

MASTER 1 - PJI

Sujet n°116

---

# Réalisation d'un simulateur de réseaux de neurones artificiels

---

*Auteur(s) :*  
Quentin BAILLEUL  
William GOUZER

*Responsable(s) :*  
P. BOULET

19 juin 2014



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Sujet $n^{\circ}116$	2
1.2	Motivations	2
<b>2</b>	<b>Analyse - Cahier des charges</b>	<b>3</b>
2.1	État de l'art	3
2.1.1	Le memristor	3
2.1.2	Le Simulateur	3
2.2	Proposition d'un modèle	4
<b>3</b>	<b>Réseau de Neurones</b>	<b>5</b>
3.1	Définition	5
3.2	Utilité	5
3.3	Modèle choisi	6
<b>4</b>	<b>Introduction au simulateur N2S3</b>	<b>7</b>
4.1	Scala	8
4.2	Akka	8
<b>5</b>	<b>Implémentation de N2S3</b>	<b>9</b>
5.1	Neurone	9
5.1.1	Code	9
5.1.2	Structure	9
5.2	Synapse	10
5.2.1	Code	10
5.2.2	Structure	10
5.3	Réseau	11
5.3.1	Code	11
5.3.2	Structure	12
5.4	Démonstration	13
5.4.1	Code	13
5.4.2	Sortie	14
5.4.3	Structure	14
<b>6</b>	<b>Conclusion</b>	<b>15</b>
6.1	Difficultés	15
6.2	Bilan	15
6.3	Perspectives d'améliorations	15
<b>7</b>	<b>Remerciements</b>	<b>17</b>
<b>8</b>	<b>Annexes</b>	<b>18</b>
8.1	Code d'un neurone	18
8.2	Code d'un synapse	20
8.3	Code du réseau	22
8.4	Messages envoyé par le réseau	26
8.5	Main du programme	28

# 1 Introduction

## 1.1 Sujet $n^{\circ}116$

Participation à la réalisation d'un simulateur de réseaux de neurones artificiels en Scala. Il s'agira de réaliser le cœur du moteur de simulation basé sur un modèle événementiel temporel. Cette réalisation doit permettre la scalabilité du simulateur. Selon l'avancement du développement, il y aura de nombreuses perspectives à ce travail.

## 1.2 Motivations

Nous (Quentin + William) avons choisi ce sujet pour des raisons très différentes. Quentin souhaitait développer ses connaissances dans les réseaux de neurones, quant à William, c'était pour l'aspect programmation avec un nouveau langage et un nouveau paradigme : Scala et la programmation fonctionnelle.

Nous sommes donc assez complémentaires pour la réalisation de ce projet, ce qui n'en sera que bénéfique.

## 2 Analyse - Cahier des charges

### 2.1 État de l’art

#### 2.1.1 Le memristor

Ce sujet a été motivé par l’émergence des memristors. Ce composant est capable de mémoriser son dernier état, il est donc non-volatile et a le potentiel de remplacer la majorité de nos mémoires comme :

- la mémoire morte (disque dur, flash)
- la mémoire vive (RAM)
- le caches de nos processeurs

Nous aurions donc un processeur travaillant directement avec de la RAM, non-volatile, sans besoin de caches ou d’un disque dur. L’impact d’un point de vue industriel est immense, des entreprises comme HP sont déjà à la pointe et souhaitent démocratiser ce genre de technologies.

De plus, certaines variétés de memristors sont capables d’avoir un comportement très similaire à ce qu’on appelle la plasticité neuronale (i.e. se développer ou régresser en fonction de son implication dans le réseau). Ainsi, le memristor peut “oublier” les informations s’il n’est pas stimulé, tout comme le comportement de nos propres neurones.

#### 2.1.2 Le Simulateur

On comprend très aisément l’importance de ces composants pour la recherche et l’industrie, mais quel est le lien avec notre simulateur de neurones ?

Il existe une grande variété de technologies pour réaliser le comportement d’un memristor mais il y a beaucoup de difficultés à accumuler un grand nombre d’entre eux sur un même circuit et comparer leur efficacité. Il faut donc pouvoir simuler logiciellement ces composants pour anticiper leurs comportements au sein d’un réseau (de type crossbar par exemple). Le but est vraiment de reproduire le comportement physique de ces memristors de manière efficace pour passer à l’échelle. En effet, la modélisation se traduit par des systèmes d’équations différentielles qui sont très coûteux en temps de calcul ; d’autres simulateurs comme “The Brian Spiking Neural Network Simulator”, codé en Python, souffre non seulement de la limite du langage mais aussi de la complexité de ces calculs.

Nous allons donc proposer un modèle permettant de passer à l’échelle avec :

- le langage Scala qui permet une bonne scalabilité
  - la librairie d’acteurs “Akka” pour représenter les neurones/memristors
- les équations différentielles précalculées et stockées au sein de chaque acteur

## 2.2 Proposition d'un modèle

Pour le modèle de base nous nous inspirons de “Brian simulator”. Brian est destiné aux chercheurs élaborant des modèles fondés sur les réseaux de neurones impulsionnels (networks of spiking neurons). La conception vise à minimiser le temps de développement des utilisateurs, avec une vitesse d'exécution d'un objectif secondaire. Les utilisateurs spécifient les modèles de neurones en donnant leurs équations différentielles sous forme mathématique standard. L'exemple suivant, va créer des groupes de neurones et les connectant via les synapses :

$$\frac{dv}{dt} = \frac{g_e + g_i - (v + 49mV)}{20ms} .$$

Le but est de rendre le processus aussi souple que possible, pour que les chercheurs ne soient pas limités à l'utilisation de modèles de neurones déjà intégrés dans le simulateur. Initialement, le simulateur est entièrement écrit en Python, en utilisant les paquets de calcul numérique et scientifique Numpy et SciPy (nous prendrons donc l'équivalent Java de ces deux bibliothèques). Certaines parties du simulateur peuvent éventuellement être exécutées à l'aide du code C généré à la volée. Pour le calcul, Brian utilise des techniques de vectorisation qui permettent, pour un grand nombre de neurones, d'obtenir une vitesse d'exécution équivalente au même algorithme programmé en C. Par exemple :

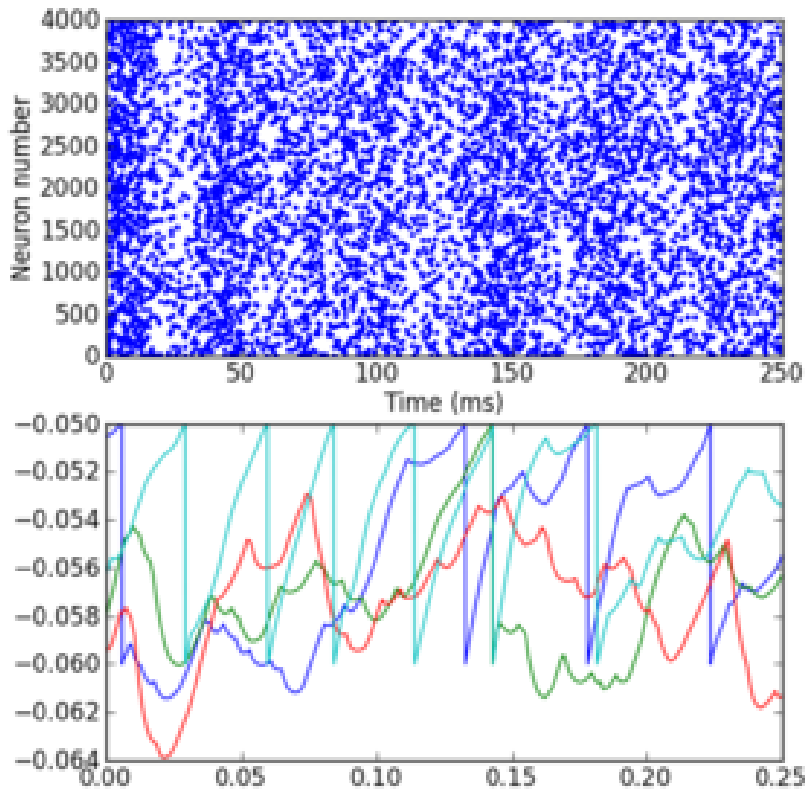


FIGURE 1 – Graphiques produits par la configuration «par défaut» de Brian

## 3 Réseau de Neurones

### 3.1 Définition

Un réseau de neurones artificiels est un modèle de calcul dont la conception est très schématiquement inspirée du fonctionnement des neurones biologiques.

Les réseaux de neurones sont généralement optimisés par des méthodes d'apprentissage de type probabiliste, en particulier bayésien. Ils sont placés d'une part dans la famille des applications statistiques, qu'ils enrichissent avec un ensemble de paradigmes permettant de créer des classifications rapides (réseaux de Kohonen en particulier), et d'autre part dans la famille des méthodes de l'intelligence artificielle auxquelles ils fournissent un mécanisme perceptif indépendant des idées propres de l'implémenteur, et fournissant des informations d'entrée au raisonnement logique formel.

En modélisation des circuits biologiques, ils permettent de tester quelques hypothèses fonctionnelles issues de la neurophysiologie, ou encore les conséquences de ces hypothèses pour les comparer au réel.

### 3.2 Utilité

Les réseaux de neurones, en tant que système capable d'apprendre, mettent en œuvre le principe de l'induction, c'est-à-dire l'apprentissage par l'expérience. Par confrontation avec des situations ponctuelles, ils infèrent un système de décision intégré dont le caractère générique est fonction du nombre de cas d'apprentissages rencontrés et de leur complexité par rapport à la complexité du problème à résoudre. Par opposition, les systèmes symboliques capables d'apprentissage, s'ils implémentent également l'induction, le font sur base de la logique algorithmique, par complexification d'un ensemble de règles déductives.

Grâce à leur capacité de classification et de généralisation, les réseaux de neurones sont généralement utilisés dans des problèmes de nature statistique tels que la classification automatique de codes postaux ou la prise de décision concernant un achat boursier en fonction de l'évolution des cours de la bourse. Dans un autre domaine, une banque peut créer un jeu de données sur les clients qui ont effectué un emprunt constitué : de leur revenu, de leur âge, du nombre d'enfants à charge et s'il s'agit d'un bon client. Si ce jeu de données est suffisamment grand, il peut être utilisé pour l'entraînement d'un réseau de neurones. La banque pourra alors présenter les caractéristiques d'un potentiel nouveau client, et le réseau répondra s'il sera bon client ou non, en généralisant à partir des cas qu'il connaît.

Si le réseau de neurones fonctionne avec des nombres réels, la réponse traduit une probabilité de certitude (de -1 « sera un mauvais client » à 1 pour « sera un bon client »).

Le réseau de neurones ne fournit pas toujours de règle exploitable par un humain. Le réseau reste souvent une boîte noire qui fournit une réponse quand on lui présente une donnée, mais le réseau ne fournit pas de justification facile à interpréter.

### 3.3 Modèle choisi

Notre simulateur crée des réseaux de neurones composés d'une succession de couches dont chacune prend ses entrées sur les sorties de la précédente. Chaque couche  $i$  est composée de  $N_i$  neurones, prenant leurs entrées sur les  $N_{i-1}$  neurones de la couche précédente.

À chaque synapse est associé un poids synaptique, de sorte que les  $N_{i-1}$  soient multipliés par ce poids, puis additionnés par les neurones de niveau  $i$ , ce qui est équivalent à multiplier le vecteur d'entrée par une matrice de transformation.



## 4 Introduction au simulateur N2S3

Notre simulateur a été baptisé N2S3 pour **N**eural **N**etwork **S**calable **S**pike **S**imulator. Ce projet est une adaptation de Brian avec un focus sur la scalabilité ; nous utilisons pour ce faire le langage *Scala* et la librairie d'acteurs *Akka*.

Le simulateur N2S3 fonctionne avec la structure suivante :

- le réseau est linéaire : chaque couche de neurones n'a accès qu'à une ou deux couches
- un neurone est relié avec un autre par le biais de synapses
- deux neurones sont connectés au plus par un synapse
- un synapse ou un neurone est représenté par un acteur
- les synapses et les neurones communiquent uniquement par messages
- une impulsion équivaut à un message

## 4.1 Scala

Scala est un langage de programmation multi-paradigme conçu à l'école polytechnique fédérale de Lausanne (EPFL) pour exprimer les modèles de programmation courants dans une forme concise et élégante. Son nom vient de l'anglais *scalable* signifiant « adaptable » ou « qui peut être mis à l'échelle ».

Scala intègre les paradigmes de programmation orientée objet et de programmation fonctionnelle, avec un typage statique. Il concilie ainsi ces deux paradigmes habituellement opposés (à de rares exceptions près, telle que le langage OCaml) et offre au développeur la possibilité de choisir le paradigme le plus approprié à son problème.

Ce langage est prévu pour être compilé en bytecode Java exécutable sur la machine virtuelle Java (JVM). Lorsque le bytecode Scala est utilisé sur la JVM, il est alors possible d'utiliser les bibliothèques écrites en Java de façon complètement transparente. Ainsi, Scala bénéficie de la maturité et de la diversité des bibliothèques qui ont fait la force de Java depuis une dizaine d'années. De plus, il est possible d'invoquer du code écrit en Scala à partir de programmes écrits en Java ce qui facilite la transition entre ces deux langages.

Les développeurs habitués à un seul paradigme (programmation orientée objet avec Java par exemple) peuvent trouver ce langage déroutant et difficile car il nécessite l'apprentissage de concepts différents pour permettre d'exploiter tout son potentiel. Néanmoins, il est tout à fait possible de l'utiliser dans un premier temps comme remplaçant de Java, en profitant alors de sa syntaxe épurée, puis en utilisant les “nouveaux” concepts au fur et à mesure de leur apprentissage.

## 4.2 Akka

Akka est un framework d'acteurs OpenSource pour la JVM maintenant soutenu par l'entreprise TypeSafe. Il permet de gérer efficacement des applications concurrentes et encourage la programmation réactive et événementielle. Son but est donc de simplifier la construction d'applications simultanées et réparties sur la JVM.

Akka supporte plusieurs modèles de programmation pour la concurrence, mais il met l'accent sur la concurrence basée sur les acteurs, inspirée du langage d'Ericsson : Erlang. Avec le côté concurrent du projet, le système des acteurs, ainsi que le langage Scala, Akka est au coeur de notre projet.

## 5 Implémentation de N2S3

Les exemples de code dans cette partie sont allégés pour se focaliser sur les fonctions importantes du programme. Le code complet de ces classes avec la documentation et les commentaires, se trouve dans la section «8 Annexes».

### 5.1 Neurone

#### 5.1.1 Code

Listing 1 – Version «minimale» d’un neurone

```
class Neuron(  
  layer: Int,  
  var parents: Seq[ActorRef],  
  var childs: Seq[ActorRef],  
  threshold: Double) extends Actor {  
  
  def receive = {  
    case PreSynapticSpike(v: Double) =>  
      if (v > threshold) {  
        log.info("[Neuron] PreSynapticSpike (Pass)")  
        childs map { w => w ! PreSynapticSpike(v)}  
      } else {  
        log.info("[Neuron] PreSynapticSpike (Block)")  
      }  
  
    case PostSynapticSpike =>  
      log.info("[Neuron] PostSynapticSpike")  
  
    case ImYourFather(father: ActorRef) =>  
      parents = parents :+ father  
  
    case ImYourChild(child: ActorRef) =>  
      childs = childs :+ child  
  }  
}
```

#### 5.1.2 Structure

Lors de la création du neurone, l’emplacement de sa couche, ainsi que son seuil lui sera indiqué.

Les séquences contenant les synapses parents et enfants seront vides lors de l’instanciation de notre objet, mais remplis au fur et à mesure de la construction des synapses par le biais des messages “ImYourChild” et “ImYourFather”. Quant aux messages “PreSynapticSpike” et “PostSynapticSpike” ils servent d’impulsion que l’on transmet aux synapses.

## 5.2 Synapse

### 5.2.1 Code

Listing 2 – Version «minimale» d’un synapse

```
class Synapse(  
  var weight: Double,  
  parent: ActorRef,  
  child: ActorRef) extends Actor {  
  
  def demo_formula(v: Double): Double = {  
    val oldw = weight  
    weight = weight + (v / 10)  
    if (weight < 0) weight = 0  
    log.info("Weight: " + oldw + " => " + weight)  
    v - 1  
  }  
  
  def receive = {  
    case PreSynapticSpike(v: Double) =>  
      child ! PreSynapticSpike(demo_formula(v))  
  
    case InformNeurons =>  
      child ! ImYourFather(self)  
      parent ! ImYourChild(self)  
      sender ! InformNeurons_ack  
  
    case takeWeight(w :Double) => weight = w + weight  
  }  
}
```

### 5.2.2 Structure

Lors de la création du synapse, on indiquera le neurone parent, enfant ainsi que son poids ; ici le réseau de neurones est linéaire, il y a donc bien un sens de parent à enfant. Lors de la création du synapse, il enverra un message aux neurones attachés pour finaliser la liaison entre eux.

Le message “PreSynapticSpike” est utilisé comme impulsion puis est transmis aux neurones après être passé par la formule du réseau. Cette formule a la possibilité de changer le poids d’un synapse lors de la réception d’un spike. Il est à noter que le message “PostSynapticSpike” existe mais n’est pas encore utilisé dans nos exemples.

## 5.3 Réseau

### 5.3.1 Code

Listing 3 – Version «minimale» de notre réseau de neurones

```
class Network(  
  threshold: Int,  
  nbNeuronPerLayer: Seq[Int],  
  edges: Option[Seq[(Int, Int)]] = None) {  
  
  def createNeurons() : Array[(ActorRef, Int)] = {  
    val neuroRefs = new Array[(ActorRef, Int)](_nbNeuron)  
    var ind = 0  
  
    for (i<-0 until nbNeuronPerLayer.length; j<-0 until nbNeuronPerLayer(i)) {  
      neuroRefs(ind) = (system.actorOf(Props(new Neuron(i, Seq(), Seq(),  
        threshold))), i)  
      ind += 1  
    } return neuroRefs  
  }  
  
  def createFullConnection(neuroRefs: Array[(ActorRef, Int)]) : Seq[ActorRef] = {  
    val rand = new Random()  
    var arefs = Seq.empty[ActorRef]  
  
    for (i<-0 until _nbNeuron; j<-0 until _nbNeuron) {  
      if (i != j && (neuroRefs(j)._2 - neuroRefs(i)._2) == 1) {  
        arefs = arefs :+ system.actorOf(Props(new Synapse(rand.nextDouble()*10,  
          neuroRefs(i)._1, neuroRefs(j)._1)))  
      }  
    } return arefs  
  }  
  
  def createManualConnection(neuroRefs: Array[(ActorRef, Int)], edges: Seq[(Int,  
    Int)]) : Seq[ActorRef] = {  
    val rand = new Random()  
    var arefs = Seq.empty[ActorRef]  
  
    edges map { w =>  
      if ((neuroRefs(w._1)._2 - neuroRefs(w._2)._2) == 1) {  
        arefs = arefs :+ system.actorOf(Props(new Synapse(rand.nextDouble(),  
          neuroRefs(w._1)._1, neuroRefs(w._2)._1)))  
      }  
    } return arefs  
  }  
  
  def reportAndInformNeurons(synapRefs: Seq[ActorRef]) : Unit = {  
    implicit val timeout = Timeout(Duration(60, "seconds"))  
    synapRefs map { w => Await.result(w ? InformNeurons, Duration(300, "millis")) }  
  }  
}
```

### 5.3.2 Structure

Le réseau de Neurones sera composé de neurones, et de synapses. Il faudra indiquer lors de sa création le nombre de couches ainsi que le nombre de neurones par couche, si rien d'autre n'est indiqué le réseau créera les synapses selon une structure prédéfinie :

- Chaque neurone sera relié avec tout les autres neurones de la couche suivante par un synapse. On pourra néanmoins choisir la structure manuellement
- Le système de création du réseau fonctionne par message, il ne laisse donc pas la main à la simulation tant que tout les message n'ont pas été envoyés

## 5.4 Démonstration

### 5.4.1 Code

Listing 4 – Simulation préparée pour tester le fonctionnement de notre simulateur

```
def main(args: Array[String]) = {  
  if (args.length < 2) {  
    usage  
    sys.error("Missing arguments")  
  }  
  
  val nbNeuronPerLayer = Seq(1,2,2,3)  
  val threshold = 8  
  val net = new Network(threshold, nbNeuronPerLayer)  
  net.input(10.0)  
}
```

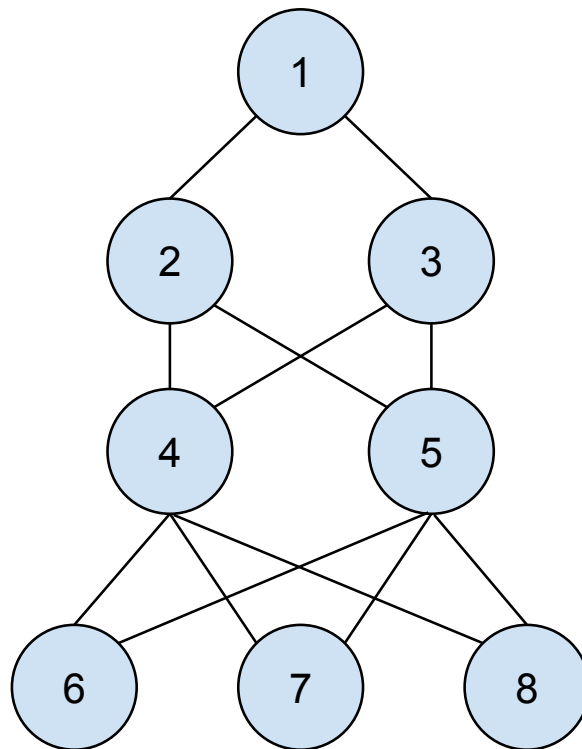


FIGURE 2 – Représente le graphe du réseau lors de notre simulation, les neurones sont les sommets, les synapses sont les arcs entre ces sommets

### 5.4.2 Sortie

Listing 5 – Affichage permettant de suivre l'état de la simulation

```
[info] Running fr.lifl.emeraude.n2s3.Main [14:27:21.416]

[akka:$a] [Neuron] PreSynapticSpike (Pass) [14:27:21.418]
[akka:$i] Weight: 4.354702288089502 => 5.354702288089502 [14:27:21.418]
[akka:$j] Weight: 7.60229729912811 => 8.60229729912811 [14:27:21.418]
[akka:$b] [Neuron] PreSynapticSpike (Pass) [14:27:21.418]
[akka:$c] [Neuron] PreSynapticSpike (Pass) [14:27:21.418]
[akka:$m] Weight: 7.417952446139916 => 8.317952446139916 [14:27:21.418]
[akka:$n] Weight: 6.324756304660703 => 7.224756304660703 [14:27:21.418]
[akka:$l] Weight: 4.977987216513773 => 5.877987216513773 [14:27:21.418]
[akka:$e] [Neuron] PreSynapticSpike (Block) [14:27:21.418]
[akka:$e] [Neuron] PreSynapticSpike (Block) [14:27:21.418]
[akka:$d] [Neuron] PreSynapticSpike (Block) [14:27:21.420]
[akka:$k] Weight: 9.164312997948851 => 10.064312997948852 [14:27:21.420]
[akka:$d] [Neuron] PreSynapticSpike (Block)
```

### 5.4.3 Structure

Lors de cette démonstration un réseau de neurones comprenant quatre couches ayant chacun 1, 2, 2, 3 neurones a été créé. La connexion est dite “complète et automatique” : tous les neurones de couches différentes sont interconnectés.

Le seuil est de 8 (l'unité n'est pas indiqué et utile pour notre simulation, on supposera que l'on parle de Volts), ainsi une impulsion de 10v est envoyée à la première couche.

La formule d'abaissement de la tension dans la synapse effectue une soustraction d'un volt à chaque passage de l'impulsion. L'impulsion parcourt le réseau jusqu'à être bloquée, elle modifie le poids des synapses lors de son passage (voir figure ci-dessus).



## 6 Conclusion

### 6.1 Difficultés

La principale difficulté du projet a été l'apprentissage et la compréhension du fonctionnement d'un réseau de neurones. Plusieurs semaines de travail avec l'équipe ont été nécessaires pour permettre une base solide dans le développement de notre application.

Pour Quentin, une difficulté supplémentaire a été l'utilisation de Scala ; ce nouveau langage est en effet atypique et encourage l'utilisateur à utiliser des constructions du paradigme fonctionnel. Il est donc courant de voir l'utilisation de "functor" (fonction d'ordre supérieur) comme par exemple map, filter, fold dans des projets Scala. De plus, le coeur de notre système utilise un système d'acteurs qui nécessite de penser de manière complètement asynchrone et décentralisée.

### 6.2 Bilan

Ce projet n'était initialement pas prévu dans la liste des sujets PJI mais après plusieurs recherches de thèmes, nous avons trouvé avec M. Boulet un sujet qui nous correspond. Nous avons connu un départ difficile, en particulier la construction de l'état de l'art des domaines suivants : "Discret Event Simulation" et "Spiking Neuron Simulator". Dans un premier temps, nous avons proposé la structure de notre simulateur, puis après validation auprès de notre équipe de recherche, nous avons pu programmer une première version fonctionnelle.

L'objectif principal est donc atteint, le réseaux de neurones est fonctionnel. Cependant, certains éléments peuvent être améliorés, nous sommes donc en train de programmer la version 2 qui sera disponible pour le début du mois de juin.

Nous regrettons le manque de temps disponible pour effectuer notre PJI. Nos différentes unités d'enseignement sont en effet très prenantes et nous ont obligées à attribuer moins de temps que ce que nous voulions pour notre projet.

### 6.3 Perspectives d'améliorations

Nous pouvons améliorer notre simulateur avec les fonctionnalités suivantes :

- restructuration du code (meilleure généricité) pour extraire le modèle du simulateur
- ajout d'un modèle de temps dans notre simulation
- ajout d'une liste d'événements récents dans les neurones et synapses pour la gestion de l'apprentissage et de l'oubli
- prise en charge du format de données .aer
- ajout d'un langage dédié (DSL) pour la création automatique d'un réseau
- ajout d'un monitoring dans le réseau pour visualiser chaque neurone ou synapse

Nous avons mis l'équivalent d'un "sleep" avant de lancer une simulation, pour permettre à notre simulateur de créer les connexions entre neurones et synapses. Cette méthode n'est pas satisfaisante, nous avons donc commencé les améliorations en transformant l'instruction sleep sous la forme de "Future" qui permettent un contrôle plus fin et plus propre.

Le développement de notre simulateur continuera durant le mois de juin avec les améliorations décrites ci-dessus. Une réunion organisée avec d'autres équipes de recherche se tiendra mi-juin pour discuter de leurs besoins dans ce domaine et potentiellement modifier notre simulateur pour intégrer leurs suggestions. Le simulateur N2S3 sera probablement open source, ceci nous permettrait de continuer à développer ce projet.

## 7 Remerciements

Nous souhaitons remercier M. Boulet pour son aide précieuse en Scala, Akka et bien sûr la proposition de ce sujet. Nous souhaitons également remercier M. Devienne et M. Shahsavari pour leur aide et leurs conseils sur le fonctionnement des neurones, synapses et de manière plus générale d'un simulateur de réseaux de neurones.

## 8 Annexes

### 8.1 Code d'un neurone

```
package fr.lifl.emeraude.n2s3.actors

import akka.actor._
import akka.event.Logging
import fr.lifl.emeraude.n2s3.actors.messages._

/**
 * Neurons are the "brain" of the simulator, they will compute and send spikes.
 *
 * @constructor create a new Neuron in a form of an actor with a layer, its parents,
 * its childs and a threshold.
 * @param layer in which the neuron is
 * @param parents sequence of synapses' references, for all "parent-synapse".
 * @param childs sequence of synapses' references, for all "child-synapse".
 * @param threshold the trigger value for the neurons.
 *
 * @author wgouzer & qbailleul
 */
class Neuron(
  layer: Int,
  var parents: Seq[ActorRef], // actor ref of synapse
  var childs: Seq[ActorRef], // actor ref of synapse
  threshold: Double) extends Actor {

  val log = Logging(context.system, this)

  def setParents(p: Seq[ActorRef]) {
    parents = p
  }

  def setChilds(c: Seq[ActorRef]) {
    childs = c
  }

  def receive = {
    case PreSynapticSpike(v: Double) => {

      if (v > threshold) {
        log.info("[Neuron] PreSynapticSpike (Pass)")
        childs map { w => w ! PreSynapticSpike(v)}
      } else {
        log.info("[Neuron] PreSynapticSpike (Block)")
      }
    }

    case PostSynapticSpike =>
      log.info("[Neuron] PostSynapticSpike")
  }
}
```

```
case ImYourFather(father: ActorRef) =>
  parents = parents :+ father

case ImYourChild(child: ActorRef) =>
  childs = childs :+ child

case Welcome =>
  childs foreach { w => println(w.toString) }

case _ => throw new Exception(
  "Neuron receive a unknown message, don't know what to do"
)
}
```

## 8.2 Code d'un synapse

```
package fr.lifl.emeraude.n2s3.actors

import akka.actor._
import akka.event.Logging
import fr.lifl.emeraude.n2s3.actors.messages._

/**
 * Synapses are the link between two Neurons.
 *
 * @constructor create a new Synapse in a form of an actor with a weight, a parent
 * and a child
 * @param weight mechanism for reinforcing or deprecating a link (synapse)
 * @param parent the "parent-neuron" connected with this synapse
 * @param child the "child-neuron" connected with this synapse
 *
 * @author wgouzer & qbailleul
 */
class Synapse(
  var weight: Double, // next version will no longer need this variable

  parent: ActorRef,
  child: ActorRef) extends Actor {

  val log = Logging(context.system, this)

  def compute_spike() {
    // given by mahyar
  }

  /**
   * a formula to test the network
   */

  def demo_formula(v: Double): Double = {
    val oldw = weight
    weight = weight + (v / 10)
    if (weight < 0) weight = 0
    log.info("Weight: " + oldw + " => " + weight)
    v - 1
  }

  def receive = {
    case PreSynapticSpike(v: Double) =>
      child ! PreSynapticSpike(demo_formula(v))

    case PostSynapticSpike => // not used yet

    case InformNeurons =>
      child ! ImYourFather(self)
      parent ! ImYourChild(self)
  }
}
```

```
    sender ! InformNeurons_ack

    case Welcome => // debug purposes

    case takeWeight(w :Double) => weight = w + weight

    case _ => throw new Exception(
        "Synapse receive a unknown message, don't know what to do"
    )
}
}
```

## 8.3 Code du réseau

```
package fr.lifl.emeraude.n2s3

import akka.actor._
import akka.routing._
import akka.pattern.ask
import akka.util.Timeout

import scala.math
import scala.util.Random
import scala.concurrent.duration._
import scala.concurrent._

import fr.lifl.emeraude.n2s3.actors.{ Neuron, Synapse }
import fr.lifl.emeraude.n2s3.actors.messages.{ InformNeurons, PreSynapticSpike,
  Welcome }

/**
 * Create the neuronal network with a threshold, the number of neuron per layer
 * <br>
 * and the edges (synapses) between neurons
 *
 * @constructor Network in a form of actors inside a tree/graph.
 * @param threshold is the neurons's trigger value for propagating spikes.
 * @param nbNeuronPerLayer is the number of neurons in a layer. Conveniently,
 * the number of elements also represents the number of layer. <br>
 * The following example will produce 3 layers, the first and second will have 4
 * neurons and the last one 2 neurons :
 * {{{
 *   Seq(4,4,2)
 * }}} <br>
 * @param edges are the synapses in the network, which is a Seq of a couple of
 * Integers,
 * like this (FromNeuron1, ToNeuron2). <br>
 * The following example will produce three oriented edges 1->2, 1->3 and 2->3 :
 * {{{
 *   Seq((1,2), (1,3), (2,3))
 * }}}
 * @author wgouzer & qbailleul
 */
class Network(
  threshold: Int,
  nbNeuronPerLayer: Seq[Int],
  edges: Option[Seq[(Int, Int)]] = None) {

  /**
   * Total number of neurons in this network, it's global for avoiding the
   * re-computing
   * everytime a function needs it (at least 2-3 times in this code).
   */
  lazy val _nbNeuron = nbNeuronPerLayer.sum
}
```



```

/**
 * Creates all the needed neurons.
 * @return an array of couple : neurons' references and layer.
 */
def createNeurons() : Array[(ActorRef, Int)] = {
  val neuroRefs = new Array[(ActorRef, Int)](_nbNeuron)
  var ind = 0

  // create layer with x neurons into each layer
  for (
    i <- 0 until nbNeuronPerLayer.length;
    j <- 0 until nbNeuronPerLayer(i)
  ) {
    neuroRefs(ind) = (system.actorOf(Props(
      new Neuron(i, Seq(), Seq(), threshold))), i)
    ind += 1
  }

  neuroRefs
}

/**
 * Create the links (synapses) between neurons.
 * @param neuroRefs the array that contains the couples of neurons' references
 *   and their layer.
 * @return the references of all the synapses created.
 */
def createSynapses(neuroRefs: Array[(ActorRef, Int)]) : Seq[ActorRef] = {
  // # maximum of edges in a graph => n * (n-1) / 2
  edges match {
    case None => createFullConnection(neuroRefs)
    case Some(t) => createManualConnection(neuroRefs, t)
  }
}

/**
 * Connects each neuron on a higher layer to
 * every neuron in its inferior layer.
 * @param neuroRefs the array that contains the couples of neurons' references
 *   and their layer.
 * @return the references of all the synapses created.
 */
def createFullConnection(neuroRefs: Array[(ActorRef, Int)]) : Seq[ActorRef] = {
  val rand = new Random()
  var arefs = Seq.empty[ActorRef]

  for (
    i<-0 until _nbNeuron;
    j<-0 until _nbNeuron
  ) {

    if (i != j && (neuroRefs(j)._2 - neuroRefs(i)._2) == 1) {

```

```

        arefs = arefs :+ system.actorOf(Props(
            new Synapse(rand.nextDouble()*10, neuroRefs(i)._1, neuroRefs(j)._1)))
    }
}

arefs
}

/**
 * Connects manually the neurons with synapses. It is done through the
 * constructor.
 * param edges in [[fr.lifl.emeraude.n2s3.Network]].
 * @param neuroRefs the array that contains all the couples of neurons'
 *   references and their layer.
 * @param edges represents the connections between two neurons (synapses).
 * @return the references of all the synapses created.
 */
def createManualConnection(neuroRefs: Array[(ActorRef, Int)], edges: Seq[(Int,
    Int)]) : Seq[ActorRef] = {
    val rand = new Random()
    var arefs = Seq.empty[ActorRef]

    edges map { // w is a couple of neurons' references
        w =>
            // checks if the layers are connected with +/- 1 of difference
            if ((neuroRefs(w._1)._2 - neuroRefs(w._2)._2) == 1) {
                arefs = arefs :+ system.actorOf(Props(
                    new Synapse(rand.nextDouble(), neuroRefs(w._1)._1, neuroRefs(w._2)._1)))
            }
    }

    arefs
}

/**
 * Makes all the synapses send messages to their neurons
 * in order to inform them about the topology of the network.
 * @param synapRefs array that contains all the synapses' references.
 */
def reportAndInformNeurons(synapRefs: Seq[ActorRef]) : Unit = {
    implicit val timeout = Timeout(Duration(60, "seconds"))

    synapRefs map { w =>
        Await.result(w ? InformNeurons, Duration(300, "millis"))
    }
}

/** create a new actor system with the specific name "Neuronal-network" */

implicit val system = ActorSystem("Neuronal-network")

```

```

/** create an Array of couple for all the neurons' references and their
    associated layers */
val neuronsActorRef = createNeurons()

/** create a Sequence of all the synapses' references */
val synapsesActorRef = createSynapses(neuronsActorRef)

/** tells the neurons about their new friends : synapses (edges of the graph) */
reportAndInformNeurons(synapsesActorRef)

/**
 * Send a message to the first Neuron in the first layer
 * @param v spike (that's the message)
 */
def input(v: Double) {
  for ( i<-0 until _nbNeuron) {
    if (neuronsActorRef(i)._2 == 0) { neuronsActorRef(i)._1 ! PreSynapticSpike(v)}
  }
}
}

```

## 8.4 Messages envoyé par le réseau

```
package fr.lifl.emeraude.n2s3.actors

import akka.actor._

/**
 * Represents the messages that Neurons and Synapses will send to each other.
 * @author wgouzer & qbailleul
 */
object messages { // abstract class "Spike"

  /**
   * Typical hello world type message, it isn't useful for real world application,
   * it's here for debug and test purposes
   */
  case class Welcome

  /**
   * Used for discovering the network, it'll allow the synapses to send messages
   * to the neurons they're connected to in order to let him update its
   * sequences (parents & childs).
   */
  case class InformNeurons

  /**
   * Used for discovering the network, it'll allow the neurons to acknowledge
   * the reception of the message (from the synapses)
   */
  case class InformNeurons_ack

  /**
   * Tell the neuron connected to this synapse : "this synapse is your father"
   * @param neuron for which the synapse is the father
   */
  case class ImYourFather (neuron: ActorRef) // tell the neuron you're its father

  /**
   * Tell the neuron connected to this synapse : "this synapse is your child"
   * @param neuron for which the synapse is the child
   */
  case class ImYourChild (neuron: ActorRef) // tell the neuron you're its child

  /**
   * Simulate a pre-synaptic spike, a simple spike before all the computing
   * done on it
   */
  case class PreSynapticSpike (v: Double) // spikeF (forward)

  /**
   * Simulate a post-synaptic spike, a simple spike altered by the computing
   * done on it
   */
}
```

```
    **/  
case class PostSynapticSpike // spikeB (backward)  
  
/**  
  Those two messages (addMessage, takeWeight) are present just for "demoing"  
  the simulation to people  
  */  
case class addMessage(immune :ActorRef)  
case class takeWeight(w: Double)  
}
```

## 8.5 Main du programme

```
package fr.lifl.emeraude.n2s3

/**
 * Instanciate the neuronal network with akka's actors.
 * @author wgouzer & qbailleul
 */
object Main {

  def usage() {
    println("\nUsage: [VM] [PROGRAM] [OPTION]")
    println("\t e.g. java -jar n2s3.jar")
    println("\nAvailable options :")
    println("\t--debug")
    println("\t--verbose\n")
  }

  def main(args: Array[String]) = {
    if (args.length < 2) {
      usage
      sys.error("Missing arguments")
    }

    // instanciate the graph
    val nbNeuronPerLayer = Seq(1,2,2,3) // args(0)
    val threshold = 8 // args(1)
    val net = new Network(threshold, nbNeuronPerLayer)
    net.input(10.0)

  }
}
```