- Il est possible de créer des fonctions génératrices d'objets avec des paramètres spécifiques sur le modèle de la modification 4.

- L'une des contraintes intrinsèques de *Simbrain* concerne le tracé des courbes, nommées ***times series***. On ne peut tracer que les ***activations*** des neurones et les poids synaptiques, et ces tracés sont ralentissent beaucoup les réseaux. Lorsque j'ai utilisé le langage Python pour tracer plus rapidement le potentiel des neurones Morris-Lecar, j'ai été impressionné par la différence de temps. C'est pour cela que je suis persuadé que les performances de traçage des courbes peuvent être améliorées en évitant les calculs inutiles

- Il peut être possible de se débarrasser du spike de début de simulation en imposant les valeurs de repos du neurone comme valeurs initiales pour ***vM*** et ***n***.

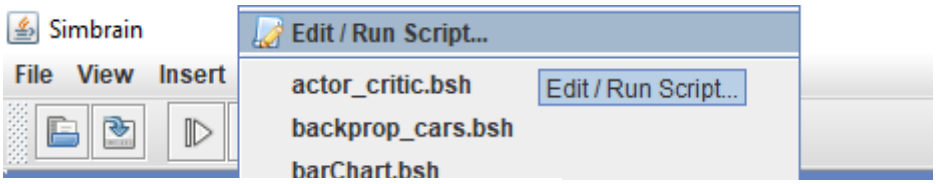
3. Présentation des réseaux « IRCICA »

Voici une présentation des différents réseaux développés pendant mon stage. Ils peuvent être utilisés ou simplement servir de modèles pour créer d'autres réseaux. Ces réseaux sont compatibles avec les modifications présentées dans la partie précédente et risquent de ne pas fonctionner sur une version de *Simbrain* non adaptée.

a. Généralités sur les réseaux

i. Les scripts 'IRCICA_'

Tous les réseaux que j'ai développés portent un nom qui commence par IRCICA_. Pour les ouvrir depuis *Simbrain* il suffit de cliquer sur ***Scripts*** puis ***edit / run script*** et d'aller les trouver dans l'ordinateur, ils sont disponibles en annexe. Les réseaux doivent être rangés dans le dossier ***scripts/scriptmenu*** du dossier de *Simbrain* et porter l'extension '.bsh' pour apparaître dans la liste défilante.



Scripts, edit / run script …

ii. Structure des scripts

Tous les scripts 'IRCICA_' ont des similitudes et une structure semblable. Les neurones y sont nommés selon la couche puis numérotés afin d'être accessibles facilement pour le tracé des courbes ou autre.

Les paramètres du réseau sont tous entrés en dehors du `main()`. Cela permet de ne modifier qu'une ligne pour toutes les fois où le paramètre est utilisé.

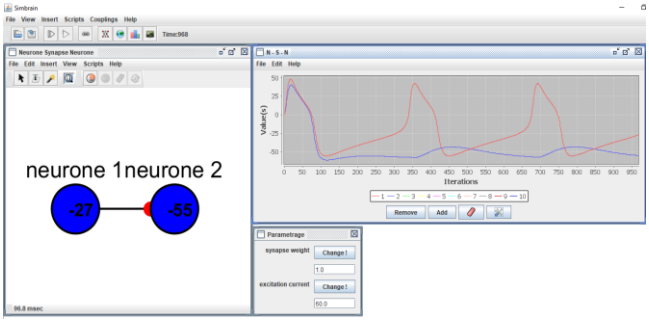
L' ***update rule*** baptisée ***generalRule*** paramètre le modèle de Morris-Lecar du réseau. Chaque paramètre non modifié garde sa valeur par défaut.

L' ***update rule*** baptisée ***pixel*** correspond aux sources de courants.

Les ***time series*** permettent de tracer des valeurs d'intérêt et les ***buttons*** permettent eux d'effectuer facilement certaines actions utiles sur le réseau depuis l'interface.

b. IRCICA_N_S_N

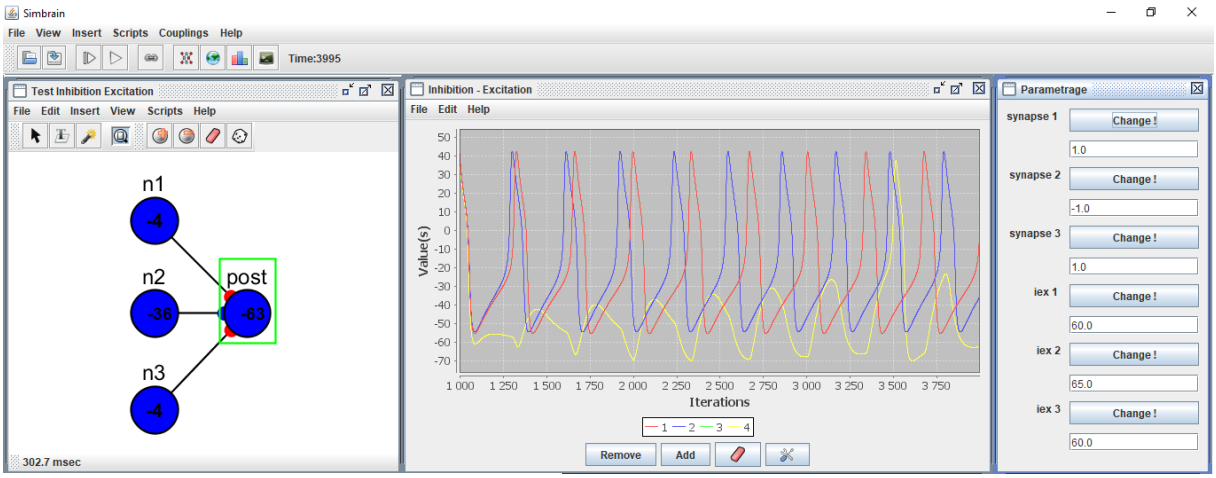
Le réseau *IRCICA_N_S_N* pour « neurone-synapse-neurone » est constitué de deux neurones de Morris-Lecar. Le ***time series*** a pour ***entrée 1*** le potentiel membranaire du ***neurone 1*** et en ***entrée 2*** celui du ***neurone 2***. Les boutons permettent de modifier le poids de la synapse ainsi que la valeur du courant d'excitation du ***neurone 1***.



Workspace N_S_N

c. IRCICA_TestInhibition

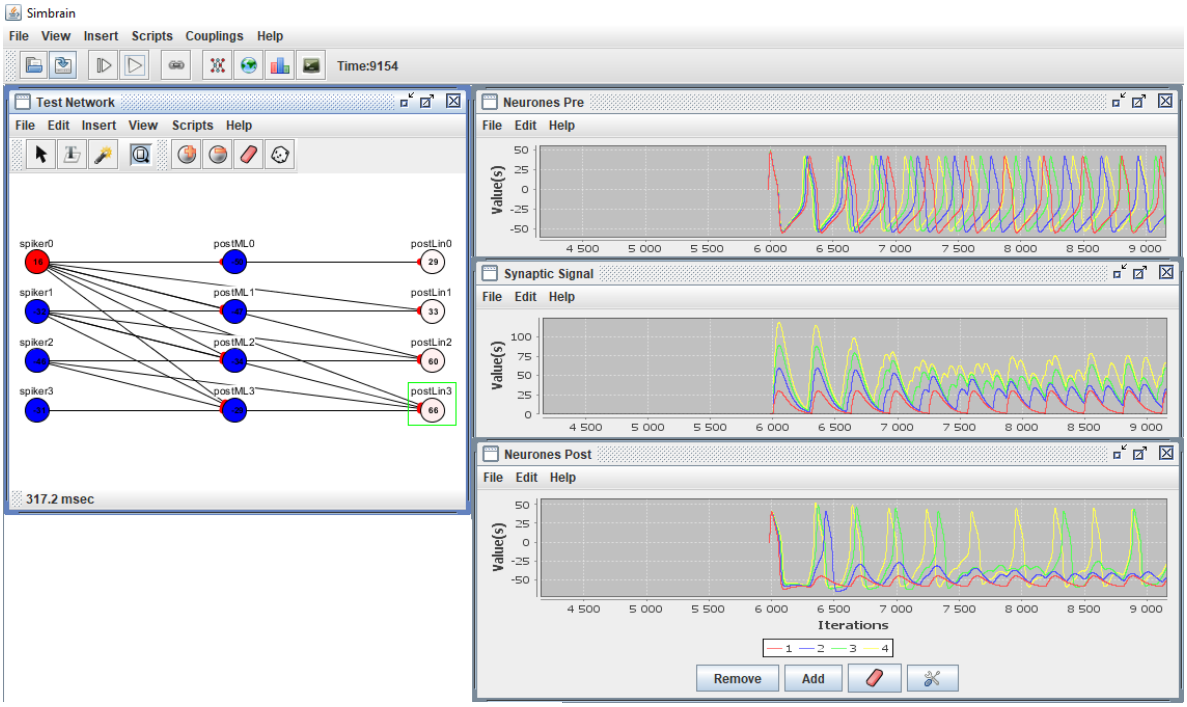
Le réseau *IRCICA_TestInhibition* sert à étudier des synapses inhibitrices et excitatrices. Les trois premières entrées du graphe tracent respectivement l'activation de ***n1***, ***n2*** et ***n3***. L'***entrée 4*** trace celle du postneurone. Les boutons permettent de modifier le poids de chaque synapse indépendamment et les courants d'excitation des neurones ***n1***, ***n2*** et ***n3***.



Workspace Testinhibition

d. IRCICA_RiseAndDecayTest

Ce réseau sert à tester le « spike responder » « Rise and Decay » et à le calibrer. Il est composé de cinq neurones de Morris_Lecar ***spikers_***, ceux-ci sont excités par des courant différents et sont donc déphasés. Cinq neurones de Morris_Lecar ***postML_*** sont reliés respectivement à 1, 2, 3, 4 ou 5 ***spikers_***. Cinq « pixels » ***postLin_***sont connectés de la même façon que les ***postML_*** et sont les témoins des courants synaptiques qu'ils reçoivent. Trois courbes sont tracées, la courbe ***neurones Pre*** trace l' « activation » des ***spikers_***, la courbe ***Synaptic Signal*** trace l' « activation des ***postLin_*** et donc les courants synaptiques envoyés par les ***spikers_,*** enfin, la courbe ***neurones Post*** trace l' « activation » des neurones ***postML_***. Nous pouvons donc observer l'influence des courants synaptiques émis par les ***spikers_*** sur les ***postML.***



Workspace RiseandDecaytest

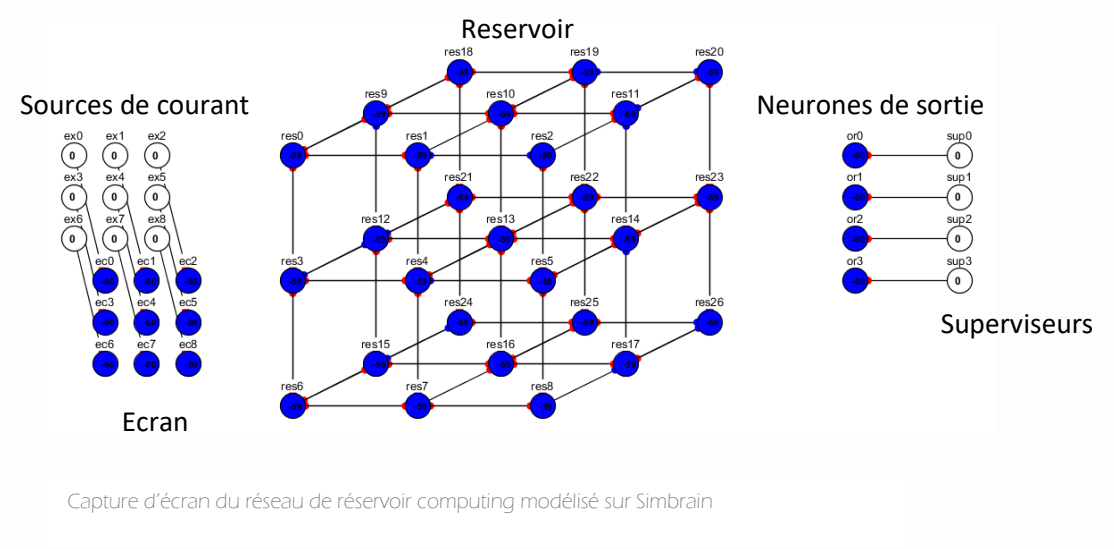
e. IRCICA_ReservoirComputing et IRCICA_ReservoirComputingSpherique

Le réseau *IRCICA_ReservoirComputing* et *IRCICA_ReservoirComputingSpherique* correspond au réseau présent sur la « chip 3 » et développé pour tester les mécanismes de la STDP ainsi que le fonctionnement du neurone artificiel en réseau. Une fois fabriqué, nous pourrons comparer les simulations à la réalité.

i. Extrait du rapport de stage dédié au réseau de réservoir computing

1. Reservoir computing

Le réseau nommé reservoir computing sera le premier réseau de neurones artificiels fabriqué par l'IRCICA. Les reservoir computing sont une famille de réseaux de neurones non bioinspirés pensés pour l'apprentissage. Un signal d'entrée est transmis à un réseau aléatoire dynamique de neurone appelé réservoir qui aura pour fonction d'augmenter la dimension de ce signal. Les neurones de sortie sont ensuite entrainés à lire les signaux de plus haute dimension sortant du réservoir. Voici une capture d'écran du modèle de réservoir que j'ai développé, sans les connexions entre les différentes couches pour plus de clarté :



Capture d'écran du réseau de réservoir computing modélisé sur Simbrain

Ce réseau au fonctionnement basique est fabriqué dans le but de tester le fonctionnement des neurones artificiels en réseau. Les membres de l'équipe Hardware ayant développé ce réseau savent qu'ils ont peu de chances d'obtenir un apprentissage satisfaisant pour plusieurs raisons. D'une part le réservoir de 27 neurones est relativement petit et cela n'assure pas d'augmenter assez la dimension du signal d'entrée pour l'apprentissage. De plus, les paramètres du réseau présentés dans le paragraphe suivant n'ont pas été optimisés, en effet, les logiciels de simulation de circuits électriques étaient incapables de simuler le comportement d'un tel réseau. Mais les attentes concernant ce réseau ne sont pas de cet ordre, il aura pour principal objectif de tester l'outil *Simbrain* en comparant les comportements réels et simulés du même réseau. Ainsi, les délais dans sa fabrication m'ont permis de chercher à optimiser les paramètres 'a' et 'b' par sa modélisation et sa simulation dans *Simbrain*, et cette modélisation sera déjà prête lorsqu'il sera temps de la comparer avec le réseau réel.

a. Présentation du réseau de reservoir computing de l'IRCICA

Le reservoir computing développé par l'IRCICA est un réseau simple pensé pour apprendre à reconnaitre les quatre orientations, horizontale, verticale, et les deux diagonales. Tous les neurones utilisés sont des neurones de Morris-Lecar délivrant des courants d'excitations de type « Rise and Decay » et trois potentiels d'action de préneurones sont nécessaires pour provoquer un potentiel d'action de postneurone.

Le signal d'entrée est constitué de neuf neurones de Morris-Lecar composant un écran de dimensions '3 x 3' sur lequel on peut projeter les quatre orientations. Les neurones de cet écran possèdent des connexions synaptiques réparties aléatoirement dans le réservoir selon un ratio qu'on notera 'a' dont la valeur initiale vaut 20%. Cela signifie qu'un neurone de l'écran et un neurone du réservoir ont une probabilité de partager une connexion synaptique égale à 'a'.

Le réservoir est composé de 27 neurones de Morris-Lecar représentant un cube de trois neurones de côté. Chaque neurone de ce cube est connecté à ses plus proches voisins, ces connexions peuvent être excitatrices ou inhibitrice. Le pourcentage de connexions inhibitrice est noté 'b' est a pour valeur initiale 20%. Ainsi, le nombre de voisins d'un neurone du réservoir dépend de sa position dans le cube, cette diversité asymétrique ajoute de l'efficacité au réseau.

Chaque neurone du réservoir est connecté aux quatre neurones de Morris-Lecar de sortie par des synapses plastiques de poids initialement nul. L'apprentissage aura pour but de faire converger leurs poids vers les valeurs 1 pour l'excitation et -1 pour l'inhibition. Dans la pratique, les poids synaptiques adoptent des valeurs intermédiaires pendant l'apprentissage avant de converger vers -1 et 1.

b. Résultats concernant le réseau

Déroulement de la simulation d'apprentissage

Après l'avoir modélisé, j'ai simulé le comportement du reservoir computing. J'ai observé le comportement du réservoir et programmé un apprentissage automatique dans le logiciel *Simbrain.* Le fonctionnement du réseau est le suivant :

L'expérimentateur excite les neurones de l'écran pour y faire apparaître une des quatre orientations. Les courants synaptiques générés par les potentiels d'action des neurones de l'écran enclenchent une réaction en chaine dans le réservoir et ses neurones se mettent alors à spiker dans un ordre dépendant des paramètres 'a' et 'b' et du tirage aléatoire des connexions synaptiques et de leur état excitateur ou inhibiteur. Les courants synaptiques des potentiels d'action du réservoir sont tous perçus par les neurones de sortie.
L'expérimentateur excite le neurone destiné à être sensible à l'orientation apparue à l'écran quelques instants après avoir excité l'écran. Cette excitation est brève et ne provoque qu'un potentiel d'action du neurone de sortie, cela enclenche le mécanisme d'apprentissage nommé STDP (cf. II.3.c) sur les synapses reliant le réservoir à ce neurone. De ce fait, les neurones du réservoir ayant spiké avant ce potentiel d'action du neurone de sortie vont voir le poids de leur connexion synaptique avec celui-ci augmenter. Et dans le cas où la réaction en chaîne n'est pas terminée, les neurones qui spikent après le neurone de sortie vont voir le poids de leur connexion synaptique avec celui-ci diminuer.

Valeurs des paramètres

Selon les valeurs des paramètres 'a' et 'b', la réaction en chaine du réservoir est plus ou moins longue. Si le taux de connexion entre l'entrée et le réservoir est trop élevé, c'est-à-dire si la valeur de 'a' est trop élevée, alors les signaux d'entrée vont surexciter le réservoir et ses neurones vont spiker très longtemps, voir indéfiniment. Cet effet peut être en partie compensé par un fort taux d'inhibition dans le réservoir, obtenu avec une haute valeur de 'b'. Si au contraire le taux de connexion entre l'entrée et le réservoir est trop bas, alors les signaux d'entrée ne vont pas assez exciter le réservoir et peu de ses neurones vont spiker, parfois aucun. Cet effet sera relativement compensé par un fort taux d'excitation dans le réservoir, et donc une valeur basse de 'b'. Il est donc possible de trouver un équilibre optimal entre les valeurs de 'a' et 'b'.

Si 'a' ou 'b' est trop proche de 0% ou de 100%, le réservoir est dysfonctionnel, c'est pourquoi les valeurs optimales sont situées dans une zone intermédiaire. Le taux d'inhibition ne doit pas dépasser 50% sous peine de 'taire' totalement le réservoir. J'ai trouvé ces valeurs optimisées pour l'apprentissage: 'a' = 33% et 'b'= 20%. La valeur de 'a' égale à un tiers correspond aux trois potentiels d'action simultanés nécessaires pour exciter un neurone de l'écran. Elle assure qu'un certain nombre de neurones du réservoir enclencheront la réaction en chaîne. La valeur de 'b' correspond aux observations de la biologie, il y a généralement 20% d'inhibition dans les réseaux de neurones qu'on observe dans le vivant.

Ces valeurs des paramètres 'a' et 'b' n'assurent pas un apprentissage systématique. En effet les connexions liées à ces paramètres sont tirés aléatoirement à chaque simulation et il arrive que le réseau ne soit pas adapté pour l'apprentissage. Je n'ai pas pu tester tous les paramétrages possibles, en effet, la convergence des poids synaptiques est relativement longue et les simulations nécessitaient environ une demi-heure par

paramétrage. Les autres paramétrages fonctionnels que j'ai trouvés étaient proches des valeurs proposées ci-dessus.

Autres résultats

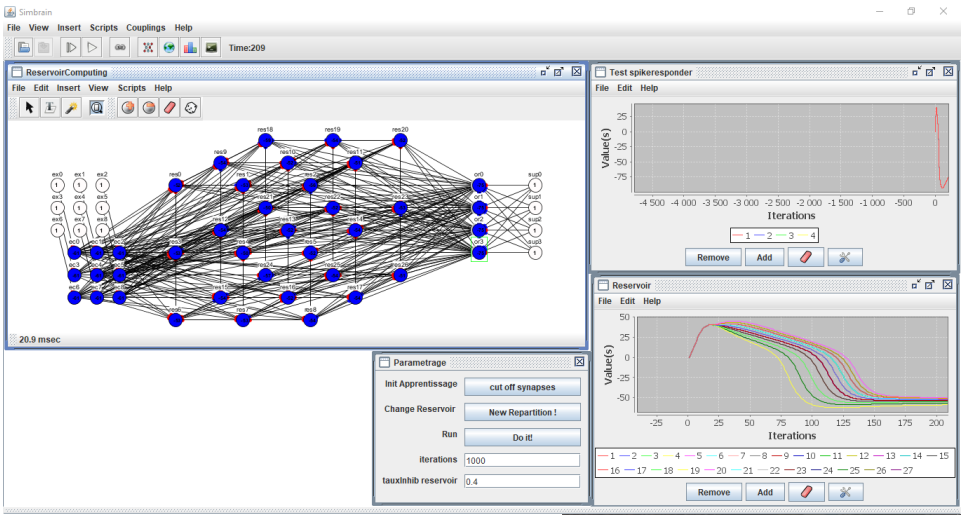
La réaction en chaîne excite souvent une grande partie du réservoir. Il a été nécessaire de modifier les paramètres des neurones de sortie pour qu'ils soient deux fois moins sensibles, car ceux-ci recevaient de trop grands courants d'excitation au terme de l'apprentissage.

J'ai tenté de rendre le réseau symétrique en créant un réservoir sphérique, c'est-à-dire en connectant chaque face à son opposée. Ainsi, tous les neurones avaient le même nombre de voisins directs. Cette modification a totalement empêché le réseau d'apprendre à reconnaître les orientations. Cela a confirmé l'utilité de l'asymétrie générée par la forme cubique.

Améliorations apportées au logiciel *Simbrain*

L'étude du reservoir computing sur les dernières semaines de mon stage a eu un impact positif sur le développement de l'outil *Simbrain*. Ce réseau plus simple que la rétine artificielle, et dont on connait déjà le comportement théorique a été l'occasion de remettre en question et d'améliorer ma compréhension du logiciel. La simulation du reservoir computing m'a d'ailleurs permise de réaliser certaines limites de ma reprogrammation. J'ai pris consciences des limites du paramétrage du neurone de Morris-Lecar que j'avais déterminé, j'ai également redéfinit les conditions de spike à partir du comportement attendu du réseau.

ii. Description du script



Workspace reservoir computing

La seule différence entre *IRCICA_ReservoirComputing* et *IRCICA_ReservoirComputingSpherique* réside dans le fait que dans le réservoir sphérique, le cube n'a pas de bords puisque tout se passe comme si chaque face était reliée à son opposée. Dans ce réseau tous les neurones sont des neurones de Morris-Lecar sauf les neurones ***sup_*** et ***ex_***.

J'ai choisi de programmer le tracé du comportement du réservoir, l'évolution des poids des synapses plastiques au cours de l'apprentissage, ainsi que le potentiel membranaire des neurones de sortie. Cependant le logiciel devient bien trop lent lorsqu'on cherche à tout tracer et c'est pour cela qu'il faut choisir quelles courbes tracer à chaque simulation.

Plusieurs boutons sont disponibles pour ce réseau. Certains de ces boutons sont codés mais doivent être décommentés dans le script pour apparaître dans la boite de dialogue.

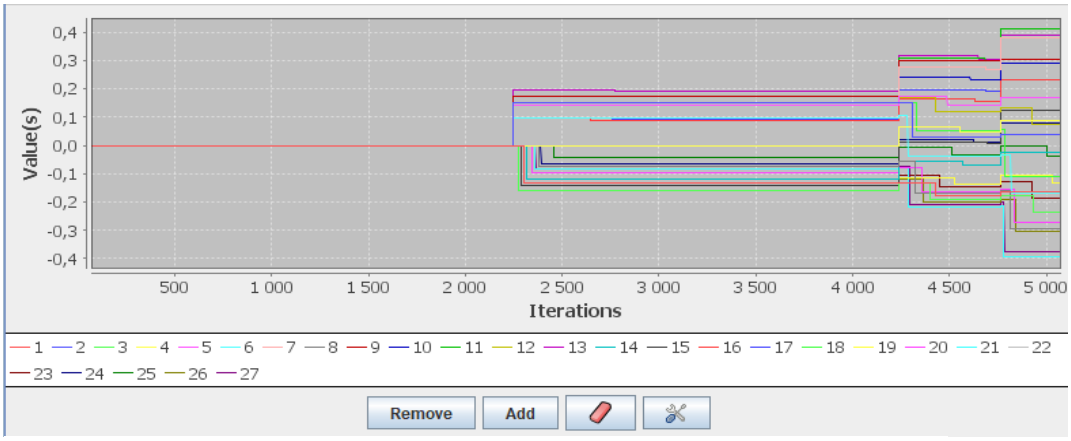
- ***Init Apprentissage*** permet de débuter un nouvel apprentissage en cliquant sur ***cut off synapse***. Cela réinitialise les poids des synapses plastiques à zéro.
- ***Finish learning*** permet de bloquer les poids synaptiques en cliquant sur ***freeze all***
- ***Restart learning*** permet de débloquer les poids synaptiques en cliquant sur ***defreeze***
- ***Change Reservoir*** permet de modifier la répartition ***inhibition/excitation*** du réservoir selon la valeur de la case ***tauxInhib reservoir***
- ***Run*** permet de lancer autant d'itérations que le nombre dans la case ***itérations***

Les paramètres du réseau :

- ***nRowEc*** correspond à la taille d'un côté de l'écran ***ec_*** sachant que cet écran est carré.
- ***nRow*** correspond à la longueur d'une arrête du cube dans le réservoir.
- ***tauxCo*** : pourcentage de connexion entre les neurones de l'écran et ceux du réservoir
- ***tauxInhib*** : pourcentage de connexions inhibitrices au sein du réservoir
- ***wectores*** : weight ***ec_*** to ***res_***, le poids synaptique des connexions ***ec_*** à ***res_***
- ***wrestores*** : même principe
- Les autres paramètres correspondent à ceux des fonctions *Simbrain* utilisées dans le réseau. Les paramètres des fonctions « Rise and Decay » diffèrent selon les neurones concernés.

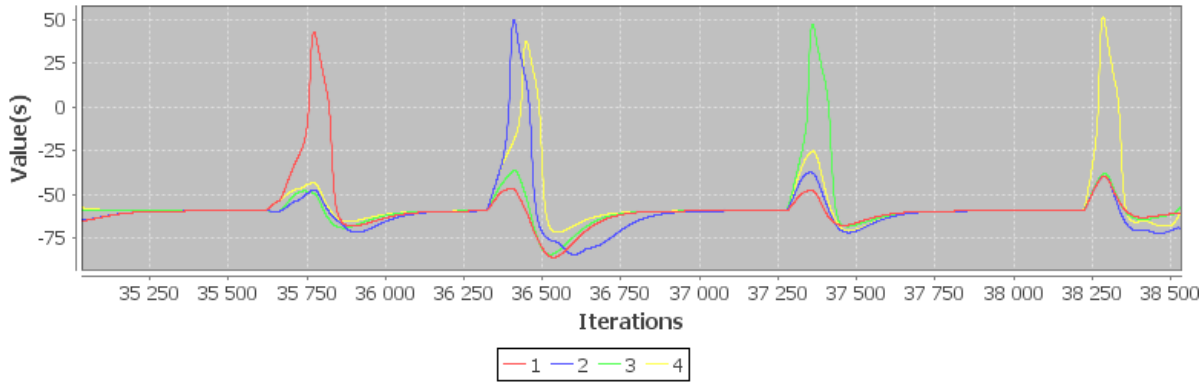
iii. Resultats en images

La courbe suivant représente les poids des synapses plastiques entre le reservoir et les neurones de sortie en début d'apprentissage. On voit que certains poids prennent des valeurs positives et d'autres des valeurs négatives. Ces poids vont converger entre les valeurs -1, 0 et 1 à la fin de la séance d'apprentissage. Pour obtenir des poids négatifs, il faut exciter le neurone de sortie avant la fin de la réaction en chaîne dans le réservoir, ainsi, des neurones spikeront après le neurone de sortie et la STDP attribuera alors un poids négatif à ces connexions.

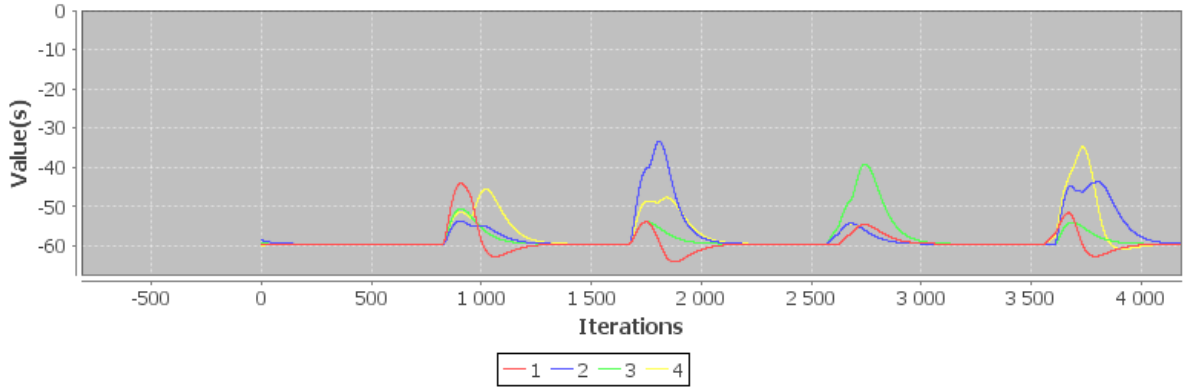


Evolution des poids des synapses plastiques reliant le reservoir à un des neurones de sortie

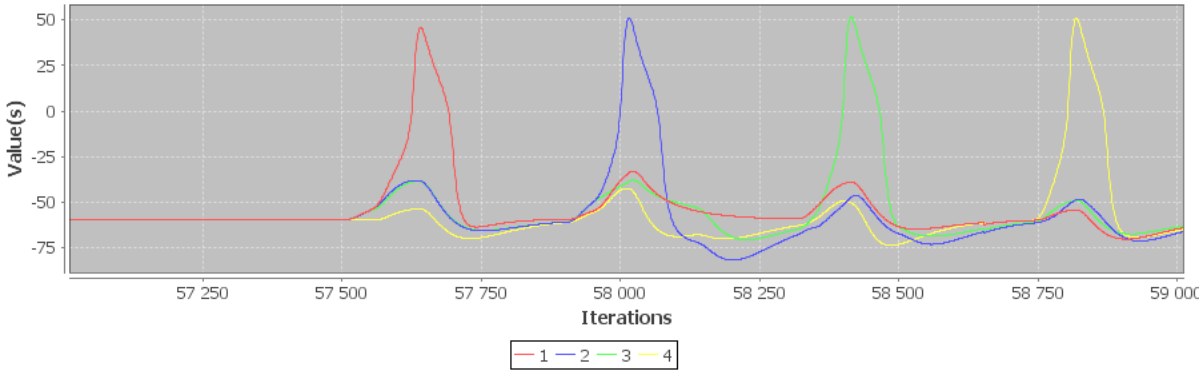
Les trois courbes suivantes montrent l'activation des quatre neurones de sortie pour des réseaux Reservoir Computing non sphériques aux paramètres différents. Ces courbes sont tracées pour vérifier l'efficacité de la session d'apprentissage : les quatre orientations sont présentées tour à tour à l'écran et on voit si cela excite la sortie indiquée. L'apprentissage est un succès si on des spikes dans cet ordre : rouge, bleu, vert, jaune.



Courbe 1 : apprentissage raté, deux neurones de sortie sont excités par la seconde orientation

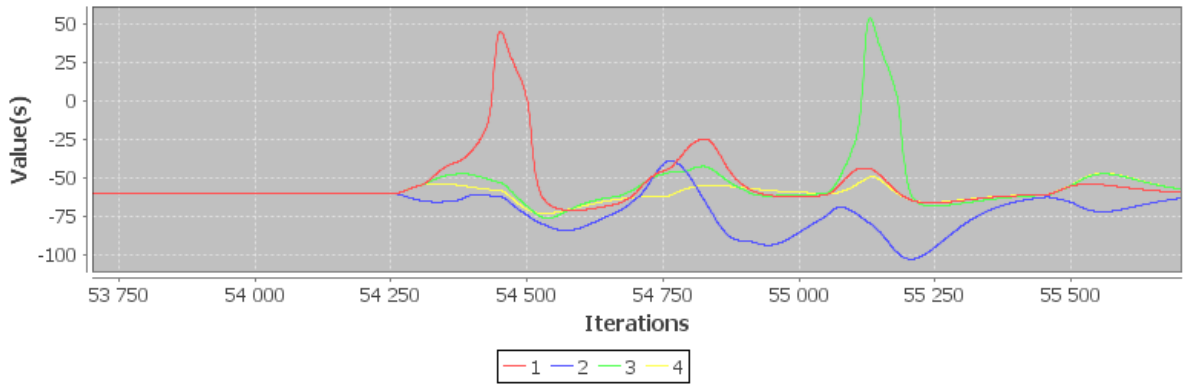


Courbe 2 : apprentissage raté, aucun neurone de sortie ne spike car les poids synaptiques n'ont pas finis de converger



Courbe 3 : apprentissage réussi

La courbe suivante montre un apprentissage non satisfaisant pour le réservoir sphérique. Le reservoir sphérique a beaucoup plus de difficultés à apprendre.



Courbe 4 : apprentissage raté, reservoir sphérique

Au fil des essais, voici les différents paramètres que j'ai fait varier :

- Les valeurs de a et b, pour finalement adopter les valeurs a=33% et b=20%
- Le taux d'apprentissage de la STDP pour finalement adopter la valeur 1/5.
- Le temps entre excitation de l'écran et excitation des neurones de sortie, pour que le neurone de sortie spike avant la fin de la réaction en chaîne.
- Le nombre de fois qu'une orientation est présentée à l'écran pour permettre l'apprentissage.
- Ajouter de l'inhibition latérale entre les neurones de sortie et leurs différents superviseurs, ce afin de limiter les apprentissages non prévus. Mais la petite taille du réservoir implique que deux orientations peuvent souvent activer des réactions en chaîne similaires dans le réservoir et donc exciter les mêmes neurones de sortie. Cette situation est illustrée sur la courbe 1 où l'orientation 'bleue' excite aussi le neurone 'jaune'.

iv. Utilisation du réseau

Pour utiliser le réseau, on peut modifier les sources de courant ***ex_*** et ***sup_*** à la main, ou de façon automatique avec une ***input table*** ou avec un ***dataworld*** (cf. 2.b.). Dans tous les cas, on allume d'abord l'orientation sur l'écran puis un certain temps après on allume le superviseur correspondant à cette l'orientation. On réitère cela plusieurs fois pour que l'apprentissage puisse converger.

v. Pistes d'étude

- Tester les couples de paramètres a et b pour connaître le meilleur apprentissage, avec plusieurs essais par jeu de paramètres pour avoir une information objective face aux aléas du tirage des connexions.

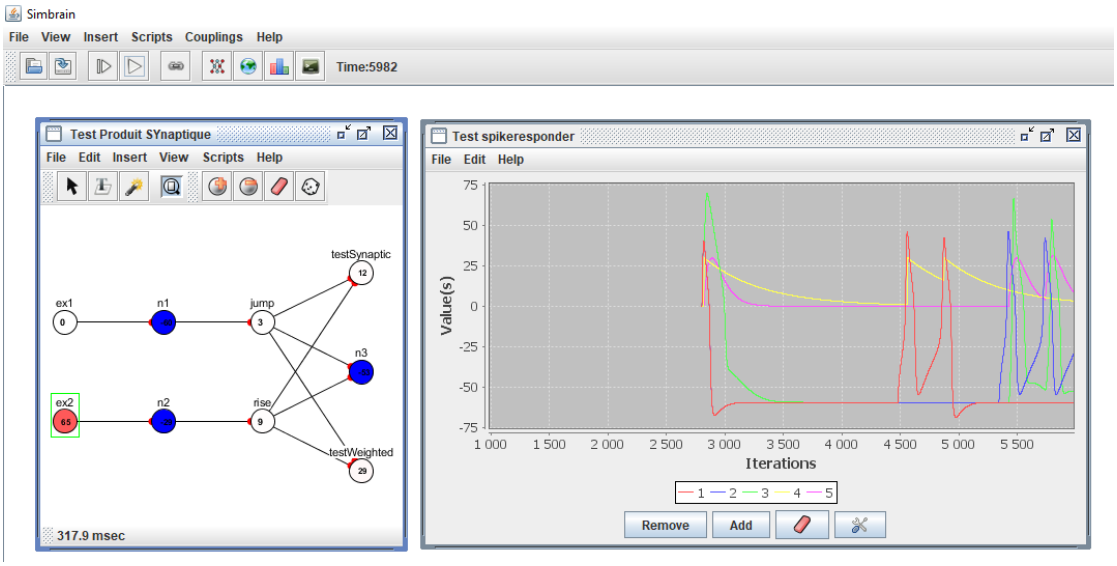
- Comprendre le fonctionnement du réservoir avec des constantes de temps de rise and decay bien plus longues que la durée d'un spike, ce qui correspond à la biologie. Ne pas hésiter à diminuer les maximums de courants pour compenser la modification.

- Tester un réseau où la connexion entre le réservoir et les neurones de sortie est aléatoire.

- Une fois que le réseau réel sera fabriqué :
    - Utiliser ce réseau pour chercher à optimiser les paramètres STDP afin de rendre l'apprentissage proche de celui du réseau réel
    - Optimiser les paramètres Rise and Decay

f. Produit synaptique

Il a été nécessaire de définir le produit synaptique pour pouvoir modéliser la détection de mouvement. Le script *IRCICA_testProduitSynaptique* présente le mécanisme et doit maintenant être inséré sur la rétine artificielle présentée ci-après.



Workspace testproduitsynaptique

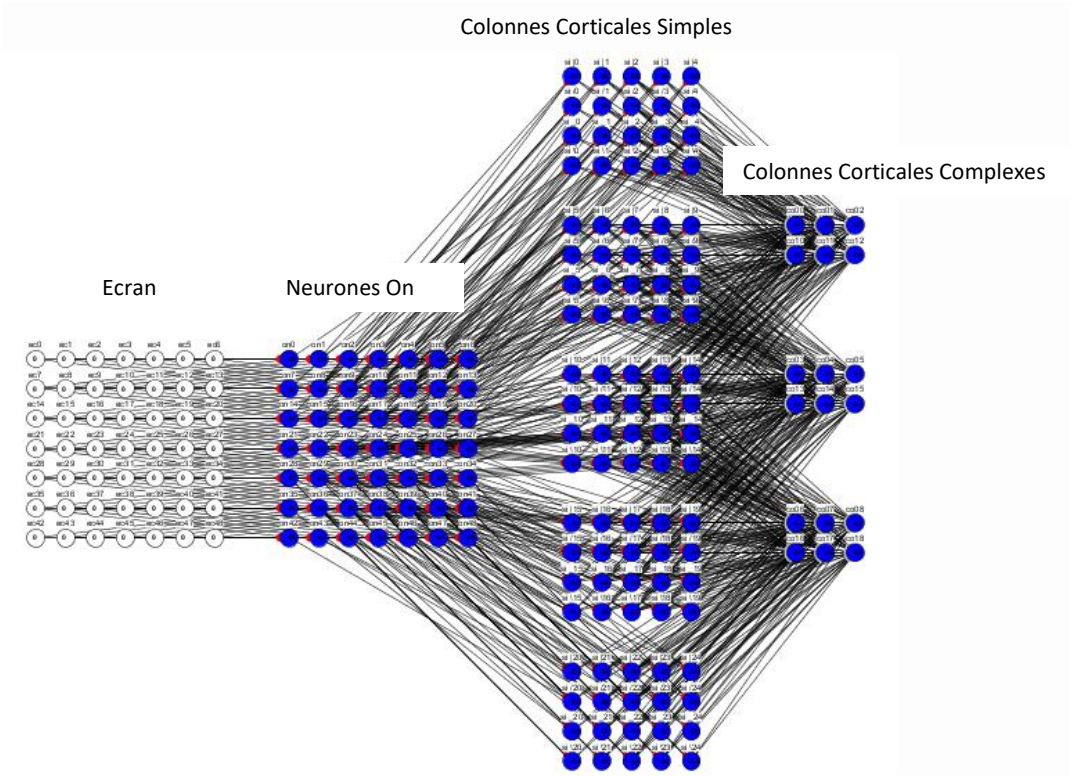
Le neurone ***n1*** (rouge) correspond au premier neurone à voir l'objet, il charge le neurone transitoire ***jump*** (jaune) par un courant synaptique « Jump and Decay » avec une longue constante de temps. Le neurone ***n2*** (bleur) spike lorsqu'il voit l'objet passer et charge le neurone transitoire ***rise*** (rose) avec un courant « Rise and Decay » de courte constante de temps. Les neurones ***jump*** et ***rise*** sont des neurones linéaires et leur « activation » est égale au courant synaptique qu'ils reçoivent à chaque instant. Le neurone ***n3*** (vert) reçoit comme courant d'excitation la valeur du produit des « activations » des deux neurones intermédiaires. Les neurones ex1 et ex2 sont des sources de courant, et les neurones testWeighted et testSynaptic sont des témoins pour comprendre comment fonctionne l' « input type » « Weighted » modifié.

g. Rétine Artificielle

i. Extrait du rapport de stage dédié au réseau de réservoir computing

3. Rétine artificielle

Le projet de modélisation de rétine artificielle dans le but de la créer à partir de neurones artificiels constitue le fil conducteur de mon stage. Le modèle de rétine développé est un premier essai et ne prétend pas égaler la biologie. L'objectif principal est plutôt de comprendre les mécanismes fondamentaux de la vision et de les reproduire en s'émancipant des contraintes liées à la biologie. Voici le réseau final obtenu à la fin de mon stage :



Capture d'écran de la rétine artificielle modélisée dans Simbrain

L'ensemble du système modélisé, depuis les photorécepteurs jusqu'au cortex V1 a été divisé en trois couches. Chaque couche joue le rôle d'un ou plusieurs types de cellules et remplit une fonction dans le processus de la vision.

Comme pour la rétine biologique, la rétine artificielle génèrera d'abord l'image observée en contraste avant d'en déduire des orientations et des contours, ainsi que des mouvements et des vitesses pour des images dynamiques. Ces fonctions seront effectuées de façon systématique sans apprentissage, c'est-à-dire sans plasticité synaptique. A partir de cela, les couches les plus profondes apprendront – à l'aide de la plasticité synaptique – à reconnaître certaines formes et à prédire le mouvement d'un objet à partir de ses déplacements.

Les réseaux de neurones dans le cerveau sont constitués d'un grand nombre de boucles de rétrocontrôle et d'inhibition latérale. Notre modèle de rétine artificielle ne possède pas ces caractéristiques les

mécanismes sous-jacents sont assez mal compris et donc difficilement reproductibles, ceci constitue la différence fondamentale avec la biologie, et donc la principale limite de la rétine artificielle.

a. Couche 0 : Les photorécepteurs

Pour modéliser les photorécepteurs, la rétine artificielle disposera d'un maillage de photodiodes éclairées par fibre optique. J'ai reproduit ce maillage dans *Simbrain* avec des neurones de type « linéaire » se comportant comme des sources de courant continu de puissance réglable. Ce maillage constitue la « couche 0 » de la rétine artificielle nommée « écran » dans la modélisation et ne contient aucun neurone artificiel, on appellera ses éléments des pixels dans la suite et cet. Cette couche ne reproduit pas les spécificités des cônes et des bâtonnets, mais elle suffit à recréer les mécanismes fondamentaux de la vision.

Chaque pixel délivre un courant proportionnel à la puissance lumineuse qu'il reçoit aux neurones de la « couche 1 » qui l'ont dans leur champ de réception. Cela permet de présenter des images nuancées à la rétine. La courbe d'ISI (cf II.2.c.iv) des neurones de la « couche 1 » permet de déterminer la plage de courant délivrée par les pixels.

Le maillage choisi est carré, car c'est le maillage le plus simple à réaliser et il permet de discerner les quatre orientations. Dans la modélisation, je choisis la taille de l'écran et les couches suivantes s'adaptent à ses dimensions.

b. Couche 1 : Les neurones On et Off – Contraste

La « couche 1 » constitue la première couche de la rétine artificielle constituée de neurones. Elle synthétise l'action des interneurones, des cellules ganglion du nerf optique et les neurones du LGN, ou du moins ce qu'on en comprend. Elle est composée d'une couche de neurones On parallèle à une couche de neurones Off. Ces couches génèrent les images de contraste, l'une étant le négatif de l'autre. Elles n'interagissent pas entre elles. Pour économiser du temps de calcul, les couches suivantes ont été modélisées et simulées à partir d'une seule de ces deux couches, la deuxième n'étant pas nécessaire aux applications simples envisagées pour cette rétine artificielle.

Pour cette couche, les éléments importants à définir sont le contraste, le calibrage des neurones On et Off, les champs de réception et les poids des connexions synaptiques.

- J'ai introduit la notion de contraste pour la rétine artificielle en m'inspirant du contraste défini en optique. Nous avons donc le contraste perçu par un neurone On définit à partir de son champ de réception :

$$Contraste = \frac{kI_{central} - (\sum_k I_k)}{kI_{central} + (\sum_k I_k)} \quad (15)$$

  Où $I_{central}$ est le courant délivré par le pixel central et les I_k sont les courants délivrés par les k pixels périphériques du champ de réception du neurone On.
- Le calibrage de cette couche correspond à la définition du contraste minimal entre deux pixels capable d'exciter un neurone On ou Off.
- Le champ de réception des cellules On et Off sur la « couche 0 » doit comporter une zone centrale et une zone périphérique. Le choix définitif s'est porté sur l'architecture « en petites croix » présentée sur la figure suivante car elle s'étend sur peu de neurones et est performante. Elle permet de gérer simplement les effets de bord. Le neurone C correspond à la zone centrale et les neurones A, B, D et E correspondent à la zone périphérique.

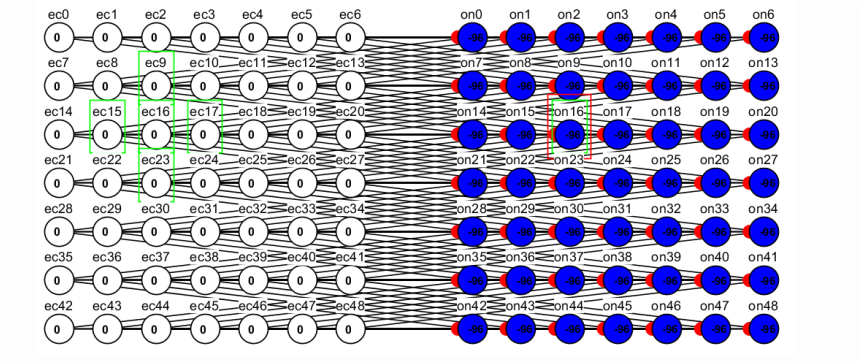
Capture d'écran de l'architecture du champ de réception d'un neurone On modélisé dans Simbrain

- Les poids des connexions synaptiques reliant le champ de réception au neurone sont définis à partir de ce contraste. Un éclairement homogène sur les cinq pixels du champ de réception correspond à un contraste nul donc cela n'active pas le neurone.
  Dès que le contraste atteint la valeur minimale définie au calibrage, le courant d'excitation reçu par le neurone devient assez grand pour l'exciter. Son 'ISI' est alors le plus grand possible.
  Les cellules On s'activent lorsque la zone centrale est la plus éclairée et les cellules Off s'activent lorsque la zone périphérique est la plus éclairée.

On peut émettre certaines réserves concernant ce choix d'architecture de champ de réception :

- Elle ne possède aucune redondance de l'information car chaque pixel appartient exactement à une zone centrale d'un champ de réception de neurone On ou Off.
- Les images possédant beaucoup de variations locales d'éclairage peuvent générer une image de contraste incompréhensible car trop détaillée.
- Le contraste entre deux pixels voisins sur une même diagonale ne provoque aucune réaction car la zone périphérique du champ de réception est limitée aux directions horizontales et verticales.

Le niveau de simplification atteint avec cette couche correspond à l'émancipation des contraintes biologiques évoquées plus haut. Les simulations effectuées jusqu'à maintenant montraient que ce modèle était satisfaisant. Voici un zoom sur les couches 1 et 2 avec un neurone On encadré en rouge et son champ de réception encadré en vert.



Capture d'écran des couches 0 et 1 de la rétine artificielle modélisée dans Simbrain

c. Couche 2 : Les colonnes corticales simples du cortex – Orientations

La « couche 2 » modélise les colonnes corticales simples du cortex V1. Cette couche est directement connectée à la « couche 1 » et possède des cellules sensibles à des barres orientées dans les quatre directions et des cellules sensibles au déplacement de ces barres. On définit ici aussi le champ de réception de chaque colonne : un carré de 3 neurones On ou Off de côté et par transitivité un carré de 3 pixels de côté sur l'écran. Le nombre de colonnes de la couche dépend donc du niveau de superposition des colonnes, la figure suivante illustre les trois types d'agencement possibles.

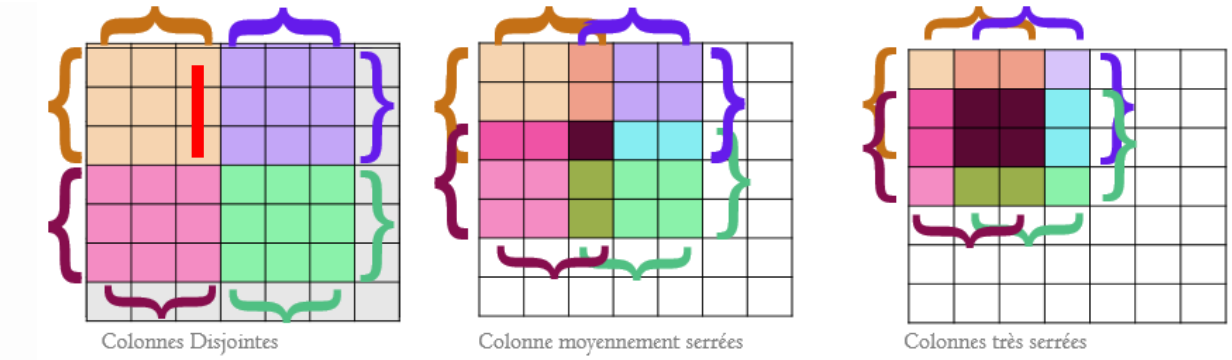
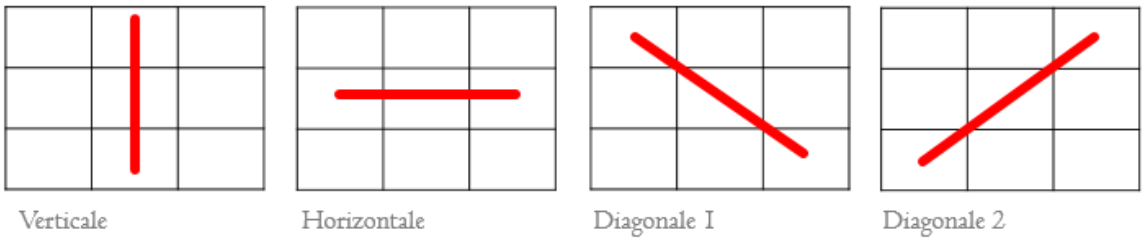


Schéma de l'agencement des champs de réception des colonnes corticales les uns par rapport aux autres

Afin de bien comprendre la problématique d'agencement, imaginons qu'on parcoure l'écran en partant du coin en haut et à gauche. On repère alors chaque champ de réception par son pixel le plus en haut à gauche. Pour l'agencement le moins serré, on doit se déplacer de trois neurones vers la droite ou de trois neurones vers le bas pour trouver le champ de réception suivant. Dans le cas intermédiaire, il suffit de se déplacer de seulement deux pixels pour rencontrer le champ de réception suivant, on a alors une superposition des champs de réception. Enfin, dans le cas le plus serré, un seul pixel suffit et la superposition des champs de réception est plus grande. J'ai recommandé le maillage le plus serré car les autres maillages rendent impossible la détection de certaines orientations, par exemple l'orientation rouge sur le premier maillage est impossible à détecter avec les deux premiers maillages car elle n'est pas placée sur la verticale centrale du maillage.

i. Les orientations

Chaque colonne corticale modélisée possède quatre neurones d'orientation et est connectée à une zone carrée de l'écran de 3 pixels de côté. Ce champ de réception de la colonne permet ainsi de voir les deux orientations diagonales, l'orientation verticale et l'orientation horizontale. Ce sont les neurones On et Off qui vont exciter les neurones des colonnes corticales. Les poids synaptiques sont choisis de telle sorte que l'activation des trois neurones On ou Off de l'orientation vont pouvoir activer le neurone sensible à cette orientation dans la colonne. La figure suivante affiche les 4 orientations détectables respectivement par les 4 neurones de la colonne liée à ce champ de réception :



Représentation des orientations associées à un champ de réception de colonne corticale

Les simulations de cette couche furent satisfaisantes et les colonnes pouvaient reconnaître les orientations avec une très grande efficacité. Certaines réserves ont d'abord été émises pour les cas où les trois neurones

de l'orientation ne spikeraient pas simultanément mais ce problème disparait dès l'instant ou l'on ajuste la constante de temps des courants synaptiques – « spikes responders » – comme dans la biologie (cf II.3.b.)

ii. Les mouvements

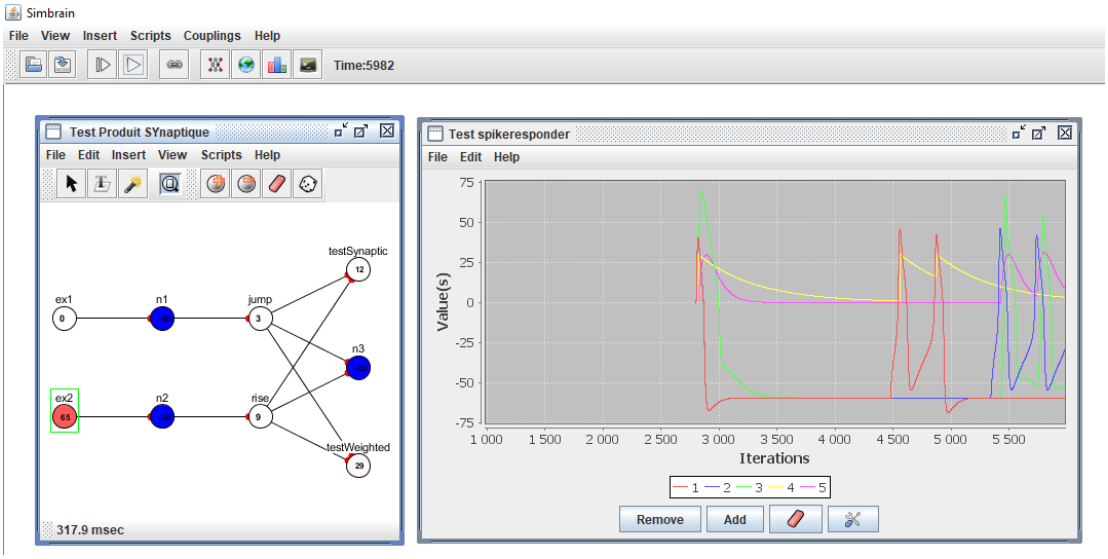
Le mécanisme proposé par les ingénieurs de l'équipe bioinspirée pour la sensibilité au mouvement repose sur le même principe que celui adopté pour la STDP : la charge d'un condensateur à décharge lente va garder en mémoire l'information du passage de l'objet sur un temps court. S'il a lieu avant que le condensateur se soit trop déchargé, le potentiel d'action généré par le passage de l'objet à la position suivante va se multiplier avec l'énergie restante dans le condensateur pour exciter le neurone témoin du mouvement.

La modélisation de cette solution a nécessité la modification de l' « input type » « WEIGHTED » afin de permettre le produit de courants d'excitation dans *Simbrain (cf.* II.2.a.iv). J'ai ensuite utilisé un neurone linéaire chargé par le courant « SYNAPTIC » « Jump and Decay » (cf. II.3.b.ii) en guise de condensateur.

La décharge du condensateur est calibrée pour être bien plus longue que la durée d'existence d'un courant d'excitation provoqué par un potentiel d'action. Ainsi, le neurone témoin du mouvement vers la droite ne peut théoriquement pas être témoin d'un mouvement vers la gauche, à moins que la vitesse de déplacement de l'objet soit très grande. Et dans ce cas, on pourrait parler d'effet d'optique pour la rétine artificielle dont les neurones des sensibles au déplacement dans les deux sens s'activeraient et elle « verrait » l'objet se déplacer en même temps dans les deux sens.

L'ingéniosité de cette solution réside dans le fait qu'elle peut déterminer la vitesse de déplacement de l'objet, car produit des courant reçu par le neurone sensible au déplacement est proportionnel à la vitesse de l'objet. En effet, si l'objet se déplace rapidement, alors le condensateur n'aura pas eu le temps de se décharger et le courant d'excitation sera élevé, si au contraire l'objet se déplace lentement, le condensateur sera relativement déchargé et le courant d'excitation sera plus bas. Ainsi, l'ISI du neurone dépendra de la vitesse et permettra donc de remonter à cette information.

La solution technique n'a été proposée que quelques semaines avant la fin de mon stage et j'ai pu la modéliser seule mais je n'ai pas pu l'implémenter sur la modélisation de rétine artificielle. La figure suivante est une capture d'écran du modèle non implémenté :



Capture d'écran du mécanisme de détection de mouvements modélisé sur Simbrain

Le neurone ***n1*** (rouge) correspond au premier neurone à voir l'objet, il charge le neurone transitoire ***jump*** (jaune) par un courant synaptique « Jump and Decay » avec une longue constante de temps. Le neurone ***n2*** (bleur) spike lorsqu'il voit l'objet passer et charge le neurone transitoire ***rise*** (rose) avec un courant « Rise and Decay » de courte constante de temps. Les neurones ***jump*** et ***rise*** sont des neurones linéaires et leur « activation » est égale au courant synaptique qu'ils reçoivent à chaque instant. Le neurone ***n3*** (vert) reçoit comme courant d'excitation la valeur du produit des « activations » des deux neurones intermédiaires. Les neurones ex1 et ex2 sont des sources de courant, et les neurones testWeighted et testSynaptic sont des témoins pour comprendre comment fonctionne l' « input type » « Weighted » modifié.

iii. Résultats

Une grande partie des tests effectués sur cette couche, principalement pour ce qui concerne les orientations, utilisaient le paramétrage erroné présenté dans la partie II.2.c.vi. avant que je ne réalise ses failles en utilisant le reservoir computing. C'est pourquoi des simulations de la couche 2 pourront plus tard remettre en question certains des choix que j'ai effectués.

d. Couche 3 : Colonnes corticales complexes – Reconnaissance de formes

Cette couche correspond à la couche des colonnes corticales complexes, capables de discerner des formes à partir des orientations de leur champ de réception. Cette couche est la première couche de la rétine non systématique car elle fait intervenir de la plasticité synaptique. Ces synapses plastiques entre la couche 2 et la couche 3 permettent un apprentissage puis une reconnaissance des formes.

Les axes de réflexion liés à cette couche s'éloignent quelque peu de la biologie dont les connaissances lacunaires de la communauté scientifique ne peuvent apporter de solutions concrètes bioinspirées.

<u>Le champ de réception de la couche</u>

Ici le champ de réception va énormément influencer la taille des formes reconnues. Après tout un carré peut aussi bien être sur une zone de l'écran de 3x3 pixels que de 9x9 pixels. Si le champ de réception est trop petit, alors certaines formes ne seront pas « vues » par la rétine. C'est pourquoi il est envisageable de devoir développer un mécanisme de généralisation des formes sans considérer leurs tailles.

<u>La méthode d'apprentissage</u>

Il existe plusieurs méthodes d'apprentissage dans les réseaux de neurones. Par exemple, le reservoir computing en est une. Dans la littérature on différentie l'apprentissage supervisé et l'apprentissage non supervisé.

Lorsqu'il est supervisé, l'apprentissage ne peut se faire que si un expérimentateur supervise le réseau pendant sa phase d'apprentissage. C'est-à-dire qu'il montre chaque objet d'une banque d'images au réseau et lui indique la sortie attendue. L'apprentissage est terminé lorsque les poids des synapses plastiques en jeu ont convergé vers leurs valeurs finales et que le réseau est capable de reconnaître des objets similaires à ceux de l'apprentissage. Cet apprentissage est le plus simple à mettre en place, cependant il présente quelques limites :

- Le nombre d'objets présentés lors de la phase d'apprentissage peut parfois être immense. Certains réseaux doivent avoir appris sur plusieurs millions d'objets pour être capable d'effectuer une reconnaissance.
- Les réseaux d'apprentissage supervisé classiques ne sont pas capables de s'adapter, un réseau entrainé à reconnaitre des chats et des chiens sera incapable de reconnaitre autre chose qu'un chat ou un chien.
- Il semble que les contrairement à ces réseaux, les cerveaux biologiques sont capables d'apprendre de façon non supervisée et de faire preuve d'abstraction et de généralisation.

L'apprentissage non supervisé est cependant plus difficile à mettre en place et ne fonctionne pas systématiquement. Il consiste à présenter des objets au réseau pendant la phase d'apprentissage sans lui indiquer la sortie attendue, puis le laisser déterminer ses sorties et faire converger ses poids synaptiques.

Je me suis intéressé aux neurones a cliques pour l'apprentissage de la « couche 3 » mais je n'ai pas trouvé de façon de les implémenter dans la rétine artificielle.

La généralisation

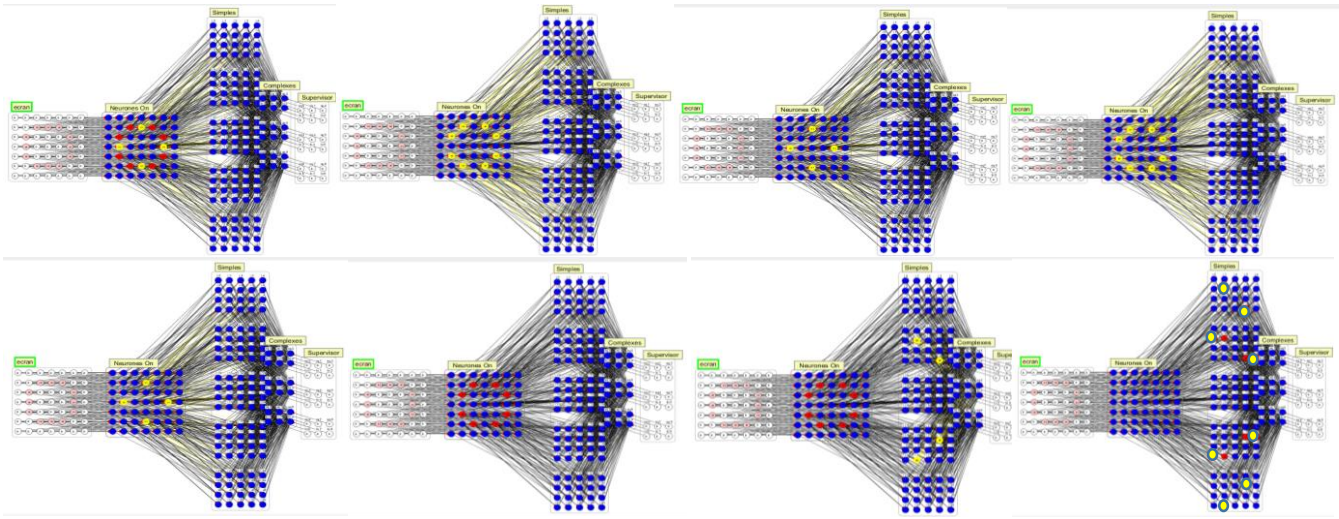
Il existe des dizaines de façons de dessiner la même forme géométrique sur un écran de pixels. Les neurones de la « couche 3 » doivent être capable de généraliser et d'abstraire une forme observée pour la reconnaître, indépendamment de sa taille ou de la précision du trait.

Tous ces problèmes sont liés les uns aux autres et cela nous laisse penser qu'il sera difficile de créer une « couche 3 » fonctionnelle et aussi économe que les couches précédentes. Les mécanismes de reconnaissance sont très dépendants d'autres zones du cerveau en lien avec la mémoire, le langage et les conceptions abstraites d'objets. Ainsi, je n'ai pas eu le temps d'étudier en profondeur les problématiques liées à cette couche de la rétine artificielle.

ii. Description du script

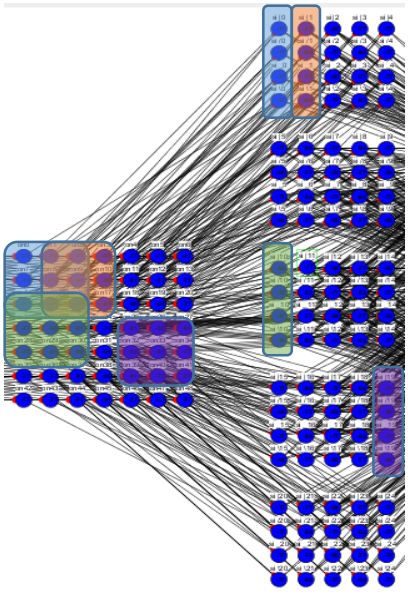
Le script *IRCICA_retine_On_complexe* définit les couches une par une depuis la couche 0 jusqu'à la couche 3 qui n'est pas fonctionnelle. La couche 0, l'écran et ses « pixels » ***ec_*** est connectée à la couche 1 et ses neurones On ***on_*** selon la structure du champ de réception de chacune. Les cellules ***on_*** sont de même reliées aux cellules de la couche 2***, si_***, représentant les colonnes corticales simples. Chaque colonne possède 4 cellules pour les quatre orientations et est reliée avec des synapses plastiques aux cellules de la couche 3, mais le mécanisme d'apprentissage doit encore être défini pour cette couche des colonnes corticales complexes ***co_***.

iii. Resultats en images



Capture d'écran de la réaction de la rétine à un affichage sur l'écran

Cette figure se lit de gauche à droite, de haut en bas. Elle représente l'activité de la rétine artificielle lorsqu'on affiche un carré à l'écran (en rose). Les neurones jaunes représentent les neurones en train de spiker. La couche 2 est composée de colonnes corticales de 4 neurones chacun sensible à une orientation. De haut en bas, les orientations détectées sont |, /, _, \. Chaque colonne correspond à une zone de la couche 1, la répartition se fait selon la figure suivante :



Champs de réception des colonnes corticales simples

iv. Pistes d'étude

Il reste beaucoup de travail sur la rétine artificielle, voici quelques pistes d'étude.

- Le mécanisme de détection du mouvement présenté dans le réseau *IRCICA_testProduitSynaptique* doit être implémenté dans une couche intermédiaire entre les couches 2 et 3.

- L’étude n’a pas pu être poussée jusqu’au bout de la rétine et aucun apprentissage n’a pu être effectué de façon satisfaisante. Les choix effectués jusqu’à présent sont donc éventuellement à revoir une fois que le modèle de rétine artificielle sera complet.

- L’apprentissage des formes nécessite une réflexion à part entière pour prendre en compte la généralisation des tailles, des orientations et des formes. Pour cela le mécanisme d’apprentissage ne pourra sûrement pas être simple et déterministe et nécessitera des études plus approfondies concernant les réseaux d’apprentissage possibles. Certains neurones devront peut-être même être capables de compter des coins ou des côtés, et de mesurer grossièrement des angles.

- Ici aussi la modification des constantes de temps de la fonction Rise and Decay devrait changer la donne concernant le fonctionnement du réseau. De cette façon les décalages dans le temps de l’excitation des différents pixels ne devraient plus être dérangeants.

- Dans un premier temps le réseau a été développé de façon à ce que les photodiodes soient directement connectées aux neurones on et off, mais il peut être intéressant pour la rétine qu’une couche intermédiaire entre les photorécepteurs et les neurones On et Off soit installée et comporte des neurones Morris-Lecar simplement excités par les photos récepteurs, sur le modèle de l’écran du réservoir computing.

Annexes 2 : Scripts *Simbrain* des réseaux

Bibliothèque commune à tous les scripts

```
// On importe tout dans le doute de manquer de quelque chose (source
de beaucoup d erreurs)
import org.simbrain.network.*;
import org.simbrain.network.core.*;
import org.simbrain.network.networks.*;
import org.simbrain.network.neuron_update_rules.*;
import org.simbrain.workspace.*;
import org.simbrain.network.interfaces.*;
import org.simbrain.network.connections.*;
import org.simbrain.network.NetworkComponent;
import org.simbrain.network.core.*;
import org.simbrain.network.core.SynapseUpdateRule;
import org.simbrain.network.synapse_update_rules.spikeresponders.*;
import org.simbrain.network.desktop.*;
import org.simbrain.network.layouts.*;
import org.simbrain.network.networks.*;
import org.simbrain.network.neuron_update_rules.*;
import org.simbrain.network.synapse_update_rules.*;
import org.simbrain.network.update_actions.*;
import org.simbrain.network.util.*;
import org.simbrain.network.core.NeuronUpdateRule.InputType;
import org.simbrain.network.groups.*;
import org.simbrain.util.*;
import org.simbrain.plot.timeseries.*;
import org.simbrain.docviewer.*;
import org.simbrain.util.math.*;
import org.simbrain.util.environment.*;
import org.simbrain.workspace.updater.*;
import org.simbrain.world.odorworld.*;
import org.simbrain.world.odorworld.entities.*;
import org.simbrain.world.odorworld.sensors.*;
import java.util.concurrent.*;
import javax.swing.JInternalFrame;
import javax.swing.JSlider.*;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.geom.Point2D;
import javax.swing.*;
import java.util.*;
import java.util.concurrent.*;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.geom.Point2D;
import org.simbrain.network.NetworkComponent;
import org.simbrain.network.core.*;
import org.simbrain.network.connections.*;
import org.simbrain.network.desktop.*;
```

```
import org.simbrain.network.layouts.*;
import org.simbrain.network.neuron_update_rules.*;
import org.simbrain.network.synapse_update_rules.*;
import org.simbrain.workspace.*;
import org.simbrain.util.*;
import javax.swing.JInternalFrame;
import java.util.*;
```

Neurone-synapse-neurone

```
// Parametres Neurone Morris-Lecar
      //Il suffit de commenter les paramÃ¨tres pour que leur valeur
soit la valeur standard de Simbrain
MorrisLecarRule generalRule = new MorrisLecarRule();
//generalRule.setG_Ca(6.9);
//generalRule.setG_K(10);
//generalRule.setG_L(0.9);
//generalRule.setvRest_Ca(100);
//generalRule.setvRest_k(-100);
//generalRule.setvRest_L(-100);
//generalRule.setcMembrane(0.05);
//generalRule.setV_m1(0);
//generalRule.setV_m2(40);
//generalRule.setV_w1(0);
//generalRule.setV_w2(40);
//generalRule.setPhi(0.3);
generalRule.setI_bg(0);  // courant d excitation du neurone toujours
present pendant la simulation
generalRule.setThreshold(0);
//generalRule.setAddNoise(false);
generalRule.setInputType(InputType.SYNAPTIC); // Les entrees soient
provoquees par les spikes uniquement


double timeStep = 0.1;
double tau = 7;
double maxResp = 30;
double iex = 60; // courant d excitation neurone 1
double poidsSynapse = 1;


void main() {
    buildNetwork();
      addTimeSeries();
      addButtons();

}

void buildNetwork() {
    NetworkComponent networkComponent = new
NetworkComponent("Neurone Synapse Neurone ");
    workspace.addWorkspaceComponent(networkComponent);
    Network network = networkComponent.getNetwork();
    network.setTimeStep(timeStep);



      Neuron nML1 = new Neuron(network, generalRule.deepCopy());
      nML1.setLocation(0,0);
      nML1.setLabel("neurone 1");
      nML1.getUpdateRule().setI_bg(iex);
      network.addNeuron(nML1);
```

```
      Neuron nML2 = new Neuron(network, generalRule.deepCopy());
      nML2.setLocation(50,0);
      nML2.setLabel("neurone 2");
      nML2.getUpdateRule().setI_bg(0);
      network.addNeuron(nML2);


    Synapse syn = new Synapse(nML1 , nML2, poidsSynapse);
      syn.setSpikeResponder(new RiseAndDecay());
      syn.getSpikeResponder().setMaximumResponse(maxResp);
      syn.getSpikeResponder().setTimeConstant(tau);
      network.addSynapse(syn);
}

void addTimeSeries() {

    NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent("Neurone Synapse Neurone "));
      Network network = networkComponent.getNetwork();

      // Make time series chartLM LM
    TimeSeriesPlotComponent chartML = new TimeSeriesPlotComponent("N
- S - N");
      chartML.getModel().setWindowSize(5000);
    workspace.addWorkspaceComponent(chartML);

desktop.getDesktopComponent(chartML).getParentFrame().setBounds(500,
0,800,350);


      Neuron nML1 = network.getNeuronByLabel("neurone 1");
      Neuron nML2 = network.getNeuronByLabel("neurone 2");

      PotentialProducer neuronProducerML1 =
NetworkComponent.getNeuronProducer(networkComponent, nML1 ,
"getActivation");
      PotentialConsumer timeSeriesConsumerML1 =
chartML.getPotentialConsumers().get(0);
      workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML1, timeSeriesConsumerML1));

      PotentialProducer neuronProducerML2 =
NetworkComponent.getNeuronProducer(networkComponent, nML2 ,
"getActivation");
      PotentialConsumer timeSeriesConsumerML2 =
chartML.getPotentialConsumers().get(1);
      workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML2, timeSeriesConsumerML2));




}
```

```
void addButtons() {

      // Recuperation des elements utiles
    NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent("Neurone Synapse Neurone "));
      Network network = networkComponent.getNetwork();


    // Start making the buttons!
    JInternalFrame internalFrame = new JInternalFrame("Parametrage",
true, true);
    LabelledItemPanel panel = new LabelledItemPanel();

      Neuron nML1 = network.getNeuronByLabel("neurone 1");
      Neuron nML2 = network.getNeuronByLabel("neurone 2");
      Synapse s = network.getSynapse(nML1, nML2);

      JButton buttonPursuer = new JButton("Change !");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                s.setStrength(Double.parseDouble(sw.getText()));
                nML1.setActivation(10);
                nML1.getUpdateRule().setI_bg(10);
            workspace.iterate();
        }});
    panel.addItem("synapse weight", buttonPursuer);

      JTextField sw = new JTextField(String.valueOf(poidsSynapse));
    panel.addItem(" ", sw);

      JButton buttonPursuer = new JButton("Change !");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {

      nML1.getUpdateRule().setI_bg(Double.parseDouble(iex.getText()))
;
            workspace.iterate();
        }});
    panel.addItem("excitation current", buttonPursuer);

      JTextField iex = new JTextField(String.valueOf(iex));
    panel.addItem(" ", iex);


  // Set up Frame
    internalFrame.setLocation(0,500);
    internalFrame.getContentPane().add(panel);
    internalFrame.setVisible(true);
    internalFrame.pack();
    desktop.addInternalFrame(internalFrame);

}
main();
```

IRCICA_TestInhibition

```
// Parametres Neurone Morris-Lecar
      //Il suffit de commenter les paramÃ¨tres pour que leur valeur
soit la valeur standard de Simbrain
MorrisLecarRule generalRule = new MorrisLecarRule();
//generalRule.setG_Ca(6.9);
//generalRule.setG_K(10);
//generalRule.setG_L(0.9);
//generalRule.setvRest_Ca(100);
//generalRule.setvRest_k(-100);
//generalRule.setvRest_L(-100);
//generalRule.setcMembrane(0.05);
//generalRule.setV_m1(0);
//generalRule.setV_m2(40);
//generalRule.setV_w1(0);
//generalRule.setV_w2(40);
//generalRule.setPhi(0.3);
generalRule.setI_bg(0);  // courant d excitation du neurone toujours
present pendant la simulation
generalRule.setThreshold(0);
//generalRule.setAddNoise(false);
generalRule.setInputType(InputType.SYNAPTIC); // Les entrees soient
provoquees par les spikes uniquement

double tau = 7;
double maxResp = 30;
double poidsSynaptic1 = 1;
double poidsSynaptic2 = -1;
double poidsSynaptic3 = 1;
double iex1 = 60;
double iex2 = 65;
double iex3 = 60;
void main() {
    buildNetwork();
      addTimeSeries();
      addButtons();
}

void buildNetwork() {
    NetworkComponent networkComponent = new NetworkComponent("Test
Inhibition Excitation");
    workspace.addWorkspaceComponent(networkComponent);
    Network network = networkComponent.getNetwork();
    network.setTimeStep(.1);

      Neuron nML1 = new Neuron(network, generalRule.deepCopy());
      nML1.setLocation(2,2);
      nML1.setLabel("n1");
      nML1.getUpdateRule().setI_bg(iex1);
      nML1.getUpdateRule().setAddNoise(false);
      nML1.getUpdateRule().setInputType(InputType.SYNAPTIC);
      network.addNeuron(nML1);

      Neuron nML2 = new Neuron(network, generalRule.deepCopy());
```

```
      nML2.setLocation(2,50);
      nML2.setLabel("n2");
      nML2.getUpdateRule().setI_bg(iex2);
      nML2.getUpdateRule().setAddNoise(false);
      nML2.getUpdateRule().setInputType(InputType.SYNAPTIC);
      network.addNeuron(nML2);

      Neuron nML3 = new Neuron(network, generalRule.deepCopy());
      nML3.setLocation(2,100);
      nML3.setLabel("n3");
      nML3.getUpdateRule().setI_bg(iex3);
      nML3.getUpdateRule().setAddNoise(false);
      nML3.getUpdateRule().setInputType(InputType.SYNAPTIC);
      network.addNeuron(nML3);

      Neuron post = new Neuron(network, generalRule.deepCopy());
      post.setLocation(50,50);
      post.setLabel("post");
      post.getUpdateRule().setI_bg(0);
      post.getUpdateRule().setAddNoise(false);
      post.getUpdateRule().setInputType(InputType.SYNAPTIC);
      network.addNeuron(post);

    Synapse syn = new Synapse(nML1 , post, poidsSynaptic1);
      syn.setSpikeResponder(new RiseAndDecay());
      syn.getSpikeResponder().setMaximumResponse(maxResp);
      syn.getSpikeResponder().setTimeConstant(tau);
      network.addSynapse(syn);

      Synapse syn = new Synapse(nML2 , post, poidsSynaptic2);
      syn.setSpikeResponder(new RiseAndDecay());
      syn.getSpikeResponder().setMaximumResponse(maxResp);
      syn.getSpikeResponder().setTimeConstant(tau);
      network.addSynapse(syn);

      Synapse syn = new Synapse(nML3 , post, poidsSynaptic3);
      syn.setSpikeResponder(new RiseAndDecay());
      syn.getSpikeResponder().setMaximumResponse(maxResp);
      syn.getSpikeResponder().setTimeConstant(tau);
      network.addSynapse(syn);
}

void addTimeSeries() {

    NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent("Test Inhibition Excitation"));
      Network network = networkComponent.getNetwork();

      // Make time series chartLM LM
    TimeSeriesPlotComponent chartML = new
TimeSeriesPlotComponent("Inhibition - Excitation", 4);
      chartML.getModel().setWindowSize(5000);
    workspace.addWorkspaceComponent(chartML);

desktop.getDesktopComponent(chartML).getParentFrame().setBounds(500,
0,800,350);
```

```
      Neuron nML1 = network.getNeuronByLabel("n1");
      Neuron nML2 = network.getNeuronByLabel("n2");
      Neuron nML3 = network.getNeuronByLabel("n3");
      Neuron post = network.getNeuronByLabel("post");

      PotentialProducer neuronProducerML1 =
NetworkComponent.getNeuronProducer(networkComponent, nML1 ,
"getActivation");
      PotentialConsumer timeSeriesConsumerML1 =
chartML.getPotentialConsumers().get(0);
      workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML1, timeSeriesConsumerML1));

      PotentialProducer neuronProducerML2 =
NetworkComponent.getNeuronProducer(networkComponent, nML2 ,
"getActivation");
      PotentialConsumer timeSeriesConsumerML2 =
chartML.getPotentialConsumers().get(1);
      workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML2, timeSeriesConsumerML2));

      PotentialProducer neuronProducerML3 =
NetworkComponent.getNeuronProducer(networkComponent, nML3 ,
"getActivation");
      PotentialConsumer timeSeriesConsumerML3 =
chartML.getPotentialConsumers().get(2);
      workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML3, timeSeriesConsumerML3));

      PotentialProducer neuronProducerML4 =
NetworkComponent.getNeuronProducer(networkComponent, post ,
"getActivation");
      PotentialConsumer timeSeriesConsumerML4 =
chartML.getPotentialConsumers().get(3);
      workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML4, timeSeriesConsumerML4));




}

void addButtons() {

      // Recuperation des elements utiles
     NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent("Test Inhibition Excitation"));
      Network network = networkComponent.getNetwork();


     // Start making the buttons!
     JInternalFrame internalFrame = new JInternalFrame("Parametrage",
true, true);
     LabelledItemPanel panel = new LabelledItemPanel();
```

```
      Neuron nML1 = network.getNeuronByLabel("n1");
      Neuron nML2 = network.getNeuronByLabel("n2");
      Neuron nML3 = network.getNeuronByLabel("n3");
      Neuron post = network.getNeuronByLabel("post");

      Synapse s1 = network.getSynapse(nML1, post);
      Synapse s2 = network.getSynapse(nML2, post);
      Synapse s3 = network.getSynapse(nML3, post);


      JButton buttonPursuer = new JButton("Change !");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                s1.setStrength(Double.parseDouble(sw1.getText()));
            workspace.iterate();
        }});
    panel.addItem("synapse 1 ", buttonPursuer);

      JTextField sw1 = new
JTextField(String.valueOf(poidsSynaptic1));
    panel.addItem("", sw1);

      JButton buttonPursuer = new JButton("Change !");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                s2.setStrength(Double.parseDouble(sw2.getText()));
            workspace.iterate();
        }});
    panel.addItem("synapse 2", buttonPursuer);

      JTextField sw2 = new
JTextField(String.valueOf(poidsSynaptic2));
    panel.addItem("", sw2);

      JButton buttonPursuer = new JButton("Change !");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                s3.setStrength(Double.parseDouble(sw3.getText()));
            workspace.iterate();
        }});
    panel.addItem("synapse 3", buttonPursuer);

      JTextField sw3 = new
JTextField(String.valueOf(poidsSynaptic3));
    panel.addItem("", sw3);

      JButton buttonPursuer = new JButton("Change !");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {

      nML1.getUpdateRule().setI_bg(Double.parseDouble(iex1.getText())
);
            workspace.iterate();
        }});
    panel.addItem("iex 1 ", buttonPursuer);
```

```
      JTextField iex1 = new JTextField(String.valueOf(iex1));
    panel.addItem("", iex1);

      JButton buttonPursuer = new JButton("Change !");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {

      nML2.getUpdateRule().setI_bg(Double.parseDouble(iex2.getText())
);
            workspace.iterate();
        }});
    panel.addItem("iex 2", buttonPursuer);

      JTextField iex2 = new JTextField(String.valueOf(iex2));
    panel.addItem("", iex2);

      JButton buttonPursuer = new JButton("Change !");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {

      nML3.getUpdateRule().setI_bg(Double.parseDouble(iex3.getText())
);
            workspace.iterate();
        }});
    panel.addItem("iex 3", buttonPursuer);

      JTextField iex3 = new JTextField(String.valueOf(iex3));
    panel.addItem("", iex3);

  // Set up Frame
    internalFrame.setLocation(0,500);
    internalFrame.getContentPane().add(panel);
    internalFrame.setVisible(true);
    internalFrame.pack();
    desktop.addInternalFrame(internalFrame);

}

main();
```

IRCICA_RiseAndDecayTest

```
// Parametres Utiles
int n = 4; // nombre de lignes de neurones
double iex = 65; // courant d excitation des preneurones
double iphi = 5; // induire un courant de dephasage dans les
preneurones
double poidsSynaptic = 1; // Toutes les synapses ont le mÃªme poids
dans ce reseau
double maxResp = 30; // maximum du courant d excitation synaptique
rise and decay
double tau = 5; // constante de temps du courant d excitation
synaptique rise and decay
int window = 5000; // taille de la fenÃªtre des graphiques
```

```
double timeStep = 0.1; // Pas de temps du reseau

      //Parametres Neurone Morris-Lecar

//Il suffit de commenter les paramÃ¨tres pour que leur valeur soit
la valeur standard de Simbrain

MorrisLecarRule generalRule = new MorrisLecarRule();
//generalRule.setG_Ca(6.9);
//generalRule.setG_K(10);
//generalRule.setG_L(0.9);
//generalRule.setvRest_Ca(100);
//generalRule.setvRest_k(-100);
//generalRule.setvRest_L(-100);
//generalRule.setcMembrane(0.05);
//generalRule.setV_m1(0);
//generalRule.setV_m2(40);
//generalRule.setV_w1(0);
//generalRule.setV_w2(40);
//generalRule.setPhi(0.3);
//generalRule.setI_bg(0);  // courant d excitation du neurone
toujours present pendant la simulation
generalRule.setThreshold(0);
//generalRule.setAddNoise(false);
generalRule.setInputType(InputType.SYNAPTIC); // Les entrees soient
provoquees par les spikes uniquement


void main() {
    buildNetwork();
      addTimeSeries();

}

void buildNetwork() {

      // Creation Network
    NetworkComponent networkComponent = new NetworkComponent("Test
Network");
    workspace.addWorkspaceComponent(networkComponent);
    Network network = networkComponent.getNetwork();
    network.setTimeStep(timeStep);

      // Creation neurones et synapses
      for(i = 0; i < n ; i++){

            // Neurone excitateur
            Neuron pre = new Neuron(network, generalRule.deepCopy());
            pre.setLocation(0, 50 * i);
            pre.setLabel("spiker" + String.valueOf(i));
            pre.getUpdateRule().setI_bg(iex + i * iphi);
            network.addNeuron(pre);

            // Neurone Morris-Lecar excite par les fonctions "rise
and decay"
            Neuron postML = new Neuron(network,
```

```
generalRule.deepCopy());
           postML.setLocation(200, 50 * i);
           postML.setLabel("postML" + String.valueOf(i));
           postML.getUpdateRule().setI_bg(0);
           network.addNeuron(postML);

           // Neurone de type "Lineaire" permettant de connaÃ®tre le
comportement de la fonction "rise and decay"
           Neuron postLin = new Neuron(network, "LinearRule");
           postLin.setLocation(400, 50 * i);
           postLin.setLabel("postLin" + String.valueOf(i));
           postLin.setUpperBound(1000);
           postLin.setLowerBound(-1000);
           postLin.getUpdateRule().setInputType(InputType.SYNAPTIC);
           network.addNeuron(postLin);

           // Connexion avec tous les neurones pre deja crees
           for (j = 0; j <= i ; j++){

                pre = network.getNeuronByLabel( "spiker" +
String.valueOf(j));

                // Synapse entre l excitateur et le neurone Morris-
Lecar
                Synapse sML = new Synapse(pre, postML,
poidsSynaptic);
                sML.setSpikeResponder(new RiseAndDecay());
                sML.getSpikeResponder().setMaximumResponse(maxResp);
                sML.getSpikeResponder().setTimeConstant(tau);
                network.addSynapse(sML);

                // Synapse entre l excitateur et le neurone Lineaire
                Synapse sLin = new Synapse(pre, postLin,
poidsSynaptic);
                sLin.setSpikeResponder(new RiseAndDecay());

     sLin.getSpikeResponder().setMaximumResponse(maxResp);
                sLin.getSpikeResponder().setTimeConstant(tau);

                network.addSynapse(sLin);
           }
     }
}



void addTimeSeries() {

     // Recuperation des elements necessaires
    NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent("Test Network"));
     Network network = networkComponent.getNetwork();

     // Graphe des neurones excitateurs
    TimeSeriesPlotComponent chartPre = new
TimeSeriesPlotComponent("Neurones Pre", n);
```

```
      chartPre.getModel().setWindowSize(window);
     workspace.addWorkspaceComponent(chartPre);
//
desktop.getDesktopComponent(chartPre).getParentFrame().setBounds(500
,0,800,350);

      // Graphe des neurones Morris-Lecar
     TimeSeriesPlotComponent chartML = new
TimeSeriesPlotComponent("Neurones Post", n);
      chartML.getModel().setWindowSize(window);
     workspace.addWorkspaceComponent(chartML);
//
desktop.getDesktopComponent(chartML).getParentFrame().setBounds(500,
0,800,350);

      // Graphe des neurones Lineaires
      TimeSeriesPlotComponent chartLin = new
TimeSeriesPlotComponent("Synaptic Signal", n);
      chartLin.getModel().setWindowSize(window);
     workspace.addWorkspaceComponent(chartLin);
//
desktop.getDesktopComponent(chartLin).getParentFrame().setBounds(500
,350,800,350);


      // Couplage des neurones et des graphiques
      for (i = 0; i < n; i++){

            // Recuperation des neurones
            Neuron pre = network.getNeuronByLabel("spiker" +
String.valueOf(i));
            Neuron postML = network.getNeuronByLabel("postML" +
String.valueOf(i));
            Neuron postLin = network.getNeuronByLabel("postLin" +
String.valueOf(i));

            // trace du neurone excitateur
            PotentialProducer neuronProducerPre =
NetworkComponent.getNeuronProducer(networkComponent, pre ,
"getActivation");
            PotentialConsumer timeSeriesConsumerPre =
chartPre.getPotentialConsumers().get(i);
            workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerPre, timeSeriesConsumerPre));

            // trace du neurone Morris-Lecar
            PotentialProducer neuronProducerML =
NetworkComponent.getNeuronProducer(networkComponent, postML ,
"getActivation");
            PotentialConsumer timeSeriesConsumerML =
chartML.getPotentialConsumers().get(i);
            workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML, timeSeriesConsumerML));

            // trace du neurone Lineaire
            PotentialProducer neuronProducerLin =
```

```
NetworkComponent.getNeuronProducer(networkComponent, postLin ,
"getActivation");
           PotentialConsumer timeSeriesConsumerLin =
chartLin.getPotentialConsumers().get(i);
           workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerLin, timeSeriesConsumerLin));
      }
}

main();
```

```
// Parametres utiles

//Parameters reseau
int nRowEc = 3; // Taille d un cote de l ecran carre
int nRow = 3; // Taille des arretes du reservoir cubique
int distNeuron = 40; // Parametre de la distance entre deux
neurones, permet d aerer le reseau
double timeStep = 0.1; // pas de temps du reseau

double tauxCo = 0.5 ; // Taux de connexion entre l ecran et le
reservoir
double tauxInhib = 0.4; // Pourcentage de connexions inhibitrices
dans le reservoir
double wectores = 1; // Poids synaptique des connexions entre l
ecran et le reservoir
double wrestores = 1; // Poids synaptique des connexions internes au
reservoir

// Parametres Neurone Morris-Lecar
     //Il suffit de commenter les paramÃ¨tres pour que leur valeur
soit la valeur standard de Simbrain
MorrisLecarRule generalRule = new MorrisLecarRule();
//generalRule.setG_Ca(6.9);
//generalRule.setG_K(10);
//generalRule.setG_L(0.9);
//generalRule.setvRest_Ca(100);
//generalRule.setvRest_k(-100);
//generalRule.setvRest_L(-100);
//generalRule.setcMembrane(0.05);
//generalRule.setV_m1(0);
//generalRule.setV_m2(40);
//generalRule.setV_w1(0);
//generalRule.setV_w2(40);
//generalRule.setPhi(0.3);
generalRule.setI_bg(0);  // courant d excitation du neurone toujours
present pendant la simulation
generalRule.setThreshold(0);
//generalRule.setAddNoise(false);
generalRule.setInputType(InputType.SYNAPTIC); // Les entrees soient
provoquees par les spikes uniquement

// Parametres neurones lineaires du reseau ( servent de sources de
courant)
LinearRule pixel = new LinearRule();
pixel.setLowerBound(0);
pixel.setUpperBound(100);


//Parameters STDP
double taum = 40.0/3;
double taup = 40.0/3;
double learningrate = 0.25 ;
double wp = 1;
```

```
double wm = 1;
double restoormax = 1;
double restoormin = -1;

// Parameters rise and decay synapses ecran to reservoir
double tau_ectores = 5;
double maxResp_ectores = 30;

// Parameters rise and decay synapses reservoir to reservoir
double tau_restores = 5;
double maxResp_restores = 30;

// Parameters rise and decay synapses reservoir to orientation
double tau_restoor = 5;
double maxResp_restoor = 15;

// Parameters rise and decay synapses reservoir to orientation
double tau_inhiblator = 5;
double maxResp_inhiblator = 30;

// Parameters rise and decay synapses reservoir to orientation
double tau_inhiblatsup = 5;
double maxResp_inhiblatsup = 30;


void main(){
     workspace.clearWorkspace();
     buildNetwork();
//   loadCustomUpdater();
     addTimeSeries();
     addButtons();

}

void buildNetwork() {
    // Parameters

    // Set up network
     NetworkComponent networkComponent = new
NetworkComponent("ReservoirComputing");
     workspace.addWorkspaceComponent(networkComponent);
     Network network = networkComponent.getNetwork();
     network.setTimeStep(timeStep);


     // Creation Ecran et excitateurs de l ecran, geometrie carree
     for (int i = 0; i < nRowEc ; i++) {
          for (int j = 0; j < nRowEc; j++) {

               // Creation neurone excitateur
               Neuron n = new Neuron(network, pixel.deepCopy());
               n.setLocation(  (j-0.75) * distNeuron,   i *
distNeuron);
//             n.setClamped(true);     // On enlÃ¨ve cette ligne
pour pouvoir utiliser le DAtaworld dans l apprentissage
               n.setLabel("ex" + String.valueOf( nRowEc * i + j )
```

```
);
                network.addNeuron(n);

                // Creation neurone de l ecran
                Neuron nec = new Neuron(network,
generalRule.deepCopy());
                nec.setLocation( j * distNeuron, nRowEc *distNeuron
+ i * distNeuron);
                nec.setLabel("ec" + String.valueOf( nRowEc * i + j )
);
                nec.getUpdateRule().setI_bg(0);
                network.addNeuron(nec);

                // connexion excitateur ecran
                network.addSynapse(new Synapse( n, nec, 1));
          }
     }



     // Reservoir

     //RepÃ¨re Position Reservoir
     Neuron topRight0 = network.getNeuronByLabel("ec" +
String.valueOf(nRowEc-1));
     int xCouche1 = topRight0.getX() + 100 ;
     //topRight0.setLabel(String.valueOf(100 )); //test pour savoir
si le bon neurone a Ã©tÃ© choisi

     //CrÃ©ation Neurones Reservoir  geometrie cubique
     // Les neurones sont numerotes de gauche a droite, de haut et
bas et la face de devant est la face la plus basse
     for (int i = 0; i < nRow ; i++) {
          for (int j = 0; j < nRow; j++) {
                for (int k = 0; k < nRow; k++){

                     // Creation du neurone du reservoir positionne
a la k eme colonne, j eme ligne et i eme face
                     Neuron n = new Neuron(network,
generalRule.deepCopy());
                     n.setLocation( xCouche1 + k * distNeuron * 3 +
i * distNeuron * 2, j * distNeuron * 3 - i * distNeuron );
                     n.setLabel( "res" + String.valueOf( nRow * j +
k + nRow * nRow * i));
                     network.addNeuron(n);
     }    }    }

     // Synapses Ecran to Reservoir
     for (int i = 0; i < nRow; i++){
          for (int j = 0; j < nRow; j++){
                for (int k = 0 ; k < nRow; k++){

                     // recuperat neurone du resrvoir
                     Neuron nReservoir = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * i + nRow * j + k )));
```

```
                    // Connexion de probabilite tauxCo avec chacun
des neurones de l ecran
                    for (int i2 = 0; i2 < nRowEc; i2++){
                         for (int j2 = 0; j2 < nRowEc; j2++){
                              double a = Math.random();
                              if (a < tauxCo){
                                   Neuron pixel =
network.getNeuronByLabel("ec" + String.valueOf( nRowEc * i2 + j2 ));
                                   Synapse ectores = new
Synapse(pixel, nReservoir, wectores) ;
                                   ectores.setSpikeResponder( new
RiseAndDecay());

     ectores.getSpikeResponder().setMaximumResponse(maxResp_ectores)
;

     ectores.getSpikeResponder().setTimeConstant(tau_ectores);
                                   network.addSynapse(ectores);
     }    }    }    }    }    }

     // interConnexion Reservoir     avec tauxInhib pourcents d
inhibition

     for (int i = 0; i < nRow; i++){
          for (int j = 0; j < nRow; j++){
               for (int k = 0 ; k < nRow; k++){

                    Neuron nReservoir = network.getNeuronByLabel((
"res" + String.valueOf( nRow * j + k + nRow * nRow * i)));
                    i_prec = i - 1;
                    i_suiv = i + 1;
                    j_prec = j - 1;
                    j_suiv = j + 1;
                    k_prec = k - 1;
                    k_suiv = k + 1;

                    if (i == 0){i_prec = i_suiv;}
                    if (j == 0){j_prec = j_suiv;}
                    if (k == 0){k_prec = k_suiv;}
                    if (i == nRow - 1){i_suiv = i_prec;}
                    if (j == nRow - 1){j_suiv = j_prec;}
                    if (k == nRow - 1){k_suiv = k_prec;}

                    a1 = Math.random();
                    a2 = Math.random();
                    a3 = Math.random();
                    a4 = Math.random();
                    a5 = Math.random();
                    a6 = Math.random();


                    voisin1 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * i + nRow * j + (k_prec))));
                    voisin2 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * i + nRow * j + (k_suiv))));
                    voisin3 = network.getNeuronByLabel(( "res" +
```

```
String.valueOf( nRow * nRow * i + nRow * (j_prec) + k)));
                        voisin4 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * i + nRow * (j_suiv) + k)));
                        voisin5 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * (i_prec) + nRow * j + k)));
                        voisin6 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * (i_suiv) + nRow * j + k)));

                        if (a1 < tauxInhib){
                              Synapse restores = new Synapse(
nReservoir, voisin1, -wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }else{
                              Synapse restores = new Synapse(
nReservoir, voisin1, wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }

                        if (a2 < tauxInhib){
                              Synapse restores = new Synapse(
nReservoir, voisin2, -wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }else{
                              Synapse restores = new Synapse(
nReservoir, voisin2, wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }
```

```
                    if (a3 < tauxInhib){
                         Synapse restores = new Synapse(
nReservoir, voisin3, -wrestores) ;
                         restores.setSpikeResponder( new
RiseAndDecay());

     restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

     restores.getSpikeResponder().setTimeConstant(tau_restores);
                         network.addSynapse(restores);
                    }else{
                         Synapse restores = new Synapse(
nReservoir, voisin3, wrestores) ;
                         restores.setSpikeResponder( new
RiseAndDecay());

     restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

     restores.getSpikeResponder().setTimeConstant(tau_restores);
                         network.addSynapse(restores);
                    }

                    if (a4 < tauxInhib){
                         Synapse restores = new Synapse(
nReservoir, voisin4, -wrestores) ;
                         restores.setSpikeResponder( new
RiseAndDecay());

     restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

     restores.getSpikeResponder().setTimeConstant(tau_restores);
                         network.addSynapse(restores);
                    }else{
                         Synapse restores = new Synapse(
nReservoir, voisin4, wrestores) ;
                         restores.setSpikeResponder( new
RiseAndDecay());

     restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

     restores.getSpikeResponder().setTimeConstant(tau_restores);
                         network.addSynapse(restores);
                    }
                    if (a5 < tauxInhib){
                         Synapse restores = new Synapse(
nReservoir, voisin5, -wrestores) ;
                         restores.setSpikeResponder( new
RiseAndDecay());

     restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);
```

```
      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }else{
                              Synapse restores = new Synapse(
nReservoir, voisin5, wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }

                        if (a6 < tauxInhib){
                              Synapse restores = new Synapse(
nReservoir, voisin6, -wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }else{
                              Synapse restores = new Synapse(
nReservoir, voisin6, wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }
                  }
            }
      }


      // Neurones de sortie et superviseurs

      //Reperes position
      Neuron topRight1 = network.getNeuronByLabel("res" +
String.valueOf(nRow-1));
      int xCouche2 = topRight1.getX() + 100 * nRow ;
      int yCouche2 = 0;

      //Creation Neurones de sortie et superviseurs et connexion
      for (int i = 0; i < 4; i++) {
            Neuron nOr = new Neuron(network, generalRule.deepCopy());
            Neuron sup = new Neuron(network, pixel.deepCopy());
//          sup.setClamped(true); // On enlÃ¨ve cette ligne pour
```

```
pouvoir utiliser le dataworld dans l apprentissage


            nOr.setLocation( xCouche2 , yCouche2 + i * distNeuron);
            nOr.setLabel( "or" + String.valueOf(i));

            sup.setLocation( xCouche2 + 100  , yCouche2 + i *
distNeuron);
            sup.setLabel( "sup" + String.valueOf(i));

            network.addNeuron(sup);
            network.addNeuron(nOr);

            network.addSynapse(new Synapse( sup, nOr, wrestores));
      }


      // Synapses Reservoir to Orientations
      for (int i = 0; i < (nRow); i++) {
            for (int j = 0; j < (nRow); j++) {
                  for (int k = 0; k<(nRow); k++){
                        Neuron nReservoir = network.getNeuronByLabel((
"res" + String.valueOf( nRow * j + k + nRow * nRow * i)));

                        for (int l = 0; l<4; l++){
                              Neuron nOr = network.getNeuronByLabel((
"or" + String.valueOf(l)));
                              Synapse restoor = new Synapse(nReservoir,
nOr, 0);
                              restoor.setSpikeResponder( new
RiseAndDecay());

      restoor.getSpikeResponder().setMaximumResponse(maxResp_restoor)
;

      restoor.getSpikeResponder().setTimeConstant(tau_restoor);


                              restoor.setLearningRule(new STDPRule());

      restoor.getLearningRule().setTau_minus(taum);

      restoor.getLearningRule().setTau_plus(taup);

      restoor.getLearningRule().setLearningRate(learningrate);
                              restoor.getLearningRule().setW_plus(wp);

                              restoor.getLearningRule().setW_minus(wm);

                              restoor.setUpperBound(restoormax);
                              restoor.setLowerBound(restoormin);


                              network.addSynapse( restoor);

                        }
```

```
                }
            }
        }

        // Inhibition LatÃ©rale or to or
        for (int i = 0; i < 4; i++) {
            Neuron inhibiteur = network.getNeuronByLabel(( "or" +
String.valueOf(i)));
            for (int j = 0; j < 4; j++) {
                if( i != j){
                    Neuron inhibe = network.getNeuronByLabel(( "or"
+ String.valueOf(j)));

                    Synapse inhiblator = new Synapse(inhibiteur,
inhibe, -1);
                    inhiblator.setSpikeResponder( new
RiseAndDecay());

        inhiblator.getSpikeResponder().setMaximumResponse(maxResp_inhib
lator);

        inhiblator.getSpikeResponder().setTimeConstant(tau_inhiblator);

                    network.addSynapse( inhiblator);

                }
            }
        }


        // Inhibition LatÃ©rale sup to or
        for (int i = 0; i < 4; i++) {
            Neuron inhibiteur = network.getNeuronByLabel(( "sup" +
String.valueOf(i)));
            for (int j = 0; j < 4; j++) {
                if( i != j){
                    Neuron inhibe = network.getNeuronByLabel(( "or"
+ String.valueOf(j)));

                    Synapse inhiblatsup = new Synapse(inhibiteur,
inhibe, -1);
                    inhiblatsup.setSpikeResponder( new
RiseAndDecay());

        inhiblatsup.getSpikeResponder().setMaximumResponse(maxResp_inhi
blatsup);

        inhiblatsup.getSpikeResponder().setTimeConstant(tau_inhiblatsup
);
                    network.addSynapse( inhiblatsup);
                }
            }
        }
        }
```

```
void addTimeSeries() {

      // Recuperation des elements utiles
    NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent("ReservoirComputing"));
      Network network = networkComponent.getNetwork();

      // Informations sur les neurones de sortie

      // Graphe des neurones de sortie
    TimeSeriesPlotComponent chartML = new
TimeSeriesPlotComponent("Test spikeresponder", 4);
      chartML.getModel().setWindowSize(5000);
    workspace.addWorkspaceComponent(chartML);

desktop.getDesktopComponent(chartML).getParentFrame().setBounds(500,
0,800,350);

      for (i = 0; i < 4; i++){

            //Recuperation neurones de sortie captant les
orientations
            Neuron or = network.getNeuronByLabel("or" +
String.valueOf(i));

            // Trace du potentiel du neurone de sortie
            PotentialProducer neuronProducerOr0 =
NetworkComponent.getNeuronProducer(networkComponent, or ,
"getActivation");
            PotentialConsumer timeSeriesConsumerOr0 =
chartML.getPotentialConsumers().get(i);
            workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerOr0, timeSeriesConsumerOr0));


            // Suivi des poids de la stdp

//          List synor = or.getFanIn();
//          int len = synor.size() ;
//
//          // Graphe des poids synaptique du neurone de sortie
//          TimeSeriesPlotComponent chartstdp = new
TimeSeriesPlotComponent("Apprentissage orientation" +
String.valueOf(i), 27);
//          chartstdp.getModel().setWindowSize(5000);
//          workspace.addWorkspaceComponent(chartstdp);
//    //
      desktop.getDesktopComponent(chartstdp).getParentFrame().setBoun
ds(500,350,800,350);
//
//
//          for (j = 0; j < 27; j++){
//
//          Neuron res = network.getNeuronByLabel("res" +
String.valueOf(j));
//          PotentialProducer synapseProducer =
```

```
NetworkComponent.getSynapseProducer(networkComponent,
network.getSynapse(res, or),"getStrength");
//         PotentialConsumer timeSeriesConsumer =
chartstdp.getPotentialConsumers().get(j);
//         workspace.getCouplingManager().addCoupling(new
Coupling(synapseProducer, timeSeriesConsumer));
//         }
      }

      // Graphe du comportement du reservoir
      TimeSeriesPlotComponent chartreservoir = new
TimeSeriesPlotComponent("Reservoir", 27);
      chartreservoir.getModel().setWindowSize(250);
      workspace.addWorkspaceComponent(chartreservoir);
//
      desktop.getDesktopComponent(chartreservoir).getParentFrame().se
tBounds(500,700,800,350);

      for (j = 0; j < 27; j++){
            Neuron res = network.getNeuronByLabel("res" +
String.valueOf(j));
            PotentialProducer neuronProducer =
NetworkComponent.getNeuronProducer(networkComponent,
res,"getActivation");
            PotentialConsumer timeSeriesConsumer2 =
chartreservoir.getPotentialConsumers().get(j);
            workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducer, timeSeriesConsumer2));
      }

}



void addButtons() {

      // Recuperation des elements utiles
    NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent("ReservoirComputing"));
      Network network = networkComponent.getNetwork();

      Neuron ex0 = network.getNeuronByLabel("ex0");
      Neuron ex1 = network.getNeuronByLabel("ex1");
      Neuron ex2 = network.getNeuronByLabel("ex2");
      Neuron ex3 = network.getNeuronByLabel("ex3");
      Neuron ex4 = network.getNeuronByLabel("ex4");
      Neuron ex5 = network.getNeuronByLabel("ex5");
      Neuron ex6 = network.getNeuronByLabel("ex6");
      Neuron ex7 = network.getNeuronByLabel("ex7");
      Neuron ex8 = network.getNeuronByLabel("ex8");

      Neuron ec0 = network.getNeuronByLabel("ec0");
      Neuron ec1 = network.getNeuronByLabel("ec1");
      Neuron ec2 = network.getNeuronByLabel("ec2");
      Neuron ec3 = network.getNeuronByLabel("ec3");
      Neuron ec4 = network.getNeuronByLabel("ec4");
```

```
      Neuron ec5 = network.getNeuronByLabel("ec5");
      Neuron ec6 = network.getNeuronByLabel("ec6");
      Neuron ec7 = network.getNeuronByLabel("ec7");
      Neuron ec8 = network.getNeuronByLabel("ec8");

      Neuron sup0 = network.getNeuronByLabel("sup0");
      Neuron sup1 = network.getNeuronByLabel("sup1");
      Neuron sup2 = network.getNeuronByLabel("sup2");
      Neuron sup3 = network.getNeuronByLabel("sup3");

      Neuron or0 = network.getNeuronByLabel("or0");
      Neuron or1 = network.getNeuronByLabel("or1");
      Neuron or2 = network.getNeuronByLabel("or2");
      Neuron or3 = network.getNeuronByLabel("or3");


    // Start making the buttons!
    JInternalFrame internalFrame = new JInternalFrame("Parametrage",
true, true);
    LabelledItemPanel panel = new LabelledItemPanel();


      // Bouton mettant les synapses or Ã  0 pour dÃ©buter l
apprentissage sans interference
    JButton buttonPursuer = new JButton("cut off synapses");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (i = 0; i < 4; i++){
                     Neuron nOr = network.getNeuronByLabel("or" +
String.valueOf(i));
                     for (j = 0; j < 27; j++){
                          Neuron res =
network.getNeuronByLabel("res" + String.valueOf(j));
                          network.getSynapse(res,
nOr).setStrength(0);
                     }
                }
            workspace.iterate();
        }});
    panel.addItem("Init Apprentissage", buttonPursuer);

      //This button stops the learning and freeze the synapses
    JButton buttonPursuer = new JButton("freeze all");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (i = 0; i < 4; i++){
                     Neuron nOr = network.getNeuronByLabel("or" +
String.valueOf(i));
                     for (j = 0; j < 27; j++){
                          Neuron res =
network.getNeuronByLabel("res" + String.valueOf(j));
                          network.getSynapse(res,
nOr).setFrozen(true);
                     }
                }
            workspace.iterate();
```

```
        }});
//    panel.addItem("Finish learning", buttonPursuer);

           //This button let start again the learning by unfrozing
synapses
    JButton buttonPursuer = new JButton("defreeze");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (i = 0; i < 4; i++){
                     Neuron nOr = network.getNeuronByLabel("or" +
String.valueOf(i));
                     for (j = 0; j < 27; j++){
                          Neuron res =
network.getNeuronByLabel("res" + String.valueOf(j));
                          network.getSynapse(res,
nOr).setFrozen(false);
                     }
                }
            workspace.iterate();
        }});
 //   panel.addItem("Restart learning", buttonPursuer);


     JButton buttonAvoider = new JButton("New Repartition !");
    buttonAvoider.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (int i = 0; i < nRow; i++){
                     for (int j = 0; j < nRow; j++){
                          for (int k = 0 ; k < nRow; k++){

                               Neuron nReservoir =
network.getNeuronByLabel(( "res" + String.valueOf( nRow * j + k +
nRow * nRow * i)));
                               i_prec = i - 1;
                               i_suiv = i + 1;
                               j_prec = j - 1;
                               j_suiv = j + 1;
                               k_prec = k - 1;
                               k_suiv = k + 1;

                               if (i == 0){i_prec = i_suiv;}
                               if (j == 0){j_prec = j_suiv;}
                               if (k == 0){k_prec = k_suiv;}
                               if (i == nRow - 1){i_suiv = i_prec;}
                               if (j == nRow - 1){j_suiv = j_prec;}
                               if (k == nRow - 1){k_suiv = k_prec;}

                               a1 = Math.random();
                               a2 = Math.random();
                               a3 = Math.random();
                               a4 = Math.random();
                               a5 = Math.random();
                               a6 = Math.random();


                               voisin1 = network.getNeuronByLabel((
```

```
"res" + String.valueOf( nRow * nRow * i + nRow * j + (k_prec))));
                                    voisin2 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * i + nRow * j + (k_suiv))));
                                    voisin3 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * i + nRow * (j_prec) + k)));
                                    voisin4 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * i + nRow * (j_suiv) + k)));
                                    voisin5 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * (i_prec) + nRow * j + k)));

                                    voisin6 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * (i_suiv) + nRow * j + k)));


                                    if (a1 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin1).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin1).setStrength(wectores);}

                                    if (a2 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin2).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin2).setStrength(wectores);}

                                    if (a3 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin3).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin3).setStrength(wectores);}

                                    if (a4 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin4).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin4).setStrength(wectores);}

                                    if (a5 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin5).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin5).setStrength(wectores);}

                                    if (a6 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin6).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin6).setStrength(wectores);}
                                }
                            }
                        }
                        workspace.iterate();
                }});
            panel.addItem("Change Reservoir ", buttonAvoider);
```

```
    JButton buttonReverseWeights = new JButton("Do it!");
    buttonReverseWeights.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (i = 0; i < Double.parseDouble(niter.getText());
i++){
             workspace.iterate();
                 }
            }
      });
    panel.addItem("Run", buttonReverseWeights);

    JTextField niter = new JTextField("1000");
    panel.addItem("iterations", niter);
      JTextField buttonInhib = new
JTextField(String.valueOf(tauxInhib));
    panel.addItem("tauxInhib reservoir", buttonInhib);

    // Set up Frame
    internalFrame.setLocation(0,500);
    internalFrame.getContentPane().add(panel);
    internalFrame.setVisible(true);
    internalFrame.pack();
    desktop.addInternalFrame(internalFrame);

}


main();
```

IRCICA_ReservoirComputingSpherique

```
// Parametres utiles

//Parameters reseau
int nRowEc = 3; // Taille d un cote de l ecran carre
int nRow = 3; // Taille des arretes du reservoir cubique
int distNeuron = 40; // Parametre de la distance entre deux
neurones, permet d aerer le reseau
double timeStep = 0.1; // pas de temps du reseau

double tauxCo = 1.0/3 ; // Taux de connexion entre l ecran et le
reservoir
double tauxInhib = 0.2; // Pourcentage de connexions inhibitrices
dans le reservoir
double wectores = 1; // Poids synaptique des connexions entre l
ecran et le reservoir
double wrestores = 1; // Poids synaptique des connexions internes au
reservoir

// Parametres Neurone Morris-Lecar
     //Il suffit de commenter les paramÃ¨tres pour que leur valeur
soit la valeur standard de Simbrain
MorrisLecarRule generalRule = new MorrisLecarRule();
//generalRule.setG_Ca(6.9);
//generalRule.setG_K(10);
//generalRule.setG_L(0.9);
//generalRule.setvRest_Ca(100);
//generalRule.setvRest_k(-100);
//generalRule.setvRest_L(-100);
//generalRule.setcMembrane(0.05);
//generalRule.setV_m1(0);
//generalRule.setV_m2(40);
//generalRule.setV_w1(0);
//generalRule.setV_w2(40);
//generalRule.setPhi(0.3);
generalRule.setI_bg(0);  // courant d excitation du neurone toujours
present pendant la simulation
generalRule.setThreshold(0);
//generalRule.setAddNoise(false);
generalRule.setInputType(InputType.SYNAPTIC); // Les entrees soient
provoquees par les spikes uniquement

// Parametres neurones lineaires du reseau ( servent de sources de
courant)
LinearRule pixel = new LinearRule();
pixel.setLowerBound(0);
pixel.setUpperBound(100);


//Parameters STDP
double taum = 40.0/3;
double taup = 40.0/3;
double learningrate = 0.25 ;
double wp = 1;
double wm = 1;
```

```
double restoormax = 1;
double restoormin = -1;

// Parameters rise and decay synapses ecran to reservoir
double tau_ectores = 5;
double maxResp_ectores = 30;

// Parameters rise and decay synapses reservoir to reservoir
double tau_restores = 5;
double maxResp_restores = 30;

// Parameters rise and decay synapses reservoir to orientation
double tau_restoor = 5;
double maxResp_restoor = 15;

// Parameters rise and decay synapses reservoir to orientation
double tau_inhiblator = 5;
double maxResp_inhiblator = 30;

// Parameters rise and decay synapses reservoir to orientation
double tau_inhiblatsup = 5;
double maxResp_inhiblatsup = 30;


void main(){
     workspace.clearWorkspace();
     buildNetwork();
//   loadCustomUpdater();
     addTimeSeries();
     addButtons();

}

void buildNetwork() {
    // Parameters

    // Set up network
     NetworkComponent networkComponent = new
NetworkComponent("ReservoirComputing");
     workspace.addWorkspaceComponent(networkComponent);
     Network network = networkComponent.getNetwork();
     network.setTimeStep(timeStep);


     // Creation Ecran et excitateurs de l ecran, geometrie carree
     for (int i = 0; i < nRowEc ; i++) {
          for (int j = 0; j < nRowEc; j++) {

               // Creation neurone excitateur
               Neuron n = new Neuron(network, pixel.deepCopy());
               n.setLocation(  (j-0.75) * distNeuron,   i *
distNeuron);
//             n.setClamped(true);     // On enlÃ¨ve cette ligne
pour pouvoir utiliser le DAtaworld dans l apprentissage
               n.setLabel("ex" + String.valueOf( nRowEc * i + j )
);
```

```
                network.addNeuron(n);

                // Creation neurone de l ecran
                Neuron nec = new Neuron(network,
generalRule.deepCopy());
                nec.setLocation( j * distNeuron, nRowEc *distNeuron
+ i * distNeuron);
                nec.setLabel("ec" + String.valueOf( nRowEc * i + j )
);
                nec.getUpdateRule().setI_bg(0);
                network.addNeuron(nec);

                // connexion excitateur ecran
                network.addSynapse(new Synapse( n, nec, 1));
          }
     }



     // Reservoir

     //RepÃ¨re Position Reservoir
     Neuron topRight0 = network.getNeuronByLabel("ec" +
String.valueOf(nRowEc-1));
     int xCouche1 = topRight0.getX() + 100 ;
     //topRight0.setLabel(String.valueOf(100 )); //test pour savoir
si le bon neurone a Ã©tÃ© choisi

     //CrÃ©ation Neurones Reservoir  geometrie cubique
     // Les neurones sont numerotes de gauche a droite, de haut et
bas et la face de devant est la face la plus basse
     for (int i = 0; i < nRow ; i++) {
          for (int j = 0; j < nRow; j++) {
                for (int k = 0; k < nRow; k++){

                     // Creation du neurone du reservoir positionne
a la k eme colonne, j eme ligne et i eme face
                     Neuron n = new Neuron(network,
generalRule.deepCopy());
                     n.setLocation( xCouche1 + k * distNeuron * 3 +
i * distNeuron * 2, j * distNeuron * 3 - i * distNeuron );
                     n.setLabel( "res" + String.valueOf( nRow * j +
k + nRow * nRow * i));
                     network.addNeuron(n);
     }    }    }

     // Synapses Ecran to Reservoir
     for (int i = 0; i < nRow; i++){
          for (int j = 0; j < nRow; j++){
                for (int k = 0 ; k < nRow; k++){

                     // recuperat neurone du resrvoir
                     Neuron nReservoir = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * i + nRow * j + k )));

                     // Connexion de probabilite tauxCo avec chacun
```

```
des neurones de l ecran
                    for (int i2 = 0; i2 < nRowEc; i2++){
                         for (int j2 = 0; j2 < nRowEc; j2++){
                              double a = Math.random();
                              if (a < tauxCo){
                                   Neuron pixel =
network.getNeuronByLabel("ec" + String.valueOf( nRowEc * i2 + j2 ));
                                   Synapse ectores = new
Synapse(pixel, nReservoir, wectores) ;
                                   ectores.setSpikeResponder( new
RiseAndDecay());

     ectores.getSpikeResponder().setMaximumResponse(maxResp_ectores)
;

     ectores.getSpikeResponder().setTimeConstant(tau_ectores);
                                   network.addSynapse(ectores);
     }     }     }     }     }     }

     // interConnexion Reservoir     avec tauxInhib pourcents d
inhibition
     for (int i = 0; i < nRow; i++){
          for (int j = 0; j < nRow; j++){
               for (int k = 0 ; k < nRow; k++){

                    Neuron nReservoir = network.getNeuronByLabel((
"res" + String.valueOf( nRow * j + k + nRow * nRow * i)));
                    i_prec = i - 1;
                    i_suiv = i + 1;
                    j_prec = j - 1;
                    j_suiv = j + 1;
                    k_prec = k - 1;
                    k_suiv = k + 1;

                    if (i == 0){i_prec = nRow - 1;}
                    if (j == 0){j_prec = nRow - 1;}
                    if (k == 0){k_prec = nRow - 1;}
                    if (i == nRow - 1){i_suiv = 0;}
                    if (j == nRow - 1){j_suiv = 0;}
                    if (k == nRow - 1){k_suiv = 0;}

                    a1 = Math.random();
                    a2 = Math.random();
                    a3 = Math.random();
                    a4 = Math.random();
                    a5 = Math.random();
                    a6 = Math.random();


                    voisin1 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * i + nRow * j + (k_prec))));
                    voisin2 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * i + nRow * j + (k_suiv))));
                    voisin3 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * i + nRow * (j_prec) + k)));
                    voisin4 = network.getNeuronByLabel(( "res" +
```

```
String.valueOf( nRow * nRow * i + nRow * (j_suiv) + k)));
                    voisin5 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * (i_prec) + nRow * j + k)));
                    voisin6 = network.getNeuronByLabel(( "res" +
String.valueOf( nRow * nRow * (i_suiv) + nRow * j + k)));

                    if (a1 < tauxInhib){
                         Synapse restores = new Synapse(
nReservoir, voisin1, -wrestores) ;
                         restores.setSpikeResponder( new
RiseAndDecay());

     restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

     restores.getSpikeResponder().setTimeConstant(tau_restores);
                         network.addSynapse(restores);
                    }else{
                         Synapse restores = new Synapse(
nReservoir, voisin1, wrestores) ;
                         restores.setSpikeResponder( new
RiseAndDecay());

     restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

     restores.getSpikeResponder().setTimeConstant(tau_restores);
                         network.addSynapse(restores);
                    }

                    if (a2 < tauxInhib){
                         Synapse restores = new Synapse(
nReservoir, voisin2, -wrestores) ;
                         restores.setSpikeResponder( new
RiseAndDecay());

     restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

     restores.getSpikeResponder().setTimeConstant(tau_restores);
                         network.addSynapse(restores);
                    }else{
                         Synapse restores = new Synapse(
nReservoir, voisin2, wrestores) ;
                         restores.setSpikeResponder( new
RiseAndDecay());

     restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

     restores.getSpikeResponder().setTimeConstant(tau_restores);
                         network.addSynapse(restores);
                    }

                    if (a3 < tauxInhib){
                         Synapse restores = new Synapse(
```

```
nReservoir, voisin3, -wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }else{
                              Synapse restores = new Synapse(
nReservoir, voisin3, wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }

                        if (a4 < tauxInhib){
                              Synapse restores = new Synapse(
nReservoir, voisin4, -wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }else{
                              Synapse restores = new Synapse(
nReservoir, voisin4, wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }
                        if (a5 < tauxInhib){
                              Synapse restores = new Synapse(
nReservoir, voisin5, -wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
```

```
                        }else{
                              Synapse restores = new Synapse(
nReservoir, voisin5, wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }

                        if (a6 < tauxInhib){
                              Synapse restores = new Synapse(
nReservoir, voisin6, -wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }else{
                              Synapse restores = new Synapse(
nReservoir, voisin6, wrestores) ;
                              restores.setSpikeResponder( new
RiseAndDecay());

      restores.getSpikeResponder().setMaximumResponse(maxResp_restore
s);

      restores.getSpikeResponder().setTimeConstant(tau_restores);
                              network.addSynapse(restores);
                        }
                  }
            }
      }


      // Neurones de sortie et superviseurs

      //Reperes position
      Neuron topRight1 = network.getNeuronByLabel("res" +
String.valueOf(nRow-1));
      int xCouche2 = topRight1.getX() + 100 * nRow ;
      int yCouche2 = 0;

      //Creation Neurones de sortie et superviseurs et connexion
      for (int i = 0; i < 4; i++) {
            Neuron nOr = new Neuron(network, generalRule.deepCopy());
            Neuron sup = new Neuron(network, pixel.deepCopy());
//          sup.setClamped(true); // On enlÃ¨ve cette ligne pour
pouvoir utiliser le dataworld dans l apprentissage
```

```
            nOr.setLocation( xCouche2 , yCouche2 + i * distNeuron);
            nOr.setLabel( "or" + String.valueOf(i));

            sup.setLocation( xCouche2 + 100  , yCouche2 + i *
distNeuron);
            sup.setLabel( "sup" + String.valueOf(i));

            network.addNeuron(sup);
            network.addNeuron(nOr);

            network.addSynapse(new Synapse( sup, nOr, wrestores));
      }


      // Synapses Reservoir to Orientations
      for (int i = 0; i < (nRow); i++) {
            for (int j = 0; j < (nRow); j++) {
                  for (int k = 0; k<(nRow); k++){
                        Neuron nReservoir = network.getNeuronByLabel((
"res" + String.valueOf( nRow * j + k + nRow * nRow * i)));

                        for (int l = 0; l<4; l++){
                              Neuron nOr = network.getNeuronByLabel((
"or" + String.valueOf(l)));
                              Synapse restoor = new Synapse(nReservoir,
nOr, 0);
                              restoor.setSpikeResponder( new
RiseAndDecay());

      restoor.getSpikeResponder().setMaximumResponse(maxResp_restoor)
;

      restoor.getSpikeResponder().setTimeConstant(tau_restoor);


                              restoor.setLearningRule(new STDPRule());

      restoor.getLearningRule().setTau_minus(taum);

      restoor.getLearningRule().setTau_plus(taup);

      restoor.getLearningRule().setLearningRate(learningrate);
                              restoor.getLearningRule().setW_plus(wp);

                              restoor.getLearningRule().setW_minus(wm);

                              restoor.setUpperBound(restoormax);
                              restoor.setLowerBound(restoormin);


                              network.addSynapse( restoor);

                        }
                  }
            }
```

```
	}

	// Inhibition LatÃ©rale or to or
	for (int i = 0; i < 4; i++) {
		Neuron inhibiteur = network.getNeuronByLabel(( "or" +
String.valueOf(i)));
		for (int j = 0; j < 4; j++) {
			if( i != j){
				Neuron inhibe = network.getNeuronByLabel(( "or"
+ String.valueOf(j)));

				Synapse inhiblator = new Synapse(inhibiteur,
inhibe, -1);
				inhiblator.setSpikeResponder( new
RiseAndDecay());

	inhiblator.getSpikeResponder().setMaximumResponse(maxResp_inhib
lator);

	inhiblator.getSpikeResponder().setTimeConstant(tau_inhiblator);

				network.addSynapse( inhiblator);

			}
		}
	}


	// Inhibition LatÃ©rale sup to or
	for (int i = 0; i < 4; i++) {
		Neuron inhibiteur = network.getNeuronByLabel(( "sup" +
String.valueOf(i)));
		for (int j = 0; j < 4; j++) {
			if( i != j){
				Neuron inhibe = network.getNeuronByLabel(( "or"
+ String.valueOf(j)));

				Synapse inhiblatsup = new Synapse(inhibiteur,
inhibe, -1);
				inhiblatsup.setSpikeResponder( new
RiseAndDecay());

	inhiblatsup.getSpikeResponder().setMaximumResponse(maxResp_inhi
blatsup);

	inhiblatsup.getSpikeResponder().setTimeConstant(tau_inhiblatsup
);
				network.addSynapse( inhiblatsup);
			}
		}
	}
	}


void addTimeSeries() {
```

```
      // Recuperation des elements utiles
    NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent("ReservoirComputing"));
      Network network = networkComponent.getNetwork();

      // Informations sur les neurones de sortie

      // Graphe des neurones de sortie
    TimeSeriesPlotComponent chartML = new
TimeSeriesPlotComponent("Test spikeresponder", 4);
      chartML.getModel().setWindowSize(5000);
    workspace.addWorkspaceComponent(chartML);

desktop.getDesktopComponent(chartML).getParentFrame().setBounds(500,
0,800,350);

      for (i = 0; i < 4; i++){

            //Recuperation neurones de sortie captant les
orientations
            Neuron or = network.getNeuronByLabel("or" +
String.valueOf(i));

            // Trace du potentiel du neurone de sortie
            PotentialProducer neuronProducerOr0 =
NetworkComponent.getNeuronProducer(networkComponent, or ,
"getActivation");
            PotentialConsumer timeSeriesConsumerOr0 =
chartML.getPotentialConsumers().get(i);
            workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerOr0, timeSeriesConsumerOr0));


            // Suivi des poids de la stdp

//          List synor = or.getFanIn();
//          int len = synor.size() ;
//
//          // Graphe des poids synaptique du neurone de sortie
//          TimeSeriesPlotComponent chartstdp = new
TimeSeriesPlotComponent("Apprentissage orientation" +
String.valueOf(i), 27);
//          chartstdp.getModel().setWindowSize(5000);
//          workspace.addWorkspaceComponent(chartstdp);
//    //
      desktop.getDesktopComponent(chartstdp).getParentFrame().setBoun
ds(500,350,800,350);
//
//
//          for (j = 0; j < 27; j++){
//
//          Neuron res = network.getNeuronByLabel("res" +
String.valueOf(j));
//          PotentialProducer synapseProducer =
NetworkComponent.getSynapseProducer(networkComponent,
network.getSynapse(res, or),"getStrength");
```

```
//         PotentialConsumer timeSeriesConsumer =
chartstdp.getPotentialConsumers().get(j);
//         workspace.getCouplingManager().addCoupling(new
Coupling(synapseProducer, timeSeriesConsumer));
//         }
      }

      // Graphe du comportement du reservoir
      TimeSeriesPlotComponent chartreservoir = new
TimeSeriesPlotComponent("Reservoir", 27);
      chartreservoir.getModel().setWindowSize(250);
      workspace.addWorkspaceComponent(chartreservoir);
//
      desktop.getDesktopComponent(chartreservoir).getParentFrame().se
tBounds(500,700,800,350);

      for (j = 0; j < 27; j++){
            Neuron res = network.getNeuronByLabel("res" +
String.valueOf(j));
            PotentialProducer neuronProducer =
NetworkComponent.getNeuronProducer(networkComponent,
res,"getActivation");
            PotentialConsumer timeSeriesConsumer2 =
chartreservoir.getPotentialConsumers().get(j);
            workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducer, timeSeriesConsumer2));
      }

}



void addButtons() {

      // Recuperation des elements utiles
    NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent("ReservoirComputing"));
      Network network = networkComponent.getNetwork();

      Neuron ex0 = network.getNeuronByLabel("ex0");
      Neuron ex1 = network.getNeuronByLabel("ex1");
      Neuron ex2 = network.getNeuronByLabel("ex2");
      Neuron ex3 = network.getNeuronByLabel("ex3");
      Neuron ex4 = network.getNeuronByLabel("ex4");
      Neuron ex5 = network.getNeuronByLabel("ex5");
      Neuron ex6 = network.getNeuronByLabel("ex6");
      Neuron ex7 = network.getNeuronByLabel("ex7");
      Neuron ex8 = network.getNeuronByLabel("ex8");

      Neuron ec0 = network.getNeuronByLabel("ec0");
      Neuron ec1 = network.getNeuronByLabel("ec1");
      Neuron ec2 = network.getNeuronByLabel("ec2");
      Neuron ec3 = network.getNeuronByLabel("ec3");
      Neuron ec4 = network.getNeuronByLabel("ec4");
      Neuron ec5 = network.getNeuronByLabel("ec5");
      Neuron ec6 = network.getNeuronByLabel("ec6");
```

```
      Neuron ec7 = network.getNeuronByLabel("ec7");
      Neuron ec8 = network.getNeuronByLabel("ec8");

      Neuron sup0 = network.getNeuronByLabel("sup0");
      Neuron sup1 = network.getNeuronByLabel("sup1");
      Neuron sup2 = network.getNeuronByLabel("sup2");
      Neuron sup3 = network.getNeuronByLabel("sup3");

      Neuron or0 = network.getNeuronByLabel("or0");
      Neuron or1 = network.getNeuronByLabel("or1");
      Neuron or2 = network.getNeuronByLabel("or2");
      Neuron or3 = network.getNeuronByLabel("or3");


    // Start making the buttons!
    JInternalFrame internalFrame = new JInternalFrame("Parametrage",
true, true);
    LabelledItemPanel panel = new LabelledItemPanel();


      // Bouton mettant les synapses or Ã  0 pour dÃ©buter l
apprentissage sans interference
    JButton buttonPursuer = new JButton("cut off synapses");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (i = 0; i < 4; i++){
                     Neuron nOr = network.getNeuronByLabel("or" +
String.valueOf(i));
                     for (j = 0; j < 27; j++){
                          Neuron res =
network.getNeuronByLabel("res" + String.valueOf(j));
                          network.getSynapse(res,
nOr).setStrength(0);
                     }
                }
            workspace.iterate();
        }});
    panel.addItem("Init Apprentissage", buttonPursuer);

      //This button stops the learning and freeze the synapses
    JButton buttonPursuer = new JButton("freeze all");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (i = 0; i < 4; i++){
                     Neuron nOr = network.getNeuronByLabel("or" +
String.valueOf(i));
                     for (j = 0; j < 27; j++){
                          Neuron res =
network.getNeuronByLabel("res" + String.valueOf(j));
                          network.getSynapse(res,
nOr).setFrozen(true);
                     }
                }
            workspace.iterate();
        }});
//    panel.addItem("Finish learning", buttonPursuer);
```

```
          //This button let start again the learning by unfrozing
synapses
    JButton buttonPursuer = new JButton("defreeze");
    buttonPursuer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (i = 0; i < 4; i++){
                     Neuron nOr = network.getNeuronByLabel("or" +
String.valueOf(i));
                     for (j = 0; j < 27; j++){
                          Neuron res =
network.getNeuronByLabel("res" + String.valueOf(j));
                          network.getSynapse(res,
nOr).setFrozen(false);
                     }
                }
            workspace.iterate();
        }});
 //   panel.addItem("Restart learning", buttonPursuer);


     JButton buttonAvoider = new JButton("New Repartition !");
    buttonAvoider.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (int i = 0; i < nRow; i++){
                     for (int j = 0; j < nRow; j++){
                          for (int k = 0 ; k < nRow; k++){

                               Neuron nReservoir =
network.getNeuronByLabel(( "res" + String.valueOf( nRow * j + k +
nRow * nRow * i)));
                               i_prec = i - 1;
                               i_suiv = i + 1;
                               j_prec = j - 1;
                               j_suiv = j + 1;
                               k_prec = k - 1;
                               k_suiv = k + 1;

                               if (i == 0){i_prec = nRow - 1;}
                               if (j == 0){j_prec = nRow - 1;}
                               if (k == 0){k_prec = nRow - 1;}
                               if (i == nRow - 1){i_suiv = 0;}
                               if (j == nRow - 1){j_suiv = 0;}
                               if (k == nRow - 1){k_suiv = 0;}

                               a1 = Math.random();
                               a2 = Math.random();
                               a3 = Math.random();
                               a4 = Math.random();
                               a5 = Math.random();
                               a6 = Math.random();


                               voisin1 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * i + nRow * j + (k_prec))));
                               voisin2 = network.getNeuronByLabel((
```

```
"res" + String.valueOf( nRow * nRow * i + nRow * j + (k_suiv))));
                                    voisin3 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * i + nRow * (j_prec) + k)));
                                    voisin4 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * i + nRow * (j_suiv) + k)));
                                    voisin5 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * (i_prec) + nRow * j + k)));

                                    voisin6 = network.getNeuronByLabel((
"res" + String.valueOf( nRow * nRow * (i_suiv) + nRow * j + k)));


                                    if (a1 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin1).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin1).setStrength(wectores);}

                                    if (a2 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin2).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin2).setStrength(wectores);}

                                    if (a3 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin3).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin3).setStrength(wectores);}

                                    if (a4 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin4).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin4).setStrength(wectores);}

                                    if (a5 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin5).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin5).setStrength(wectores);}

                                    if (a6 <
Double.parseDouble(buttonInhib.getText())){network.getSynapse(nReser
voir, voisin6).setStrength(-wectores);
                                    }else{network.getSynapse(nReservoir,
voisin6).setStrength(wectores);}
                                }
                            }
                        }
                        workspace.iterate();
                }});
        panel.addItem("Change Reservoir ", buttonAvoider);
```

```
    JButton buttonReverseWeights = new JButton("Do it!");
    buttonReverseWeights.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                for (i = 0; i < Double.parseDouble(niter.getText());
i++){
             workspace.iterate();
                 }
            }
      });
    panel.addItem("Run", buttonReverseWeights);

    JTextField niter = new JTextField("1000");
    panel.addItem("iterations", niter);
      JTextField buttonInhib = new
JTextField(String.valueOf(tauxInhib));
    panel.addItem("tauxInhib reservoir", buttonInhib);

    // Set up Frame
    internalFrame.setLocation(0,500);
    internalFrame.getContentPane().add(panel);
    internalFrame.setVisible(true);
    internalFrame.pack();
    desktop.addInternalFrame(internalFrame);

}


main();
```

Script du réseau de détection des mouvement (Produitsynaptic)

```
// Parametres Neurone Morris-Lecar
     //Il suffit de commenter les paramÃ¨tres pour que leur valeur
soit la valeur standard de Simbrain
MorrisLecarRule generalRule = new MorrisLecarRule();
//generalRule.setG_Ca(6.9);
//generalRule.setG_K(10);
//generalRule.setG_L(0.9);
//generalRule.setvRest_Ca(100);
//generalRule.setvRest_k(-100);
//generalRule.setvRest_L(-100);
//generalRule.setcMembrane(0.05);
//generalRule.setV_m1(0);
//generalRule.setV_m2(40);
//generalRule.setV_w1(0);
//generalRule.setV_w2(40);
//generalRule.setPhi(0.3);
generalRule.setI_bg(0);  // courant d excitation du neurone toujours
present pendant la simulation
generalRule.setThreshold(0);
//generalRule.setAddNoise(false);
generalRule.setInputType(InputType.SYNAPTIC); // Les entrees soient
provoquees par les spikes uniquement

// Parametres neurones lineaires du reseau ( servent de sources de
courant)
LinearRule pixel = new LinearRule();
pixel.setLowerBound(0);
pixel.setUpperBound(100);


double timeStep = 0.1;

//Parametres rise and decay
double tauRise = 7;
double maxResp = 30;

//Parametres jump and decay
double baseLine = 0;
double jumpHeight = 30;
double tauRamp = 50;

double iex = 60; // courant d excitation neurone 1
double poidsSynapse = 1;


void main() {
    buildNetwork();
     addTimeSeries();

}

void buildNetwork() {
    NetworkComponent networkComponent = new NetworkComponent(" Test
```

```
Produit SYnaptique");
    workspace.addWorkspaceComponent(networkComponent);
    Network network = networkComponent.getNetwork();
    network.setTimeStep(timeStep);

     // Sources de courant
     Neuron ex1 = new Neuron(network, pixel.deepCopy());
     ex1.setLocation(0,0);
     ex1.setClamped(true);     // On enlÃ¨ve cette ligne pour
pouvoir utiliser le DAtaworld dans l apprentissage
     ex1.setLabel("ex" + String.valueOf( 1 ));
     network.addNeuron(ex1);

     Neuron ex2 = new Neuron(network, pixel.deepCopy());
     ex2.setLocation(0,100);
     ex2.setClamped(true);     // On enlÃ¨ve cette ligne pour
pouvoir utiliser le DAtaworld dans l apprentissage
     ex2.setLabel("ex" + String.valueOf( 2 ));
     network.addNeuron(ex2);


     // Neurones On ou Off sur lesquels il y aura un deplacement
     Neuron nML1 = new Neuron(network, generalRule.deepCopy());
     nML1.setLocation(100,0);
     nML1.setLabel("n1");
     nML1.getUpdateRule().setI_bg(0);
     network.addNeuron(nML1);

     Neuron nML2 = new Neuron(network, generalRule.deepCopy());
     nML2.setLocation(100,100);
     nML2.setLabel("n2");
     nML2.getUpdateRule().setI_bg(0);
     network.addNeuron(nML2);

     // Neurones intermediaires necessaires (Role de synapses)
     Neuron jump = new Neuron(network, "LinearRule");
     jump.setLocation(200,0);
     jump.setLabel("jump");
     jump.setUpperBound(1000);
     jump.setLowerBound(-1000);
     jump.getUpdateRule().setInputType(InputType.SYNAPTIC);
     network.addNeuron(jump);

     Neuron rise = new Neuron(network, "LinearRule");
     rise.setLocation(200,100);
     rise.setLabel("rise");
     rise.setUpperBound(1000);
     rise.setLowerBound(-1000);
     rise.getUpdateRule().setInputType(InputType.SYNAPTIC);
     network.addNeuron(rise);

     // Neurone de la colonne qui va reagir au deplacement
     Neuron nML3 = new Neuron(network, generalRule.deepCopy());
     nML3.setLocation(300,50);
     nML3.setLabel("n3");
     nML3.getUpdateRule().setI_bg(0);
```

```
      nML3.getUpdateRule().setInputType(InputType.WEIGHTED);
      network.addNeuron(nML3);

      //Synapses sources de courant to neurones
     Synapse extoon1 = new Synapse(ex1 , nML1, poidsSynapse);
     Synapse extoon2 = new Synapse(ex2 , nML2, poidsSynapse);
      network.addSynapse(extoon1);
      network.addSynapse(extoon2);

      // Synapses neurones to intermediaires
      Synapse ontoramp = new Synapse(nML1 , jump, poidsSynapse);
      ontoramp.setSpikeResponder(new JumpAndDecay());
      ontoramp.getSpikeResponder().setBaseLine(baseLine);
      ontoramp.getSpikeResponder().setJumpHeight(jumpHeight);
      ontoramp.getSpikeResponder().setTimeConstant(tauRamp);
      network.addSynapse(ontoramp);

     Synapse ontorise = new Synapse(nML2 , rise, poidsSynapse);
      ontorise.setSpikeResponder(new RiseAndDecay());
      ontorise.getSpikeResponder().setMaximumResponse(maxResp);
      ontorise.getSpikeResponder().setTimeConstant(tauRise);
      network.addSynapse(ontorise);


      // Synapses intermediaires to colonne
     Synapse ramptocol = new Synapse(jump , nML3, poidsSynapse);
     Synapse risetocol = new Synapse(rise , nML3, poidsSynapse);
      network.addSynapse(ramptocol);
      network.addSynapse(risetocol);



      Neuron testSynaptic = new Neuron(network, "LinearRule");
      testSynaptic.setLocation(300,-50);
      testSynaptic.setLabel("testSynaptic");
      testSynaptic.setUpperBound(1000);
      testSynaptic.setLowerBound(-1000);
      testSynaptic.getUpdateRule().setInputType(InputType.SYNAPTIC);
      network.addNeuron(testSynaptic);

      Neuron testWeighted = new Neuron(network, "LinearRule");
      testWeighted.setLocation(300,150);
      testWeighted.setLabel("testWeighted");
      testWeighted.setUpperBound(1000);
      testWeighted.setLowerBound(-1000);
      testWeighted.getUpdateRule().setInputType(InputType.WEIGHTED);
      network.addNeuron(testWeighted);

     Synapse ramptowei = new Synapse(jump , testWeighted,
poidsSynapse);
     Synapse risetowei = new Synapse(rise , testWeighted,
poidsSynapse);
      network.addSynapse(ramptowei);
      network.addSynapse(risetowei);

     Synapse ramptosyn = new Synapse(jump , testSynaptic,
```

```
poidsSynapse);
    Synapse risetosyn = new Synapse(rise , testSynaptic,
poidsSynapse);
     network.addSynapse(ramptosyn);
     network.addSynapse(risetosyn);
}

void addTimeSeries() {

    NetworkComponent networkComponent = ((NetworkComponent)
workspace.getComponent(" Test Produit SYnaptique"));
     Network network = networkComponent.getNetwork();

     // Make time series chartLM LM
    TimeSeriesPlotComponent chartML = new
TimeSeriesPlotComponent("Test spikeresponder", 5);
     chartML.getModel().setWindowSize(5000);
    workspace.addWorkspaceComponent(chartML);

desktop.getDesktopComponent(chartML).getParentFrame().setBounds(500,
0,800,350);

     Neuron nML1 = network.getNeuronByLabel("n1");
     Neuron nML2 = network.getNeuronByLabel("n2");
     Neuron nML3 = network.getNeuronByLabel("n3");
     Neuron jump = network.getNeuronByLabel("jump");
     Neuron rise = network.getNeuronByLabel("rise");

     PotentialProducer neuronProducerML1 =
NetworkComponent.getNeuronProducer(networkComponent, nML1 ,
"getActivation");
     PotentialConsumer timeSeriesConsumerML1 =
chartML.getPotentialConsumers().get(0);
     workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML1, timeSeriesConsumerML1));

     PotentialProducer neuronProducerML2 =
NetworkComponent.getNeuronProducer(networkComponent, nML2 ,
"getActivation");
     PotentialConsumer timeSeriesConsumerML2 =
chartML.getPotentialConsumers().get(1);
     workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML2, timeSeriesConsumerML2));

     PotentialProducer neuronProducerML3 =
NetworkComponent.getNeuronProducer(networkComponent, nML3 ,
"getActivation");
     PotentialConsumer timeSeriesConsumerML3 =
chartML.getPotentialConsumers().get(2);
     workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerML3, timeSeriesConsumerML3));

     PotentialProducer neuronProducerRamp =
NetworkComponent.getNeuronProducer(networkComponent, jump ,
"getActivation");
     PotentialConsumer timeSeriesConsumerRamp =
```

```
chartML.getPotentialConsumers().get(3);
     workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerRamp, timeSeriesConsumerRamp));

     PotentialProducer neuronProducerRise =
NetworkComponent.getNeuronProducer(networkComponent, rise ,
"getActivation");
     PotentialConsumer timeSeriesConsumerRise =
chartML.getPotentialConsumers().get(4);
     workspace.getCouplingManager().addCoupling(new
Coupling(neuronProducerRise, timeSeriesConsumerRise));

}

main();
```

Script de la Rétine Artificielle

```
// Parametres utiles

int nRow = 7; // Nombre de lignes a l ecran
int nCol = 7; // Nombre de colonnes a l ecra
int nPixels = nCol * nRow; // nombre de pixel total
int distNeuron = 40; // distance entre deux neurones proches

// Parametres Neurone Morris Lecar
MorrisLecarRule generalRule = new MorrisLecarRule();
generalRule.setG_Ca(6.9);
generalRule.setG_K(10);
generalRule.setG_L(0.9);
generalRule.setvRest_Ca(100);
generalRule.setvRest_k(-100);
generalRule.setvRest_L(-100);
generalRule.setcMembrane(0.05);
generalRule.setV_m1(0);
generalRule.setV_m2(40);
generalRule.setV_w1(0);
generalRule.setV_w2(40);
generalRule.setPhi(0.3);
generalRule.setI_bg(0);
generalRule.setThreshold(0);
generalRule.setAddNoise(false);
generalRule.setInputType(InputType.SYNAPTIC);

// Parametres Photorecepteur
LinearRule pixel = new LinearRule();
pixel.setLowerBound(0);
pixel.setUpperBound(100);
//   pixel.setClamped(true); // s applique aux neurones et pas aux
update rule


double pds12 = 0.15; // Poids synaptique entre la couche 1 (neurones
On) et la couche 2 (colonnes simples)


double maxResp = 1; // Maximum de la reponse synaptique (Rise and
Decay)
double tau = 0.3; // temps caracteristique de la reponse synaptique
(Rise and Decay)

nFormes = 2; // nombre de formes reconnues par une colonne complexe
connectÃ©e Ã  un groupe de 5* 5 neurones on donc 3 * 3 colonnes
simples

//Parameters STDP
double taum = 40.0/3;
double taup = 40.0/3;
double learningrate = 0.25 ;
double wp = 1;
double wm = 1;
double sitocomax = 1;
```

```
double sitocomin = -1;

void main(){
     buildNetwork();
//    addTimeSeries();

}

void buildNetwork() {
    // Parameters

    // Set up network
     NetworkComponent networkComponent = new
NetworkComponent("Retine");
     workspace.addWorkspaceComponent(networkComponent);
     Network network = networkComponent.getNetwork();
     network.setTimeStep(.005);

      //Creation couche0 : Photorecepteurs
      // Geometrie Couche0 carree (neurones numerotes de gauche
Ã  droite et de haut en bas)
      for (int i = 0; i < nRow ; i++) {
            for (int j = 0; j < nCol; j++) {
                  Neuron n = new Neuron(network, pixel.deepCopy());
                  n.setLocation( j * distNeuron, i * distNeuron);
                  n.setClamped(true);
                  n.setLabel("ec" + String.valueOf( nCol * i + j ));
                  network.addNeuron(n);
            }
      }

      //Construction Couche1 : Neurones On

      //RepÃ¨re Position Couche1
      Neuron topRight0 = network.getNeuronByLabel("ec" +
String.valueOf(nCol-1));
      int xCouche1 = topRight0.getX() + 100 ;
//    topRight1.setLabel("topRight"); //test pour savoir si le bon
neurone a Ã©tÃ© choisi

      //CrÃ©ation Couche1 : Neurones On
      //Geometrie Couche1 carree (meme numerotation)
      for (int i = 0; i < nRow ; i++) {
            for (int j = 0; j < nCol; j++) {
                  Neuron n = new Neuron(network,
generalRule.deepCopy());
                  n.setLocation( xCouche1 + j * distNeuron , i *
distNeuron );
                  n.setLabel( "on" + String.valueOf( nCol * i + j ));
                  network.addNeuron(n);
            }
      }

      // Connexion Couche0 - Couche1  Synapses Photorecepteurs to
Neurones On
```

```
      for (int i = 0; i < nRow; i++){
            for (int j = 0; j < nCol; j++){

                  // Connexion entre le neurone On et son pixel
central
                  Neuron pixelCentral = network.getNeuronByLabel("ec"
+ String.valueOf( nCol * i + j ));
                  Neuron neuronOn = network.getNeuronByLabel("on" +
String.valueOf( nCol * i + j ));
                  network.addSynapse( new Synapse(pixelCentral,
neuronOn, 4)) ;

                  //La prise en compte des effets de bord implique un
traitement different pour chaque coin et chaque cote

                  // Coin en haut Ã  gauche
                  if (i == 0 && j == 0){
                        Neuron pixelBas = network.getNeuronByLabel("ec"
+ String.valueOf( nCol * (i+1) +j ));
                        Neuron pixelDroit =
network.getNeuronByLabel("ec" + String.valueOf(nCol * i + j + 1 ));
                        network.addSynapse(new Synapse(pixelBas,
neuronOn, -2));
                        network.addSynapse(new Synapse(pixelDroit,
neuronOn, -2));
                  }

                  // Coin en bas Ã  gauche
                  else if (i == nRow-1 && j == 0){
                        Neuron pixelHaut =
network.getNeuronByLabel("ec" + String.valueOf( nCol * (i-1) +j));

                        Neuron pixelDroit =
network.getNeuronByLabel("ec" + String.valueOf( nCol * i + j + 1));
                        network.addSynapse( new Synapse(pixelHaut,
neuronOn, -2)) ;
                        network.addSynapse( new Synapse(pixelDroit,
neuronOn, -2)) ;
                  }

                  //Coin en haut Ã  gauche
                  else if (i == 0 && j == nCol-1){
                        Neuron pixelBas = network.getNeuronByLabel("ec"
+ String.valueOf( nCol * (i+1) +j));
                        Neuron pixelGauche =
network.getNeuronByLabel("ec" + String.valueOf( nCol * i + j - 1));

                        network.addSynapse( new Synapse(pixelBas,
neuronOn, -2)) ;
                        network.addSynapse( new Synapse(pixelGauche,
neuronOn, -2)) ;
                  }

                  //Coin en bas Ã  droite
                  else if(i == nRow-1 && j == nCol-1){
                        Neuron pixelHaut =
```

```
network.getNeuronByLabel("ec" + String.valueOf( nCol * (i-1) + j));

                        Neuron pixelGauche =
network.getNeuronByLabel("ec" + String.valueOf( nCol * i + j - 1));
                        network.addSynapse( new Synapse(pixelHaut,
neuronOn, -2)) ;
                        network.addSynapse( new Synapse(pixelGauche,
neuronOn, -2)) ;
                 }

                 //Colonne Gauche sans les coins
                 else if (j == 0){
                        Neuron pixelHaut =
network.getNeuronByLabel("ec" + String.valueOf( nCol * (i-1) + j));

                        Neuron pixelBas = network.getNeuronByLabel("ec"
+ String.valueOf( nCol * (i+1) +j));
                        Neuron pixelDroit =
network.getNeuronByLabel("ec" + String.valueOf( nCol * (i-1) + j));
                        network.addSynapse( new Synapse(pixelHaut,
neuronOn, -1)) ;
                        network.addSynapse( new Synapse(pixelBas,
neuronOn, -1)) ;
                        network.addSynapse( new Synapse(pixelDroit,
neuronOn, -2)) ;
                 }

                 //Ligne Haut sans les coins
                 else if (i == 0){
                        Neuron pixelBas = network.getNeuronByLabel("ec"
+ String.valueOf( nCol * (i+1) +j));
                        Neuron pixelGauche =
network.getNeuronByLabel("ec" + String.valueOf( nCol * i + j - 1));

                        Neuron pixelDroit =
network.getNeuronByLabel("ec" + String.valueOf(nCol * i + j + 1));

                        network.addSynapse( new Synapse(pixelBas,
neuronOn, -2)) ;
                        network.addSynapse( new Synapse(pixelGauche,
neuronOn, -1)) ;
                        network.addSynapse( new Synapse(pixelDroit,
neuronOn, -1)) ;
                 }

                 //Ligne Bas sans les coins
                 else if (i == nRow-1){
                        Neuron pixelHaut =
network.getNeuronByLabel("ec" + String.valueOf( nCol * (i-1) + j));

                        Neuron pixelGauche =
network.getNeuronByLabel("ec" + String.valueOf( nCol * i + j - 1));
                        Neuron pixelDroit =
network.getNeuronByLabel("ec" + String.valueOf( nCol * i + j + 1));
                        network.addSynapse( new Synapse(pixelHaut,
neuronOn, -2)) ;
```

```
                        network.addSynapse( new Synapse(pixelGauche,
neuronOn, -1)) ;
                        network.addSynapse( new Synapse(pixelDroit,
neuronOn, -1)) ;
                }

                //Colonne Droite sans les coins
                else if ( j == nCol-1){
                        Neuron pixelHaut =
network.getNeuronByLabel("ec" + String.valueOf( nCol * (i-1) + j));

                        Neuron pixelBas = network.getNeuronByLabel("ec"
+ String.valueOf( nCol * (i+1) +j));
                        Neuron pixelGauche =
network.getNeuronByLabel("ec" + String.valueOf( nCol * i + j - 1));
                        network.addSynapse( new Synapse(pixelHaut,
neuronOn, -1)) ;
                        network.addSynapse( new Synapse(pixelBas,
neuronOn, -1)) ;
                        network.addSynapse( new Synapse(pixelGauche,
neuronOn, -2)) ;
                }

                //Tous sauf les bords
                else{
                        Neuron pixelHaut =
network.getNeuronByLabel("ec" + String.valueOf( nCol * (i-1) + j));

                        Neuron pixelBas = network.getNeuronByLabel("ec"
+ String.valueOf(nCol * (i+1) +j));
                        Neuron pixelGauche =
network.getNeuronByLabel("ec" + String.valueOf(nCol * i + j - 1));

                        Neuron pixelDroit =
network.getNeuronByLabel("ec" + String.valueOf( nCol * i + j + 1));

                        network.addSynapse( new Synapse(pixelHaut,
neuronOn, -1)) ;
                        network.addSynapse( new Synapse(pixelBas,
neuronOn, -1)) ;
                        network.addSynapse( new Synapse(pixelGauche,
neuronOn, -1)) ;
                        network.addSynapse( new Synapse(pixelDroit,
neuronOn, -1)) ;
                }
           }
     }

     //Construction Couche2 : Colonnes Simples

     //Reperes position Couche2
     Neuron topRight1 = network.getNeuronByLabel("on" +
String.valueOf(nCol-1));
     int xCouche2 = topRight1.getX() + 100 ;
     int yCouche2 = -(nRow * (4 * distNeuron + 5 * distNeuron) - 5 *
distNeuron ) /4;
```

```
      //CrÃ©ation groupe couche2 colonnes simples
      //Geometrie couche2 : colonnes correspondent a leurs neurones
on et off
      for (int i = 1; i < (nRow-1); i++) {
            for (int j = 1; j < (nCol-1); j++) {
                  for (int k = 0; k < 4; k++){

                        //Orientations verticales |||||
                        if (k == 0){
                              Neuron n = new Neuron(network,
generalRule.deepCopy());
                              n.setLocation( xCouche2 + j * distNeuron
, yCouche2 + i * distNeuron * 5 + k * distNeuron);
                              n.setLabel( "si |" + String.valueOf( (
(nCol-2)   *   (i-1) + (j-1)  )  ));
                              network.addNeuron(n);
                        }

                        //Orientations diagonale vers le haut ////
                        else if (k == 1){
                              Neuron n = new Neuron(network,
generalRule.deepCopy());
                              n.setLocation( xCouche2 + j * distNeuron
, yCouche2 + i * distNeuron * 5 + k * distNeuron);
                              n.setLabel( "si /" + String.valueOf( (
(nCol-2)   *   (i-1) + (j-1)  )  ));
                              network.addNeuron(n);
                        }

                        //Orientations horizontales _____
                        else if (k == 2){
                              Neuron n = new Neuron(network,
generalRule.deepCopy());
                              n.setLocation( xCouche2 + j * distNeuron
, yCouche2 + i * distNeuron * 5 + k * distNeuron);
                              n.setLabel( "si _" + String.valueOf((
(nCol-2)   *   (i-1) + (j-1)  )  ));
                              network.addNeuron(n);
                        }

                        //Orientations diagonale vers le bas \\\\\
                        else if (k == 3){
                              Neuron n = new Neuron(network,
generalRule.deepCopy());
                              n.setLocation( xCouche2 + j * distNeuron
, yCouche2 + i * distNeuron * 5 + k * distNeuron);
                              n.setLabel( "si \\" + String.valueOf( (
(nCol-2)   *   (i-1) + (j-1)  ) ));
                              network.addNeuron(n);
                        }
                  }
            }
      }
```

```
     //Connexion Couche1 - Couche2    Synapses Neurones On to
Colonnes corticales simples


     for (int i = 1; i < (nRow-1); i++) {
          for (int j = 1; j < (nCol-1); j++) {
               for (int k = 0; k < 4; k++){
                    //Orientations verticales |
                    if (k == 0){
                         Neuron n = network.getNeuronByLabel("si
|" + String.valueOf(  (   (nCol-2)   *   (i-1) + (j-1))));
                         Neuron on00 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*(i-1) + j));
                         Neuron on01 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*i + j));
                         Neuron on02 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*(i+1) + j ));


                         Synapse s00 = new Synapse(on00, n,
pds12);
                         Synapse s01 = new Synapse(on01, n,
pds12);
                         Synapse s02 = new Synapse(on02, n,
pds12);


                         s00.setSpikeResponder( new
RiseAndDecay());
                         s01.setSpikeResponder( new
RiseAndDecay());
                         s02.setSpikeResponder( new
RiseAndDecay());


     s00.getSpikeResponder().setMaximumResponse(maxResp);

     s01.getSpikeResponder().setMaximumResponse(maxResp);

     s02.getSpikeResponder().setMaximumResponse(maxResp);


     s00.getSpikeResponder().setTimeConstant(tau);

     s01.getSpikeResponder().setTimeConstant(tau);

     s02.getSpikeResponder().setTimeConstant(tau);

                         network.addSynapse( s00);
```

```
                        network.addSynapse( s01);
                        network.addSynapse( s02);

                    }

                    //Orientations diagonale vers le haut /
                    else if (k == 1){

                        Neuron n = network.getNeuronByLabel("si
/" + String.valueOf(  (   (nCol-2)   *   (i-1) + (j-1))));
                        Neuron on10 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*(i-1) + j+1
));
                        Neuron on11 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*i + j));
                        Neuron on12 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*(i+1) + j-1));


                        Synapse s10 = new Synapse(on10, n,
pds12);
                        Synapse s11 = new Synapse(on11, n,
pds12);
                        Synapse s12 = new Synapse(on12, n,
pds12);

                        s10.setSpikeResponder( new
RiseAndDecay());
                        s11.setSpikeResponder( new
RiseAndDecay());
                        s12.setSpikeResponder( new
RiseAndDecay());


     s10.getSpikeResponder().setMaximumResponse(maxResp);

     s11.getSpikeResponder().setMaximumResponse(maxResp);

     s12.getSpikeResponder().setMaximumResponse(maxResp);


     s10.getSpikeResponder().setTimeConstant(tau);

     s11.getSpikeResponder().setTimeConstant(tau);

     s12.getSpikeResponder().setTimeConstant(tau);

                        network.addSynapse( s10);
                        network.addSynapse( s11);
                        network.addSynapse( s12);
                    }

                    //Orientations horizontales _
                    else if (k == 2){
                        Neuron n = network.getNeuronByLabel("si
_" + String.valueOf(  (   (nCol-2)   *   (i-1) + (j-1))));
```

```
                              Neuron on20 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*i + j-1));
                              Neuron on21 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*i + j));
                              Neuron on22 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*i + j+1 ));


                              Synapse s20 = new Synapse(on20, n,
pds12);
                              Synapse s21 = new Synapse(on21, n,
pds12);
                              Synapse s22 = new Synapse(on22, n,
pds12);

                              s20.setSpikeResponder( new
RiseAndDecay());
                              s21.setSpikeResponder( new
RiseAndDecay());
                              s22.setSpikeResponder( new
RiseAndDecay());


     s20.getSpikeResponder().setMaximumResponse(maxResp);

     s21.getSpikeResponder().setMaximumResponse(maxResp);

     s22.getSpikeResponder().setMaximumResponse(maxResp);


     s20.getSpikeResponder().setTimeConstant(tau);

     s21.getSpikeResponder().setTimeConstant(tau);

     s22.getSpikeResponder().setTimeConstant(tau);

                              network.addSynapse( s20);
                              network.addSynapse( s21);
                              network.addSynapse( s22);

                         }

                         //Orientations diagonale vers le bas \
                         else if (k == 3){
                              Neuron n = network.getNeuronByLabel("si
\\" + String.valueOf(  (   (nCol-2)   *   (i-1) + (j-1))));
                              Neuron on30 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*(i-1) + j-1));
                              Neuron on31 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*i + j));
                              Neuron on32 =
network.getNeuronByLabel("on" + String.valueOf((nCol)*(i+1) + j+1));


                              Synapse s30 = new Synapse(on30, n,
pds12);
```

```
                              Synapse s31 = new Synapse(on31, n,
pds12);
                              Synapse s32 = new Synapse(on32, n,
pds12);

                              s30.setSpikeResponder( new
RiseAndDecay());
                              s31.setSpikeResponder( new
RiseAndDecay());
                              s32.setSpikeResponder( new
RiseAndDecay());


      s30.getSpikeResponder().setMaximumResponse(maxResp);

      s31.getSpikeResponder().setMaximumResponse(maxResp);

      s32.getSpikeResponder().setMaximumResponse(maxResp);


      s30.getSpikeResponder().setTimeConstant(tau);

      s31.getSpikeResponder().setTimeConstant(tau);

      s32.getSpikeResponder().setTimeConstant(tau);

                              network.addSynapse( s30);
                              network.addSynapse( s31);
                              network.addSynapse( s32);
                        }
                  }
            }
      }




      //Construction Couche3 Groupe Colonnes Complexes (taille
minimale de l ecran de 5 * 5 )

      if (nRow >=5 && nCol >=5 ){

            //Version 1 : apprentissage de deux types de formes
possible

            //Reperes position Couche3
            Neuron topRight2 = network.getNeuronByLabel("si |" +
String.valueOf(nCol-3));
            int xCouche3 = topRight2.getX() + 100 ;
            int yCouche3 = topRight2.getY() + 5 * distNeuron ;

            //CrÃ©ation groupe couche3 colonnes complexes
            for (int i = 1; i < (nRow-3); i++) {
                  for (int j = 1; j < (nCol-3); j++) {
```

```
                    for (int k = 0; k < nFormes; k++){
                         Neuron n = new Neuron(network,
generalRule.deepCopy());
                         n.setLocation( xCouche3 + j * distNeuron
, yCouche3 + (i-1) * distNeuron * 5 + k * distNeuron );
                         n.setLabel( "co" + String.valueOf(k) +
String.valueOf( (   (nCol-4)   *   (i-1) + (j-1)  )   ));
                         network.addNeuron(n);

                    }
               }
          }



          //Connexion Couche2 - Couche3    Synapses Colonnes
corticales simples    to Colonnes corticales complexes


          for (int i = 1; i < (nRow-3); i++) {
               for (int j = 1; j < (nCol-3); j++) {
                    for (int k = 0; k < nFormes; k++){

                         Neuron n = network.getNeuronByLabel("co"
+ String.valueOf(k)  + String.valueOf(  (   (nCol-4)   *   (i-1) +
(j-1)  )   ));

                         for (int i2 = i; i2 < i + 3 ; i2++){
                              for (int j2 = j; j2 < j + 3 ; j2++){
                                   for (int k2 = 0; k2 < 4 ;
k2++){
                                        if (k2 == 0){
                                        Neuron si =
network.getNeuronByLabel("si |" + String.valueOf((   (nCol-2)   *
(i2-1) + (j2-1)  )    )) ;
                                        Synapse sitoco= new
Synapse(si, n, 0);

                                        sitoco.setSpikeResponder(
new RiseAndDecay());

     sitoco.getSpikeResponder().setMaximumResponse(maxResp);

     sitoco.getSpikeResponder().setTimeConstant(tau);


     sitoco.setLearningRule(new STDPRule());

     sitoco.getLearningRule().setTau_plus(taup);

     sitoco.getLearningRule().setTau_minus(taum);

     sitoco.getLearningRule().setLearningRate(learningrate);

     sitoco.getLearningRule().setW_plus(wp);
```

```
      sitoco.getLearningRule().setW_minus(wm);


      sitoco.setUpperBound(sitocomax);

      sitoco.setLowerBound(sitocomin);


                                          network.addSynapse(
sitoco);

                                          }else if (k2 == 1){

                                          Neuron si =
network.getNeuronByLabel("si /" + String.valueOf((   (nCol-2)   *
(i2-1) + (j2-1)  )    )) ;
                                          Synapse sitoco= new
Synapse(si, n, 0);


                                          sitoco.setSpikeResponder(
new RiseAndDecay());

      sitoco.getSpikeResponder().setMaximumResponse(maxResp);

      sitoco.getSpikeResponder().setTimeConstant(tau);



      sitoco.setLearningRule(new STDPRule());

      sitoco.getLearningRule().setTau_plus(taup);

      sitoco.getLearningRule().setTau_minus(taum);

      sitoco.getLearningRule().setLearningRate(learningrate);

      sitoco.getLearningRule().setW_plus(wp);

      sitoco.getLearningRule().setW_minus(wm);


      sitoco.setUpperBound(sitocomax);

      sitoco.setLowerBound(sitocomin);


                                          network.addSynapse(
sitoco);
                                          }else if (k2 == 2){

                                          Neuron si =
network.getNeuronByLabel("si _" + String.valueOf((   (nCol-2)   *
(i2-1) + (j2-1)  )    )) ;
                                    //    si.setLabel("jji");
                                          Synapse sitoco= new
```

```
Synapse(si, n, 0);

                                        sitoco.setSpikeResponder(
new RiseAndDecay());
      sitoco.getSpikeResponder().setMaximumResponse(maxResp);
      sitoco.getSpikeResponder().setTimeConstant(tau);

      sitoco.setLearningRule(new STDPRule());
      sitoco.getLearningRule().setTau_plus(taup);
      sitoco.getLearningRule().setTau_minus(taum);
      sitoco.getLearningRule().setLearningRate(learningrate);
      sitoco.getLearningRule().setW_plus(wp);
      sitoco.getLearningRule().setW_minus(wm);

      sitoco.setUpperBound(sitocomax);
      sitoco.setLowerBound(sitocomin);
                                        network.addSynapse(
sitoco);
                                        }else if (k2 == 3){
                                        Neuron si =
network.getNeuronByLabel("si \\" + String.valueOf((   (nCol-2)   *
(i2-1) + (j2-1)  )    )) ;
                                        Synapse sitoco= new
Synapse(si, n, 0);

                                        sitoco.setSpikeResponder(
new RiseAndDecay());
      sitoco.getSpikeResponder().setMaximumResponse(maxResp);
      sitoco.getSpikeResponder().setTimeConstant(tau);

      sitoco.setLearningRule(new STDPRule());
      sitoco.getLearningRule().setTau_plus(taup);
      sitoco.getLearningRule().setTau_minus(taum);
      sitoco.getLearningRule().setLearningRate(learningrate);
      sitoco.getLearningRule().setW_plus(wp);
```

```
      sitoco.getLearningRule().setW_minus(wm);


      sitoco.setUpperBound(sitocomax);

      sitoco.setLowerBound(sitocomin);

                                                network.addSynapse(
sitoco);
                                                }
                                          }
                                    }
                               }
                          }
                     }
           }


           //Construction Couche4 Groupe Supervision Colonne
Complexe

           //Reperes position Couche4

           Neuron topRight3 = network.getNeuronByLabel("co"
+String.valueOf(0) + String.valueOf(nCol-5));
           int xCouche4 = topRight3.getX() + 50 ;
           int yCouche4 = topRight3.getY() +125   ;

           //CrÃ©ation groupe couche4 colonnes complexes

           //Geometrie couche4 Positions neurones supervisor

           for (int i = 1; i < (nRow-3); i++) {
                for (int j = 1; j < (nCol-3); j++) {
                     for (int k = 0; k < nFormes; k++){
                          Neuron n = new Neuron(network,
pixel.deepCopy());
                          n.setLocation(xCouche4 + j * distNeuron,
yCouche4 + (i-1) * distNeuron * 5 + k * distNeuron);
                          n.setClamped(true);
                          n.setLabel("sup" + String.valueOf(k) +
String.valueOf((   (nCol-4)   *   (i-1) + (j-1)  )   ));
                          network.addNeuron(n);

                     }
                }
           }
```

```
          //Connexion Couche3 - Couche4    Supervisors to Colonnes
corticales complexes
          for (int i = 1; i < (nRow-3); i++) {
               for (int j = 1; j < (nCol-3); j++) {
                    for (int k = 0; k < nFormes; k++){
                         Neuron n = network.getNeuronByLabel("co"
+ String.valueOf(k)  + String.valueOf(  (   (nCol-4)   *   (i-1) +
(j-1)  )   ));
                         Neuron supervisor =
network.getNeuronByLabel("sup" + String.valueOf(k)  +
String.valueOf(  (   (nCol-4)   *   (i-1) + (j-1)  )   ));

                         network.addSynapse( new Synapse(
supervisor, n, 1));
                    }
               }
          }
     }
     }

main();
```

Annexe 3 : Codes Simbrain

Tracé des courbes d'intérêt

```
# -*- coding: utf-8 -*-
"""
Created on Sun Sep  3 23:25:15 2017

@author: Anthony Ozier-lafontaine
"""

neurone_bio = {'g_Ca': 6.9,
               'g_K': 10,
               'g_L': 0.9,
               'vRest_Ca': 100,
               'vRest_K': -100,
               'vRest_L': -100,
               'cMembrane': 0.05,
               'v_m1': 0,
               'v_m2': 40,
               'v_w1': 0,
               'v_w2': 40,
               'phi': 0.3,
               'i_bg': 0,
               'dt' : 0.001,
               "activation": 0,
               "n": 0}

neurone_simbrain = {'g_Ca': 4,
                    'g_K': 8,
                    'g_L': 2,
                    'vRest_Ca': 120,
                    'vRest_K': -80,
                    'vRest_L': -60,
                    'cMembrane': 5,
                    'v_m1': -1.2,
                    'v_m2': 18,
                    'v_w1': 2,
                    'v_w2': 17.4,
                    'phi': 0.066667,
                    'i_bg': 0,
                    'dt' : 0.1,
                    "activation": 0,
                    "n": 0}

 #%%

# Cette portion de code permet de dÃ©finir le traÃ§age des courbes
d'intÃ©rÃªt et celui des isoclines en zÃ©ro
# import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
# show plots in notebook
```

```
# matplotlib inline

# define system in terms of separated differential equations

neurone = neurone_bio.copy()
def f(x,y):
    return 2*x - x**2 - x*y

def dV_dt(x, y, neurone):
    i_Ca = neurone['g_Ca'] * membraneFunction(x, neurone) * ( x -
neurone['vRest_Ca'] )
    i_K = neurone['g_K'] * y * (x - neurone['vRest_K'])
    i_L = neurone['g_L'] * (x - neurone['vRest_L'])
    i_ion = i_Ca + i_K + i_L
    return (neurone['i_bg'] - i_ion)/neurone['cMembrane']

def membraneFunction(x, neurone):
    return 0.5 * (1 + np.tanh( (x - neurone['v_m1']) /
neurone['v_m2'] ))


def dW_dt(x, y, neurone):
    return neurone['phi'] * lambdaFunction(x, neurone) * (
k_FractionFunction(x, neurone) - y )

def k_FractionFunction ( x, neurone ) :
    return 0.5 * ( 1 + np.tanh( ( x - neurone['v_w1'] ) /
neurone['v_w2'] ))

def lambdaFunction(x, neurone):
    return np.cosh( (x - neurone['v_w1']) / (2 * neurone['v_w2']))

def g(x,y):
    return - y + x*y


    # courbes d'intÃ©rÃªt
def sys(vMi, ni,trest, tex, tfin, iex, neurone):
    x = []
    y = []
    # initial values:
    x.append(vMi)
    y.append(ni)

    #tracÃ© des courants
    ica = [neurone['g_Ca'] * membraneFunction(x[0], neurone) * (
x[0] - neurone['vRest_Ca'] )]
    ik = [neurone['g_K'] * y[0] * (x[0] - neurone['vRest_K'])]
    iL = [neurone['g_L'] * (x[0] - neurone['vRest_L'])]
    itot = [ica[0] + ik[0] + iL[0] ]
    ibg =[0]

    ica1 = [neurone['g_Ca'] * membraneFunction(x[0], neurone) *100]
    ica2 = [( x[0] - neurone['vRest_Ca'] )]
    w1 = [neurone['phi'] * lambdaFunction(x[0], neurone)]
    w2 = [( k_FractionFunction(x[0], neurone) - y[0] ) ]
```

```
    #z.append(iv3)
    # compute and fill lists
    neurone['i_bg']= 0
    for i in range(trest):

##        RÃ©solution 1]
#        x.append(x[i] + (dV_dt(x[i],y[i], neurone)) *
neurone['dt'])
#        y.append(y[i] + (dW_dt(x[i],y[i], neurone)) *
neurone['dt'])

#       RÃ©solution 2
        xfut = x[i] + (dV_dt(x[i],y[i], neurone)) * neurone['dt']
        yfut = y[i] + (dW_dt(x[i],y[i], neurone)) * neurone['dt']

        x.append(x[i] + neurone['dt']/2 * (dV_dt(x[i],y[i], neurone)
+ dV_dt(xfut,yfut, neurone)))
        y.append(y[i] + neurone['dt']/2 * (dW_dt(x[i],y[i], neurone)
+ dW_dt(xfut,yfut, neurone)))

##        Resolution 3 Runge-Kutta
#        k1 = neurone['dt'] * dV_dt(x[i], y[i], neurone)
#        l1 = neurone['dt'] * dW_dt(x[i], y[i], neurone)
#        k2 = neurone['dt'] * dV_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#        l2 = neurone['dt'] * dW_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#        k3 = neurone['dt'] * dV_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#        l3 = neurone['dt'] * dW_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#        k4 = neurone['dt'] * dV_dt(x[i] + k3, y[i] + l3, neurone)
#        l4 = neurone['dt'] * dW_dt(x[i] + k3, y[i] + l3, neurone)
#
#        x.append( x[i] +  (k1 + k2 + k2 + k3 + k3 + k4) / 6  )
#        y.append( y[i] + (l1 + l2 + l2 + l3 + l3 + l4) / 6  )

#     tracÃ© courantss
        ica.append(- neurone['g_Ca'] * membraneFunction(x[i],
neurone) * ( x[i] - neurone['vRest_Ca'] ))
        ik.append(- neurone['g_K'] * y[i] * (x[i] -
neurone['vRest_K']))
        iL.append(- neurone['g_L'] * (x[i] - neurone['vRest_L']))
        itot.append( neurone['i_bg'] + (ica[i] + ik[i] + iL[i])  )
        ibg.append(neurone['i_bg'])

        ica1.append(neurone['g_Ca'] * membraneFunction(x[i],
neurone) *100)
        ica2.append(( x[i] - neurone['vRest_Ca'] ))
        w1.append(neurone['phi'] * lambdaFunction(x[i], neurone))
        w2.append(( k_FractionFunction(x[i], neurone) - y[i] ) )

    a = 0
    for i in range(i ,i + 1000):
#        neurone['i_bg'] = iex * a/1000
```

```
        neurone['i_bg'] = (np.exp(a * np.log(iex + 1) / 1000 ) - 1)

        ibg.append(neurone['i_bg'])
        a +=1
##        RÃ©solution 1
#         x.append(x[i] + (dV_dt(x[i],y[i], neurone)) *
neurone['dt'])
#         y.append(y[i] + (dW_dt(x[i],y[i], neurone)) *
neurone['dt'])
#
#       RÃ©solution 2
        xfut = x[i] + (dV_dt(x[i],y[i], neurone)) * neurone['dt']
        yfut = y[i] + (dW_dt(x[i],y[i], neurone)) * neurone['dt']

        x.append(x[i] + neurone['dt']/2 * (dV_dt(x[i],y[i], neurone)
+ dV_dt(xfut,yfut, neurone)))
        y.append(y[i] + neurone['dt']/2 * (dW_dt(x[i],y[i], neurone)
+ dW_dt(xfut,yfut, neurone)))


##        Resolution 3 Runge-Kutta
#         k1 = neurone['dt'] * dV_dt(x[i], y[i], neurone)
#         l1 = neurone['dt'] * dW_dt(x[i], y[i], neurone)
#         k2 = neurone['dt'] * dV_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#         l2 = neurone['dt'] * dW_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#         k3 = neurone['dt'] * dV_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#         l3 = neurone['dt'] * dW_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#         k4 = neurone['dt'] * dV_dt(x[i] + k3, y[i] + l3, neurone)
#         l4 = neurone['dt'] * dW_dt(x[i] + k3, y[i] + l3, neurone)
#
#         x.append( x[i] +  (k1 + k2 + k2 + k3 + k3 + k4) / 6  )
#         y.append( y[i] + (l1 + l2 + l2 + l3 + l3 + l4) / 6  )
#
#     tracÃ© courantss
        ica.append(- neurone['g_Ca'] * membraneFunction(x[i],
neurone) * ( x[i] - neurone['vRest_Ca'] ))
        ik.append(- neurone['g_K'] * y[i] * (x[i] -
neurone['vRest_K']))
        iL.append(- neurone['g_L'] * (x[i] - neurone['vRest_L']))
        itot.append( neurone['i_bg'] + (ica[i+1] + ik[i+1] +
iL[i+1])  )

        ica1.append(neurone['g_Ca'] * membraneFunction(x[i],
neurone) *100)
        ica2.append(( x[i] - neurone['vRest_Ca'] ))
        w1.append(neurone['phi'] * lambdaFunction(x[i], neurone))
        w2.append(( k_FractionFunction(x[i], neurone) - y[i] ) )


    neurone['i_bg'] = iex
    for i in range(i, i+ tex):
##        RÃ©solution 1
```

```
#         x.append(x[i] + (dV_dt(x[i],y[i], neurone)) *
neurone['dt'])
#         y.append(y[i] + (dW_dt(x[i],y[i], neurone)) *
neurone['dt'])
#
#        RÃ©solution 2
         xfut = x[i] + (dV_dt(x[i],y[i], neurone)) * neurone['dt']
         yfut = y[i] + (dW_dt(x[i],y[i], neurone)) * neurone['dt']

         x.append(x[i] + neurone['dt']/2 * (dV_dt(x[i],y[i], neurone)
+ dV_dt(xfut,yfut, neurone)))
         y.append(y[i] + neurone['dt']/2 * (dW_dt(x[i],y[i], neurone)
+ dW_dt(xfut,yfut, neurone)))

##         Resolution 3 Runge-Kutta
#         k1 = neurone['dt'] * dV_dt(x[i], y[i], neurone)
#         l1 = neurone['dt'] * dW_dt(x[i], y[i], neurone)
#         k2 = neurone['dt'] * dV_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#         l2 = neurone['dt'] * dW_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#         k3 = neurone['dt'] * dV_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#         l3 = neurone['dt'] * dW_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#         k4 = neurone['dt'] * dV_dt(x[i] + k3, y[i] + l3, neurone)
#         l4 = neurone['dt'] * dW_dt(x[i] + k3, y[i] + l3, neurone)
#
#         x.append( x[i] +  (k1 + k2 + k2 + k3 + k3 + k4) / 6  )
#         y.append( y[i] + (l1 + l2 + l2 + l3 + l3 + l4) / 6  )
#
#     tracÃ© courantss
         ica.append(- neurone['g_Ca'] * membraneFunction(x[i],
neurone) * ( x[i] - neurone['vRest_Ca'] ))
         ik.append(- neurone['g_K'] * y[i] * (x[i] -
neurone['vRest_K']))
         iL.append(- neurone['g_L'] * (x[i] - neurone['vRest_L']))
         itot.append( neurone['i_bg'] + (ica[i+1] + ik[i+1] +
iL[i+1])  )
         ibg.append(neurone['i_bg']*5)

         ica1.append(neurone['g_Ca'] * membraneFunction(x[i],
neurone)*100 )
         ica2.append(( x[i] - neurone['vRest_Ca'] ))
         w1.append(neurone['phi'] * lambdaFunction(x[i], neurone))
         w2.append(( k_FractionFunction(x[i], neurone) - y[i] ) )

    a = 0
    for i in range(i, i + 1000 ):

#         neurone['i_bg'] = iex * ( 1- a/1000)
         neurone['i_bg'] = (np.exp((1-a/1000) * np.log(iex + 1) ) -
1)
#         neurone['i_bg'] = (np.exp(1- a/1000) - 1) * iex/
(np.exp(1000) - 1)
         ibg.append(neurone['i_bg'])
```

```
        a += 1

##        RÃ©solution 1
#        x.append(x[i] + (dV_dt(x[i],y[i], neurone)) *
neurone['dt'])
#        y.append(y[i] + (dW_dt(x[i],y[i], neurone)) *
neurone['dt'])
#

#       RÃ©solution 2
        xfut = x[i] + (dV_dt(x[i],y[i], neurone)) * neurone['dt']
        yfut = y[i] + (dW_dt(x[i],y[i], neurone)) * neurone['dt']

        x.append(x[i] + neurone['dt']/2 * (dV_dt(x[i],y[i], neurone)
+ dV_dt(xfut,yfut, neurone)))
        y.append(y[i] + neurone['dt']/2 * (dW_dt(x[i],y[i], neurone)
+ dW_dt(xfut,yfut, neurone)))

##        Resolution 3 Runge-Kutta
#        k1 = neurone['dt'] * dV_dt(x[i], y[i], neurone)
#        l1 = neurone['dt'] * dW_dt(x[i], y[i], neurone)
#        k2 = neurone['dt'] * dV_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#        l2 = neurone['dt'] * dW_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#        k3 = neurone['dt'] * dV_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#        l3 = neurone['dt'] * dW_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#        k4 = neurone['dt'] * dV_dt(x[i] + k3, y[i] + l3, neurone)
#        l4 = neurone['dt'] * dW_dt(x[i] + k3, y[i] + l3, neurone)
#
#        x.append( x[i] +  (k1 + k2 + k2 + k3 + k3 + k4) / 6  )
#        y.append( y[i] + (l1 + l2 + l2 + l3 + l3 + l4) / 6  )
#
#     tracÃ© courantss
        ica.append(- neurone['g_Ca'] * membraneFunction(x[i],
neurone) * ( x[i] - neurone['vRest_Ca'] ))
        ik.append(- neurone['g_K'] * y[i] * (x[i] -
neurone['vRest_K']))
        iL.append(- neurone['g_L'] * (x[i] - neurone['vRest_L']))
        itot.append( neurone['i_bg'] + (ica[i+1] + ik[i+1] +
iL[i+1])  )

        ica1.append(neurone['g_Ca'] * membraneFunction(x[i],
neurone)*100 )
        ica2.append(( x[i] - neurone['vRest_Ca'] ))
        w1.append(neurone['phi'] * lambdaFunction(x[i], neurone))
        w2.append(( k_FractionFunction(x[i], neurone) - y[i] ) )



    neurone['i_bg'] = 0
    for i in range(i, i + tfin ):
##        RÃ©solution 1
#        x.append(x[i] + (dV_dt(x[i],y[i], neurone)) *
```

```
neurone['dt'])
#        y.append(y[i] + (dW_dt(x[i],y[i], neurone)) *
neurone['dt'])
#

#       RÃ©solution 2
        xfut = x[i] + (dV_dt(x[i],y[i], neurone)) * neurone['dt']
        yfut = y[i] + (dW_dt(x[i],y[i], neurone)) * neurone['dt']

        x.append(x[i] + neurone['dt']/2 * (dV_dt(x[i],y[i], neurone)
+ dV_dt(xfut,yfut, neurone)))
        y.append(y[i] + neurone['dt']/2 * (dW_dt(x[i],y[i], neurone)
+ dW_dt(xfut,yfut, neurone)))

##        Resolution 3 Runge-Kutta
#        k1 = neurone['dt'] * dV_dt(x[i], y[i], neurone)
#        l1 = neurone['dt'] * dW_dt(x[i], y[i], neurone)
#        k2 = neurone['dt'] * dV_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#        l2 = neurone['dt'] * dW_dt(x[i] + k1 / 2, y[i] + l1 / 2,
neurone)
#        k3 = neurone['dt'] * dV_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#        l3 = neurone['dt'] * dW_dt(x[i] + k2 / 2, y[i] + l2 / 2,
neurone)
#        k4 = neurone['dt'] * dV_dt(x[i] + k3, y[i] + l3, neurone)
#        l4 = neurone['dt'] * dW_dt(x[i] + k3, y[i] + l3, neurone)
#
#        x.append( x[i] +  (k1 + k2 + k2 + k3 + k3 + k4) / 6  )
#        y.append( y[i] + (l1 + l2 + l2 + l3 + l3 + l4) / 6  )
#
#     tracÃ© courantss
        ica.append(- neurone['g_Ca'] * membraneFunction(x[i],
neurone) * ( x[i] - neurone['vRest_Ca'] ))
        ik.append(- neurone['g_K'] * y[i] * (x[i] -
neurone['vRest_K']))
        iL.append(- neurone['g_L'] * (x[i] - neurone['vRest_L']))
        itot.append( neurone['i_bg'] + (ica[i+1] + ik[i+1] +
iL[i+1])  )
        ibg.append(neurone['i_bg'])


        ica1.append(neurone['g_Ca'] * membraneFunction(x[i],
neurone)*100 )
        ica2.append(( x[i] - neurone['vRest_Ca'] ))
        w1.append(neurone['phi'] * lambdaFunction(x[i], neurone))
        w2.append(( k_FractionFunction(x[i], neurone) - y[i] ) )

#    neurone['i_bg'] = 0
#
#    x_excite = []
#    y_excite = []
#
#    x_excite.append(vMi)
#    y_excite.append(ni)
#
```

```
#    for i in range(time):
#
#        x_excite.append(x_excite[i] +
(dV_dt(x_excite[i],y_excite[i], neurone)) * neurone['dt'])
#        y_excite.append(y_excite[i] +
(dW_dt(x_excite[i],y_excite[i], neurone)) * neurone['dt'])
#        #z.append(z[i] + (h(x[i],y[i],z[i])) * dt)

    for i in range(len(iL)):
        iL[i] = -iL[i]
        ik[i] = -ik[i]

#    null_val_x_dV, null_val_y_dV = resolution(-120, 60, 0, 0.5,
0.1, 0.005, dV_dt, neurone)
#    null_val_x_dW, null_val_y_dW = resolution(-120, 60, 0, 0.5,
0.1, 0.005, dW_dt, neurone)

 #plot
    fig = plt.figure(figsize=(20,10))
    fig.subplots_adjust(wspace = 0.5, hspace = 0.3)
    ax1 = fig.add_subplot(2,3,1)
    ax2 = fig.add_subplot(2,3,3)
    ax3 = fig.add_subplot(2,3,2)
    ax4 = fig.add_subplot(2,3,4)
    ax5 = fig.add_subplot(2,3,5)
    ax6 = fig.add_subplot(2,3,6)


#    ax1.plot(y, 'b-', label='n')
#    ax1.plot(x_excite, 'r--', label='i_bg = '+
str(neurone['i_bg']))
#    ax1.plot(y_excite, 'b--', label='n')
    #ax1.plot(z, 'g-', label='prey')
    ax1.plot(x, 'r-', label='vMembrane')
    ax1.set_title("vM")
    ax1.set_xlabel("time")
    ax1.grid()

    ax1.legend(loc='best')
#
    ax2.plot(x, y, color="blue")
    ax2.set_xlabel("vM")
    ax2.set_ylabel("n")
#    ax2.plot(null_val_x_dV, null_val_y_dV)
#    ax2.plot(null_val_x_dW, null_val_y_dW)
    ax2.set_title("Phase space")
    ax2.grid()

    ax3.plot(y, 'r-', label='n')
    ax3.plot(w1, 'b-', label='w1')
    ax3.plot(w2, 'g-', label='w2')
    ax3.set_title("w")
    ax3.set_xlabel("time")
    ax3.grid()
    ax3.legend(loc='best')
```

```
    ax4.plot(itot, 'r-', label='i')
    ax4.set_title("courant total")

    ax4.set_xlabel("time")
    ax4.grid()
    ax4.legend(loc='best')

    ax5.plot(ica, 'b-', label='ica')
    ax5.plot(ik, 'g-', label='iK')
    ax5.plot(iL, 'r-',label='iL')
    ax5.plot(ibg, 'y-',label='ibg')
    ax5.set_title("courants")
    ax5.set_xlabel("time")
    ax5.grid()
    ax5.legend(loc='best')

    ax6.plot(ica, 'b-')
    ax6.plot(ica1, 'g-')
    ax6.plot(ica2, 'r-')
    ax6.plot(ibg, 'y-')
    ax6.set_title("ica")
    ax6.set_xlabel("time")

    ax6.grid()
    ax6.legend(loc='best')

    plt.show()



#%%

# tracÃ© des courbes d'intÃ©rÃªt pour le neurone bio
neurone = neurone_bio.copy()

print(neurone_bio['phi'])

v_m1 = neurone['v_m1']
g_L = neurone['g_L']
vRest_L = neurone['vRest_L']
phi = neurone['phi']
cMembrane = neurone['cMembrane']
v_m2 = neurone['v_m2']
v_w1 = neurone['v_w1']
v_w2 = neurone['v_w2']
vRest_K = neurone['vRest_K']
vRest_Ca = neurone['vRest_Ca']
g_Ca = neurone['g_Ca']

for iex in range(19,21):
    for g_L in range(140,141,1):
        g_L = g_L/100


        for vRest_L in range(-100,-99,1):
```

```
#            for phi in range(3,688,10):
#                phi = phi/1000
            for phi in range(140,150,10):
                phi = phi/1000
    #            for v_m1 in range(-40,5,5):
    #                for cMembrane in range(1,10):
    #                    cMembrane = cMembrane /20
    #                    for v_m2 in range(40,45):
    #            for v_w1 in range(-40,10,10):
    ##                                for v_w2 in  range(40,45):
    #            for vRest_K in range(-200,0,20):
    #            for vRest_Ca in range(0,200,20):
    ##                                        for g_Ca in
range(0,20):

                print("paramÃ¨tres : ",
        #                  "v_m1 = ",v_m1,
                      "g_L = ",g_L,
                      "vRest_L = ",vRest_L,
                      "phi =",phi,
        #                        "cMembrane = ",cMembrane,
        #                        "v_m2 = ",v_m2,
    #                              "v_m1 = ",v_m1,
    #                                 "v_w1 = ",v_w1,
        #                        "v_w2 = ",v_w2,
#                               "vRest_K = ",vRest_K,
#                               "vRest_Ca = ",vRest_Ca,
        #                        "g_Ca = ",g_Ca
                        "iex =" , iex
                        )
                neurone = neurone_bio.copy()#iv1, iv2 = initial
values, dt = timestep, time = range
                neurone['v_m1'] = v_m1
                neurone['g_L'] = g_L
                neurone['vRest_L'] = vRest_L
                neurone['phi'] = phi
                neurone['cMembrane'] = cMembrane
                neurone['v_m2'] = v_m2
                neurone['v_w1'] = v_w1
                neurone['v_w2'] = v_w2
                neurone['vRest_K'] = vRest_K
                neurone['vRest_Ca'] = vRest_Ca
                neurone['g_Ca'] = g_Ca
                sys(0, 0, 20000, 0, 10000, 20, neurone)

#%%

# tracÃ© des courbes d'intÃ©rÃªt pour le neurone standard
neurone = neurone_simbrain.copy()

print(neurone_bio['phi'])

v_m1 = neurone['v_m1']
g_L = neurone['g_L']
vRest_L = neurone['vRest_L']
phi = neurone['phi']
```

```
cMembrane = neurone['cMembrane']
v_m2 = neurone['v_m2']
v_w1 = neurone['v_w1']
v_w2 = neurone['v_w2']
vRest_K = neurone['vRest_K']
vRest_Ca = neurone['vRest_Ca']
g_Ca = neurone['g_Ca']

for iex in range(200,201,5):
    for g_L in range(200,201,1):
        g_L = g_L/100


        for vRest_L in range(-60,-59):

#            for phi in range(3,688,10):
#                phi = phi/1000
            for phi in range(66,67,1):
                phi = phi/1000
    #            for v_m1 in range(-40,5,5):
    #                for cMembrane in range(1,10):
    #                    cMembrane = cMembrane /20
    #                    for v_m2 in range(40,45):
    #            for v_w1 in range(-40,10,10):
    ##                                for v_w2 in  range(40,45):
    #            for vRest_K in range(-200,0,20):
    #            for vRest_Ca in range(0,200,20):
    ##                                            for g_Ca in
range(0,20):

                print("paramÃ¨tres : ",
        #                  "v_m1 = ",v_m1,
                      "g_L = ",g_L,
                      "vRest_L = ",vRest_L,
                      "phi =",phi,
        #                        "cMembrane = ",cMembrane,
        #                        "v_m2 = ",v_m2,
    #                              "v_m1 = ",v_m1,
    #                                 "v_w1 = ",v_w1,
        #                        "v_w2 = ",v_w2,
#                               "vRest_K = ",vRest_K,
#                               "vRest_Ca = ",vRest_Ca,
        #                       "g_Ca = ",g_Ca
                        "iex =" , iex
                        )
                neurone = neurone_bio.copy()#iv1, iv2 = initial
values, dt = timestep, time = range
                neurone['v_m1'] = v_m1
                neurone['g_L'] = g_L
                neurone['vRest_L'] = vRest_L
                neurone['phi'] = phi
                neurone['cMembrane'] = cMembrane
                neurone['v_m2'] = v_m2
                neurone['v_w1'] = v_w1
                neurone['v_w2'] = v_w2
                neurone['vRest_K'] = vRest_K
```

```
neurone['vRest_Ca'] = vRest_Ca
neurone['g_Ca'] = g_Ca
sys(0, 0,10000, 100000, 20000, iex , neurone)
```

Tracé des Isoclines en zéro

```
# -*- coding: utf-8 -*-
"""
Created on Mon Sep  4 00:11:16 2017

@author: Anthony Ozier-lafontaine
"""

neurone_bio = {'g_Ca': 6.9,
               'g_K': 10,
               'g_L': 0.9,
               'vRest_Ca': 100,
               'vRest_K': -100,
               'vRest_L': -100,
               'cMembrane': 0.05,
               'v_m1': 0,
               'v_m2': 40,
               'v_w1': 0,
               'v_w2': 40,
               'phi': 0.3,
               'i_bg': 0,
               'dt' : 0.001,
               "activation": 0,
               "n": 0}

neurone_simbrain = {'g_Ca': 4,
                    'g_K': 8,
                    'g_L': 2,
                    'vRest_Ca': 120,
                    'vRest_K': -80,
                    'vRest_L': -60,
                    'cMembrane': 5,
                    'v_m1': -1.2,
                    'v_m2': 18,
                    'v_w1': 2,
                    'v_w2': 17.4,
                    'phi': 0.066667,
                    'i_bg': 0,
                    'dt' : 0.1,
                    "activation": 0,
                    "n": 0}

 #%%

# Cette portion de code permet de dÃ©finir le traÃ§age des courbes
d'intÃ©rÃªt et celui des isoclines en zÃ©ro
# import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
# show plots in notebook
# matplotlib inline

# define system in terms of separated differential equations
```

```
neurone = neurone_bio.copy()

def f(x,y):
    return 2*x - x**2 - x*y

def dV_dt(x, y, neurone):
    i_Ca = neurone['g_Ca'] * membraneFunction(x, neurone) * ( x -
neurone['vRest_Ca'] )
    i_K = neurone['g_K'] * y * (x - neurone['vRest_K'])
    i_L = neurone['g_L'] * (x - neurone['vRest_L'])
    i_ion = i_Ca + i_K + i_L
    return (neurone['i_bg'] - i_ion)/neurone['cMembrane']

def membraneFunction(x, neurone):
    return 0.5 * (1 + np.tanh( (x - neurone['v_m1']) /
neurone['v_m2'] ))


def dW_dt(x, y, neurone):
    return neurone['phi'] * lambdaFunction(x, neurone) * (
k_FractionFunction(x, neurone) - y )

def k_FractionFunction ( x, neurone ) :
    return 0.5 * ( 1 + np.tanh( ( x - neurone['v_w1'] ) /
neurone['v_w2'] ))

def lambdaFunction(x, neurone):
    return np.cosh( (x - neurone['v_w1']) / (2 * neurone['v_w2']))

def g(x,y):
    return - y + x*y

def sign(x) :
    if x >= 0 :
        return True
    else :
        return False


# isoclines en zÃ©ro
def resolution(xmin, xmax, ymin, ymax, xstep, ystep, f, neurone):
    x = xmin
    y = ymin

    dV = f (x, y, neurone)
    a = sign(dV)
    b = a

    null_val_x = []
    null_val_y = []
    while x < xmax :
        y = ymin

        if b != a :
            b = a
```

```
        while y < ymax :

            dV = f (x, y, neurone)


            if sign(dV) != b :
#                print('x =' + str (x) + ' y =' + str(y) + 'dv =' +
str(dV))
                null_val_x += [x]
                null_val_y += [y]
                b = sign(dV)
                x += xstep
                y += ystep


            y += ystep



        x += xstep
    return null_val_x, null_val_y
#%%
# Cette portion de code trace les isoclines en zÃ©ro

for i_bg in range(0, 60, 20):
    neurone = neurone_bio.copy()


    neurone['i_bg'] = i_bg
#    sys(0, 0, 50000, neurone)
    null_val_x_dV, null_val_y_dV = resolution(-100, 100, 0, 1, 0.1,
0.005, dV_dt, neurone)
    null_val_x_dW, null_val_y_dW = resolution(-100, 100, 0, 1, 0.1,
0.005, dW_dt, neurone)
    plt.plot(null_val_x_dV, null_val_y_dV)
    plt.plot(null_val_x_dW, null_val_y_dW)
    plt.xlabel('membrane voltage')
    plt.ylabel('n')
    plt.title('nullclines pour excitation Ã©gale Ã  0, 20 et 40
microA/cmÂ²')
```

Tracé des courbes d'ISI

```
# -*- coding: utf-8 -*-
"""
Created on Mon Sep  4 00:27:38 2017

@author: Anthony Ozier-lafontaine
"""

import numpy as np
import matplotlib.pyplot as plt

#Init neuron

neurone_bio = {'g_Ca': 6.9,
               'g_K': 10,
               'g_L': 0.895,
               'vRest_Ca': 100,
               'vRest_K': -100,
               'vRest_L': -100,
               'cMembrane': 0.05,
               'v_m1': 0,
               'v_m2': 40,
               'v_w1': 0,
               'v_w2': 40,
               'phi': 0.3,
               'i_bg': 20,
               'dt' : 0.001,
               "activation": 0,
               "n": 0}

neurone_fast = {'g_Ca': 207,
               'g_K': 300,
               'g_L': 0.85,
               'vRest_Ca': 100,
               'vRest_K': -100,
               'vRest_L': -100,
               'cMembrane': 0.004,
               'v_m1': 0,
               'v_m2': 40,
               'v_w1': 0,
               'v_w2': 40,
               'phi': 881,
               'i_bg': 0,
               'dt' : 0.00001,
               "activation": 0,
               "n": 0}

neurone_simbrain = {'g_Ca': 4,
                    'g_K': 8,
                    'g_L': 2,
                    'vRest_Ca': 120,
                    'vRest_K': -80,
                    'vRest_L': -60,
                    'cMembrane': 5,
```

```
                    'v_m1': -1.2,
                    'v_m2': 18,
                    'v_w1': 2,
                    'v_w2': 17.4,
                    'phi': 0.066667,
                    'i_bg': 0,
                    'dt' : 0.1,
                    "activation": 0,
                    "n": 0}
#%% Equations du modÃ¨le


def update(neurone) :
    dt =neurone['dt']
    i_syn = 0
    vMembrane = neurone["activation"]
    w_K = neurone["n"]
    dVdt = dV_dt(vMembrane, i_syn, w_K, neurone)
    dWdt = dW_dt(vMembrane, w_K, neurone)
    vmFut = vMembrane + dt * dVdt
    w_kFut = w_K + dt * dWdt
    vMembrane = vMembrane + (dt / 2) * (dVdt + dV_dt(vmFut, i_syn,
w_K, neurone))
    w_K = w_K + (dt / 2) * (dWdt + dW_dt(vMembrane,w_kFut, neurone))
    neurone["activation"] = vMembrane
    neurone["n"] = w_K

    #print(neurone)

def dV_dt(vMembrane, i_syn, w_K, neurone):
    i_Ca = neurone['g_Ca'] * membraneFunction(vMembrane, neurone) *
( vMembrane - neurone['vRest_Ca'] )
    i_K = neurone['g_K'] * w_K * (vMembrane - neurone['vRest_K'])
    i_L = neurone['g_L'] * (vMembrane - neurone['vRest_L'])
    i_ion = i_Ca + i_K + i_L
    return (neurone['i_bg'] + i_syn - i_ion)/neurone['cMembrane']

def dW_dt(vMembrane, w_K, neurone):
    return neurone['phi'] * lambdaFunction(vMembrane, neurone) * (
k_FractionFunction(vMembrane, neurone) - w_K )

def membraneFunction(vMembrane, neurone):
    return 0.5 * (1 + np.tanh( (vMembrane - neurone['v_m1']) /
neurone['v_m2'] ))

def k_FractionFunction ( vMembrane, neurone ) :
    return 0.5 * ( 1 + np.tanh( ( vMembrane - neurone['v_w1'] ) /
neurone['v_w2'] ))

def lambdaFunction(vMembrane, neurone):
    return np.cosh( (vMembrane - neurone['v_w1']) / (2 *
neurone['v_w2']))


#%% Calcul des ISI selon i_bg
```

```
# Cette portion de code permet de tracer la courbe d'ISI d'un
paramÃ©trage donnÃ©

def sign(x) :
    if x >= 0 :
        return True
    else :
        return False


neurone = neurone_simbrain.copy()
isi = []    #isi du neurone selon i_bg
imax = []   #max du spike selon i_bg

values_ibg = [] #Valeurs du courant d'excitation du neurone
for v in range(1,10,1):
    values_ibg += [v/100]
for v in range(10,100,5):
    values_ibg += [v/100]
for v in range(10,100,5):
    values_ibg += [v/10]
for v in range(10,100,5):
    values_ibg += [v/1]


for i_bg in values_ibg:
    neurone = neurone_bio.copy()
    neurone['i_bg'] = i_bg
#    neurone['i_bg'] = 1
    iex = [0]*20000 + [neurone['i_bg']]*10000 + [0]*50000

    val = []    #activation du neurone au cours du temps
    n = []      #ouverture des canaux ioniques au cours du temps
    t = []    #position des sommets des spikes lorsque le neurone
est excitÃ©
    a_1 = True
    imax_intermediaire = []
    for i in range(80000):

        neurone['i_bg'] = iex[i]
    #    val0 += [neurone0["activation"]]
    #    val1 += [neurone1["activation"]]
        val += [neurone['activation']]
        n += [neurone['n']]
        update(neurone)
        if i > 20000 and i < 30000 :
            a_2 = a_1
            x2 = neurone['activation']
            a_1 = sign(x2 - x1)
            if a_2 and (not a_1) :
                t += [i/1000]
                imax_intermediaire += [neurone['activation']]
        x1 = neurone['activation']

    #    update(neurone1)
 #   plt.plot(val)
```

```
 #   plt.plot(iex)
  #  plt.title('i_bg =' + str(i_bg) )
  #  plt.show()
  #  plt.plot(n)
  #  plt.title('i_bg =' + str(i_bg))
  #  plt.show()
  #  print(i, len(i))
    if len(t)>1:
        isi += [t[-1] - t[-2]]
    imax += [imax_intermediaire[-1]]
plt.plot(values_ibg,isi)
plt.title('isi en fonction de i_bg')
plt.ylabel('isi en ms')
plt.show()
```

Test paramètres modèle Morris-Lecar

```
# -*- coding: utf-8 -*-
"""
Created on Mon Sep  4 00:30:08 2017

@author: Anthony Ozier-lafontaine
"""

import numpy as np
import matplotlib.pyplot as plt

#Init neuron

neurone_bio = {'g_Ca': 6.9,
               'g_K': 10,
               'g_L': 0.895,
               'vRest_Ca': 100,
               'vRest_K': -100,
               'vRest_L': -100,
               'cMembrane': 0.05,
               'v_m1': 0,
               'v_m2': 40,
               'v_w1': 0,
               'v_w2': 40,
               'phi': 0.3,
               'i_bg': 20,
               'dt' : 0.001,
               "activation": 0,
               "n": 0}

neurone_fast = {'g_Ca': 207,
               'g_K': 300,
               'g_L': 0.85,
               'vRest_Ca': 100,
               'vRest_K': -100,
               'vRest_L': -100,
               'cMembrane': 0.004,
               'v_m1': 0,
               'v_m2': 40,
               'v_w1': 0,
               'v_w2': 40,
               'phi': 881,
               'i_bg': 0,
               'dt' : 0.00001,
               "activation": 0,
               "n": 0}

neurone_simbrain = {'g_Ca': 4,
                    'g_K': 8,
                    'g_L': 2,
                    'vRest_Ca': 120,
                    'vRest_K': -80,
                    'vRest_L': -60,
                    'cMembrane': 5,
```

```
                    'v_m1': -1.2,
                    'v_m2': 18,
                    'v_w1': 2,
                    'v_w2': 17.4,
                    'phi': 0.066667,
                    'i_bg': 0,
                    'dt' : 0.1,
                    "activation": 0,
                    "n": 0}
#%% Equations du modÃ¨le


def update(neurone) :
    dt =neurone['dt']
    i_syn = 0
    vMembrane = neurone["activation"]
    w_K = neurone["n"]
    dVdt = dV_dt(vMembrane, i_syn, w_K, neurone)
    dWdt = dW_dt(vMembrane, w_K, neurone)
    vmFut = vMembrane + dt * dVdt
    w_kFut = w_K + dt * dWdt
    vMembrane = vMembrane + (dt / 2) * (dVdt + dV_dt(vmFut, i_syn,
w_K, neurone))
    w_K = w_K + (dt / 2) * (dWdt + dW_dt(vMembrane,w_kFut, neurone))
    neurone["activation"] = vMembrane
    neurone["n"] = w_K

    #print(neurone)

def dV_dt(vMembrane, i_syn, w_K, neurone):
    i_Ca = neurone['g_Ca'] * membraneFunction(vMembrane, neurone) *
( vMembrane - neurone['vRest_Ca'] )
    i_K = neurone['g_K'] * w_K * (vMembrane - neurone['vRest_K'])
    i_L = neurone['g_L'] * (vMembrane - neurone['vRest_L'])
    i_ion = i_Ca + i_K + i_L
    return (neurone['i_bg'] + i_syn - i_ion)/neurone['cMembrane']

def dW_dt(vMembrane, w_K, neurone):
    return neurone['phi'] * lambdaFunction(vMembrane, neurone) * (
k_FractionFunction(vMembrane, neurone) - w_K )

def membraneFunction(vMembrane, neurone):
    return 0.5 * (1 + np.tanh( (vMembrane - neurone['v_m1']) /
neurone['v_m2'] ))

def k_FractionFunction ( vMembrane, neurone ) :
    return 0.5 * ( 1 + np.tanh( ( vMembrane - neurone['v_w1'] ) /
neurone['v_w2'] ))

def lambdaFunction(vMembrane, neurone):
    return np.cosh( (vMembrane - neurone['v_w1']) / (2 *
neurone['v_w2']))

    #%% Courbes test
# Cette portion de code permet de tester plusieurs valeurs des
mÃªmes paramÃ¨tres
```

```
# Ici, ce sont les valeurs de g_L et vRest_L qui sont testÃ©es
# La courbe tracÃ©e est l'Ã©volution de vM au cours du temps pour un
neurone excitÃ© pendant 10 000 itÃ©rations (10 ms si dt = 0.001)
neurone = neurone_bio.copy()
#Trouver le bon g_L et le bon vRest_L


for g_L in range (800,1000,200):
    for vRest_L in range(-150,-80,50):

        neurone = neurone_bio.copy()
        # Valeurs prises par iex : 0 pendant 10000 itÃ©rations puis
i_bg pendant 10000 puis 0 pendant 50000 itÃ©rations
        iex = [0]*10000 + [neurone['i_bg']]*10000 + [0]*50000
#         neurone['phi'] = 0.3
        neurone['g_L'] = g_L/1000
        neurone['vRest_L'] = vRest_L


        val = []
        for i in range(70000):  # 70000 = 10000 + 10000 + 50000
            neurone['i_bg'] = iex[i]
        #    val0 += [neurone0["activation"]]
        #    val1 += [neurone1["activation"]]
            val += [neurone['activation']]
            update(neurone)
        #    update(neurone1)
        plt.plot(val)
        plt.plot(iex)
        plt.title('vRest_L =' + str(neurone['vRest_L']) + ' g_L = '
+ str(neurone['g_L']))
        plt.show()
```

Tableau input table et dataworld

```
# -*- coding: utf-8 -*-
"""
Created on Mon May 15 15:12:28 2017

@author: Anthony Ozier-lafontaine
"""


#%% Reservoir computing


#%%

#Ce script est permet d'automatiser l'apprentissage du 'reservoir
computing' modÃ©lisÃ© sur Simbrain
# Le tableau gÃ©nÃ©rÃ© est fait pour Ãªtre connectÃ© aux 9 neurones
excitateurs de l'Ã©cran et aux 4 neurones excitateurs des neurones
de sortie

import os
os.chdir('C:/Users/ASUS/Documents/SNN/simbrain-master/simbrain-
master/simulations/tables') # Adresse de l'endroit oÃ¹ l'on souhaite
enregistrer le tableau

import csv
import random

#Formes
iex = 65 #Courant d'excitation des neurones linÃ©aires activateurs
de l'Ã©cran ou des neurones de sortie

# input Ecran (ec) pour un square (sq) sur un Ã©cran de taille 7*7
with open('exandsup2.15.1.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ')
    ex = []
    sup = []

    #Liste correspondant Ã  une ligne pendant le repos (un zÃ©ro
pour chaque pixel et une virgule entre chaque)

    noactivity =
[0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0
,',',0]
    #Exitation verticale |||
    or0 =
[0,',',iex,',',0,',',0,',',iex,',',0,',',0,',',iex,',',0,',',0,',',0
,',',0,',',0]
    #Exitation Horizontale __
    or1 =
[0,',',0,',',0,',',iex,',',iex,',',iex,',',0,',',0,',',0,',',0,',',0
,',',0,',',0]
    #Exitation Diagonale \\\
```

```
    or2 =
[iex,',',0,',',0,',',0,',',iex,',',0,',',0,',',0,',',iex,',',0,',',0
,',',0,',',0]
    #Exitation Diagonale ///
    or3 =
[0,',',0,',',iex,',',0,',',iex,',',0,',',iex,',',0,',',0,',',0,',',0
,',',0,',',0]

    # Supervision verticale |||
    sup0 =
[0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',65,',',0,',',
0,',',0]
    # Supervision Horizontale __
    sup1 =
[0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',65,',',
0,',',0]
    # Supervision Diagonale \\\
    sup2 =
[0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',6
5,',',0]
    # Supervision Diagonale ///
    sup3 =
[0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0
,',',65]

    #Compteurs : on veut au moins 5 apprentissages par orientation
    compteur0 = 0
    compteur1 = 0
    compteur2 = 0
    compteur3 = 0
    compteurtot = 0
    # Remplissage du tableau input avec les lignes de repos pour que
les neurones aient un temps pour s'Ã©quiliblre
    for i in range(300):
        spamwriter.writerow(noactivity)

    while (compteur0 < 15 or compteur1 < 15 or compteur2 < 15 or
compteur3 < 15 ):
        a = random.randrange(0,4)
        compteurtot += 1
        if a == 0:
            compteur0 += 1
            for i in range(100):
                spamwriter.writerow(or0)
            for i in range(150):
                spamwriter.writerow(noactivity)
            for i in range(100):
                spamwriter.writerow(sup0)
            for i in range(300):
                spamwriter.writerow(noactivity)

        if a == 1:
            compteur1 += 1
            for i in range(100):
                spamwriter.writerow(or1)
            for i in range(150):
```

```
                spamwriter.writerow(noactivity)
            for i in range(100):
                spamwriter.writerow(sup1)
            for i in range(300):
                spamwriter.writerow(noactivity)

        if a == 2:
            compteur2 += 1
            for i in range(100):
                spamwriter.writerow(or2)
            for i in range(150):
                spamwriter.writerow(noactivity)
            for i in range(100):
                spamwriter.writerow(sup2)
            for i in range(300):
                spamwriter.writerow(noactivity)

        if a == 3:
            compteur3 += 1
            for i in range(100):
                spamwriter.writerow(or3)
            for i in range(150):
                spamwriter.writerow(noactivity)
            for i in range(100):
                spamwriter.writerow(sup3)
            for i in range(300):
                spamwriter.writerow(noactivity)

    #test  version 2.0
    for i in range(500):
        spamwriter.writerow(noactivity)
    for i in range(100):
        spamwriter.writerow(or0)
    for i in range(300):
        spamwriter.writerow(noactivity)
    for i in range(100):
        spamwriter.writerow(or1)
    for i in range(300):
        spamwriter.writerow(noactivity)
    for i in range(100):
        spamwriter.writerow(or2)
    for i in range(300):
        spamwriter.writerow(noactivity)
    for i in range(100):
        spamwriter.writerow(or3)
    for i in range(300):
        spamwriter.writerow(noactivity)



    spamwriter.writerow(
[compteurtot,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',com
pteur0,',',compteur1,',',compteur2,',',compteur3])
```

```
  #%%
import os
os.chdir('C:/Users/ASUS/Documents/SNN/simbrain-master/simbrain-
master/simulations/tables')

import csv
import random

# Ce script n'automatise que l'allumage de l'Ã©cran et permet de
tester le rÃ©seau une fois que l'apprentissage a convergÃ©


#Formes
iex = 65

# input Ecran (ec) pour un square (sq) sur un Ã©cran de taille 7*7
with open('IRCICA_testReservoir.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ')
    ex = []
    sup = []

    #Liste correspondant Ã  une ligne pendant le repos (un zÃ©ro
pour chaque pixel et une virgule entre chaque)

    noactivity = [0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0]
    #Exitation verticale |||
    or0 = [0,',',iex,',',0,',',0,',',iex,',',0,',',0,',',iex,',',0]
    #Exitation Horizontale __
    or1 = [0,',',0,',',0,',',iex,',',iex,',',iex,',',0,',',0,',',0]
    #Exitation Diagonale \\\
    or2 = [iex,',',0,',',0,',',0,',',iex,',',0,',',0,',',0,',',iex]
    #Exitation Diagonale ///
    or3 = [0,',',0,',',iex,',',0,',',iex,',',0,',',iex,',',0,',',0]

    for i in range(300):
        spamwriter.writerow(noactivity)
    for i in range(100):
        spamwriter.writerow(or0)
    for i in range(300):
        spamwriter.writerow(noactivity)
    for i in range(100):
        spamwriter.writerow(or1)
    for i in range(300):
        spamwriter.writerow(noactivity)
    for i in range(100):
        spamwriter.writerow(or2)
    for i in range(300):
        spamwriter.writerow(noactivity)
    for i in range(100):
        spamwriter.writerow(or3)
    for i in range(400):
        spamwriter.writerow(noactivity)
```

Extrait du tableau généré

0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0
0,65,0,0,65,0,0,65,0