# Table des matières

# Première partie

# Introduction

Ce rapport s'inscrit dans une étude des champs neuronaux dynamiques. Dans ce papier il sera question d'étudier plusieurs routages, c'est à dire les manières de faire communiquer les neurones entre eux. On va ,la plupart du temps, étudier le routage d'un graphe ou d'une superposition de graphe de taille 9x9. En général un nœud du graphe fera office de neurone et une arrête entre deux nœuds, une synapse.

Jusqu'alors nous travaillons sur le modèle CNFT (ref) . Rappelons tout d'abord en quoi consiste ce modèle. Nous avons un réseau de neurones. Chaque neurone est connecté avec l'ensemble des autres neurones. Chaque neurone dispose d'un seuil d'activation. Après avoir reçu une charge suffisamment importante en tension, le neurone s'active et envoie un spike (une décharger en tension) aux neurones afférents. Cet envoie suit un routage bien particulier : l'idée est d'envoyer des tensions plus faibles aux neurones les plus éloignés en suivant une courbe type différence de deux gaussiennes. Le problème de ce modèle est qu'il requière une interconnexion de neurones globale. Chaque neurone est lié aux autres neurones. Du fait du nombre de connexions,il sera difficile d'implémenter ce routage sur une carte FPGA par exemple. Or c'est notre objectif ici. Un tel modèle pourrait permettre à un robot de faire de la reconnaissance de forme et du suivi de cible par exemple.

Afin d'alléger le modèle CNFT dans son ensemble, on va s'intéresser à des type de routage qui ne nécessite pas autant de connexions neurales. Ce rapport a donc pour objectif d'exposer les différents outils mis à disposition afin de tester différents routages dans le cadre de ce modèle. On va voir comment générer des données relatifs à ces routages et comment les analyser. La mise en pratique sera faite avec le routage "Probabilistic flooding" (ref). Ce routage connecte les neurones seulement voisins de manière 4-connexe. Avant l'envoie d'un spike, le neurone emmeteur fait un test de Bernouilli avec un poids particulier (qu'on appellera weigth ou poids par la suite) commun à tous les neurones du modèle, si le test est raté, alors il envoie le spike sinon il le détruit.

Ce rapport résume le travail que réalisé au cour des deux mois d'été ,juillet-août 2014, dans l'équipe Cortex sous la direction de Benoit Chapellet de Vangel et Bernard Girau. Il a été question de développer un framework de simulation pour différents routages puis un framework de test de ces routages. Le framework de simulation est écrit en Java et se greffe au logiciel DNFSim de Benoit Chapellet de Vangel. Il permet de créer visualiser et analyser un réseau. La conception du framework est telle qu'il est assez générique pour créer tout type de graphe. Dans un second temps, nous nous sommes concentré sur un framework de test pour ces réseaux. Ce dernier permet de générer un jeu de donnée produit par DNFSim,d'analyser la répartition des paquets sur ce réseau et de vérifier l'émergence de certaines propriétés du réseau. Afin de valider ce travail et par la même occasion émettre une piste de nouveau routage, on a prit pour exemple le routage "Probabilistic flooding" qu'on a par la suite intégrer au modèle CNFT.

# Deuxième partie

# Network

Dans cette partie, nous allons présenter le framework de simulation de réseaux ainsi que son application sur plusieurs exemples.

## 0.1 Objectif

Notre but est de pouvoir concevoir des réseaux de manières génériques et assez intuitive. Le programme se doit d'être évolutif. On s'oriente vers la creation d'un framework en java afin de le greffer sur le logiciel DNFSim déjà existant.

## 0.2 Genericité

Dans l'optique de représenter différents routage, le framework se doit d'être générique. On doit être capable de créer un graphe avec des nœuds et des arêtes avec un comportement particulier. Les informations circulant sur le réseau (paquet) sont eux aussi déclinables sous plusieurs formes. Très succinctement, un réseau est représenté par un **SpreadingGraph.java** . Ce réseau possède des nœuds **Node.java** ainsi que des arêtes qui lient ces nœuds. Sur ce réseau circulent des informations sous forme de paquet **Packet.java** . Toutes ces classes sont paramétrables et déclinables sous des classes filles. Voir la javadoc pour plus d'informations.

## 0.3 Le routage probabilistic flooding

Le routage probabilistic flooding se base sur la notion de probabilistic flooding à la spécificité près que les nœuds sont liés de manières 4 connexes i.e chaque nœud est lié à son voisin nord, sud, est et ouest. On associe à tout les noeuds un poids commun. Ce poids est utilisé à chaque computation du graphe. Lors d'une computation, le nœud choisit un paquet qu'il possède en file d'attente (FIFO). Il lance alors une expérience de Bernouilli paramétré par le poids donné. Si il réussit alors il détruit le paquet, si il rate il le laisse passer. On retrouve toutes les classes relatives à ce routage dans le package main.java.network.probabilisticFlooding

## 0.4 Le modèle probabilistic flooding spike

Afin de tester l'efficacité du routage probabilistic flooding sur le modèle CNFT, nous avons crée le modèle probabilistic flooding spike (PFS). Rappelons que le modèle CNFT représente une interconnexion de neurones qui envoient deux types de signaux : inhibiteur et excitateur. C'est la différence des deux qui définit quel voltage le neurone reçoit. La propagation de chaque type de signal ressemble à une courbe gaussienne. Distinguons maintenant les deux couches dans ce modèle. Les nœuds qui envoient des signaux excitateurs et les nœuds qui envoient des signaux inhibiteurs. Ces deux couches suivent donc un routage bien

particulié. C'est ce routage que l'on va remplacer par le routage probabilistic flooding. Ces deux routages sont paramétrés suite aux tests qu'on a pu faire (en partie grâce au framework de test) afin de copier la répartition gaussienne des routages initiaux.

# Troisième partie

# Stimulus

Après la création d'un réseau grâce au framework présenté précédemment, il est possible de paramétrer un certain nombre d'événement sur ce réseau. Ça consistera à envoyer des paquets sur le graphes pendant ses computations.

## 0.5 Objectif

On peut exciter chacun des nœuds du graphe à différents moments. L'objectif est de voir le comportement du graphe (et du routage associé) selon différents événements. Ces événements sont caractérisés par l'envoie de stimulus, i.e l'ajout de paquet à la file d'attente d'un ou plusieurs nœuds à un ou plusieurs moments de la simulation.

## 0.6 Utilisation

Le framework pour la programmation d'envoi de stimulus se base sur le format xml. Tous les fichiers de programmation doivent se trouver sous le dossier statistiques/stimulis avec l'extension .stimulis (et non .xml). Cette extension est nécessaire aux fichiers de scripts notamment.

Commençons par un exemple. Admettons que l'on veuille envoyer un paquet de type spike au nœud zéro du graphe à l'instant t=0 :

asend.stimulis

```
<?xml version="1.0" encoding="UTF-8"?>
<programmation class="main.java.network.generic.packet.Spike">
        <time t="0.0">
                <add indice="0"/>
        </time>
</programmation>
```

On retrouve ici trois balise :

- <programmation> : On stipule qu'on va commencer un programme d'envoi stimulis. L'attribut "class" est le nom de la class Java des paquets que l'on veut envoyer en tant que stimulis.
- <time> : Pour spécifier les envois à un temps t donné. A noté que

$$dtStimulis = k * dtGraphe \ \forall k \epsilon \mathbb{N}^*$$

  i.e on ne peut pas envoyer des stimulis entre deux computations.
- <add> : On envoie un paquet à l'indice indice. Notez que les indices sont en représentation 1D row-major du graphe.

Un exemple d'envoi de paquets ipv4 sur un réseau :

ipv4.stimulis

```
<?xml version="1.0" encoding="UTF-8"?>
<programation class="main.java.network.generic.packet.IPv4Datagramme">
```

```
        <time t="0.0">
                <add indice="0">
                        <params> A </params>
                        <params> yes </params>
                        <params> ca fonctionne ! </params>
                </add>
        </time>
        <time t="0.1">
                <add indice="80">
                        <params> B </params>
                        <params> yes </params>
                        <params> ca fonctionne ! </params>
                </add>
        </time>
</programation>
```

On découvre la balise <params>. L'utilisation de cette balise est directement liée à la structure du constructeur de la class spécifiée dans la balise programation. Dans l'exemple précédent on utilisait des spikes, objet simple ne prenant aucun paramètre lors de la construction, on a pas eu besoin d'utiliser la balise <params>. Par contre un paquet ipv4 transporte un message. Ce message est "A". Les valeurs "yes" et "ça fonctionne !" n'ont aucun effet sur la construction d'un objet ipv4 vu qu'il ne prend qu'un string en params.

Il faut regarder de plus près le code java.

Packet.java

```java
public class Packet {

    private int size;

    protected Packet(Object ... params){
    }

    //...
```

Chaque classe fille de Packet doit implémenter le constructeur protected.(ça doit être le premier constructeur dans le code pour des raisons d'introspection, même si il y a moyen de faire mieux).

Spike.java

```java
public final class Spike extends Packet {

    public Spike(Object ... params) {
        setSize(1);
    }
```

```
6
7   }
```

On voit que Spike n'utilise aucun de ses paramètres.

IPv4Datagramme.java

```
1   public final class IPv4Datagramme extends Packet{
2
3           private String message;
4
5           public static final int MESSAGE_INDEX = 0;
6
7           public IPv4Datagramme(Object ... params) {
8           setSize(16);
9           setMessage((String)params[MESSAGE_INDEX]);
10          }
11
12          //...
```

IPv4 utilise un seul paramètre. A noté que les objets récupérés seront de type String.

# Quatrième partie

# Génération de données

## 0.7 Introduction

Afin de tester certaines propriétés du graphe, il est nécessaire de générer un jeu de donnée conséquent concernant différents caractéristiques du routage.

## 0.8 Spikes reçus jusqu'alors

Dans cette section, nous allons nous intéressé à la génération du jeu de donnée représentant la quantité de spike reçu par chaque nœud du graphe, de l'instant 0 à l'instant t. Pour ce faire, il faut utiliser le script generateData.py. Ce script s'occupe de lancer cette expérience :

*Alice travaille sur le PFModel (probabilistic flooding model), elle envoie des stimulis sur le graphe associé à un ou plusieurs moments de la propagation. Le graphe a une taille et un poids (probabilité de détruire un spike sortant d'un nœud). Alice laisse tourner la propagation pendant t secondes.Elle récupère la matrice représentant le nombre total de spikes reçus par chaque nœuds jusqu'à l'instant t*

On lance cette expérience plusieurs fois (itérations) et selon différents paramètres (taille, poids, programmation des stimulis).

Un exemple d'appel de cette commande :

```
python generateData.py --weigths 0.0 0.1 0.2 0.3 0.4 --times 1 2
--packet_initialisation twoa twob twoab --iterations 100
--tailles_grilles 9 --forcerewrite
```

Ici on génère les données pour les combinaisons de poids temps. On stipule que ce sera de taille 9 mais on aurait pu écrire

```
--taille_grilles 1 2 3 4 5 6 7
```

On veut que l'expérience soit réalisée selon plusieurs programmations (twoa twob et twoab). Les dites programmation doivent se trouver dans le dossier statistiques/stimulis sous la forme nom.stimulis (exemple : twoab.stimulis).

Les données générées se trouvent sous le dossier statistiques/data (avec une arborescence dépendante des paramètres).

A noter que ce script n'est utilisable qu'avec le PFModel pour le moment mais il est facilement extensible à d'autre modèles.

Faire

```
python generateData.py --help
```

pour plus d'informations

# Cinquième partie

# Additivité

Il est nécessaire de conserver la propriété d'additivité du réseau lors de l'utilisation de nouveaux routages. Pour cela, posons une définition théorique de l'additivité.

## 0.9 Théorie

A partir de maintenant, on considère un graphe G(N,E) avec N l'ensemble de ses nœuds et E l'ensemble de ses arêtes. Chaque nœud possède une file d'attente FIFO. On peut lancer une propagation sur ce graphe.

**Definition 1.** *(Paquet) (PAS ASSEZ RIGOUREUX)*
*Un* ***paquet*** $\mathtt{p}$ *est un objet. Il contient un certain nombre d'informations. Il se déplace sur le graphe de nœuds en nœuds au fil du temps (lors de la propagation du graphe). Si le* ***paquet*** *ne contient aucune information, on parle de* ***spike****.*

**Definition 2.** *(Stimulis)*
*Un* ***stimulis*** $\mathtt{S}_{p,n,t}$ *est le placement du paquet* $\mathtt{p}$ *dans la file d'attente du nœud n à l'instant t. Si t=0 et si p est un spike, on parle alors d'* ***activation****. Une* ***activation*** $\mathtt{A}_n$ *est le* ***stimulis*** $\mathtt{S}_{s,n,0}$

**Definition 3.** *(Configuration)*
*Une* ***configuration*** $\mathtt{C}$ *est un ensemble de stimulis. On parle d'initialisation* $\mathtt{I}$ *si c'est un ensemble d'activation.*

**Definition 4.** *(Additivité)*
*Soit* $\mathtt{A}^0, \mathtt{A}^1, .., \mathtt{A}^K$ *des activations où* $\mathtt{A}^i = \mathtt{A}_{n_i} \forall i$.
*On pose l'initialisation* $\mathtt{I} = \{\mathtt{A}^k \mid n \epsilon [0, K] \text{ et } K > 0\}$
*On pose* $\mathtt{I}_k = \{\mathtt{A}^k\}\ \forall n \epsilon [0, K]$

*Soit* $M_{t,n,\mathtt{I}}$ *la variable aléatoire qui représente l'ensemble des paquets reçus par le nœud n depuis le début de la propagation pour chaque nœud du graphe à l'instant t selon l'initialisation* $\mathtt{I}$.

*On pose* $M_{t,\mathtt{I}} = (M_{t,0,\mathtt{I}}, M_{t,1,\mathtt{I}}, ..., M_{t,card(N),\mathtt{I}})$ *et* $M_{t,\mathtt{I}_i} = (M_{t,0,\mathtt{I}_i}, M_{t,1,\mathtt{I}_i}, ..., M_{t,card(N),\mathtt{I}_i})$

*On dit que le routage est* ***additif pour l'initialisation*** $\mathtt{I}$ *si* $\forall t$

$$\mathbb{E}(M_{t,\mathtt{I}}) = \sum_{i=1}^{n} (\mathbb{E}(M_{t,\mathtt{I}_i}))$$

*On dit qu'un routage est* ***additif*** *si il l'est pour n'importe quelle initialisation.*

*Note : On pourrait parler d'additivité pour une configuration qui n'utilise que des spikes, et pas forcement une initialisation.*

## 0.10 Utilisation du script

Afin de tester cette propriété d'additivité sur des résultats expérimentaux, nous avons crée un script en Scilab. Ce dernier lit un ensemble de donnée de plusieurs répétition d'expériences afin d'extraire une moyenne de ces données, puis il va calculer à quel point le routage est additif sur ces résultats moyens.

Par exemple

```
scilab -nw -f Additivite.sce -args '9' '1 2'
'0.0 0.1 0.2 0.3 0.4' '5 20 50 75 100'
'twoa twob' twoab 0
```

vérifie l'additivité twoa+twob=twoab sur les combinaisons possibles entre

- tailles : 9
- times : 1 2 (secondes)
- poids : 0.0 0.1 0.2 0.3 0.4
- tailles échantillons : 5 20 50 75 100 (nombre de répétition de l'expérience)

## 0.11 Résultats

On lance le script avec taille 9 dt 0.1 et time 5s. Sur l'image suivante on remarque qu'a partir d'un certain temps de computation, on a une additivité maximale autour de 0.7. On remarque aussi que l'activité fluctue bien moins quand on augmente le nombre d'expériences. Cette dernière observation pourrait nous amener à la conclusion qu'il faille lancer plusieurs spikes en même temps par activation de neurone (on simulerait en quelque sorte la répétition d'expérience même si on a aucune garantie que le comportement soit similaire. C'est à tester).

{TODO pourquoi c'est meilleurs en 0.7 et pas en 0.0 ?}

En zoomant autour de 0.7 on trouve une valeur minimale de finesse en 0.72. A noter que cette valeur n'est valable que pour cette configuration (taille 9 time 5s dt 0.1)

# Sixième partie

# Propagation des spikes

Pour qu'un nouveau model soit viable, en plus d'avoir un comportement additif, il doit avec une répartition voltaïque de type "différence de deux gaussienne". Dans cette partie on va voir comment vérifier cette propriété et si notre modèle "Probabilistic flooding" permet d'obtenir la répartition désirée.

## 0.12 Utilisation du script

Le script MeanAndVariance.sce permet d'obtenir les statistiques moyennes variances et écart type sur un échantillon d'expérience et ce en utilisant différents paramètres. Un exemple de cette commande :

```
scilab -nw -f MeanAndVariance.sce -args
9 '1 3 5' '0.7' 1000 a_send_20 temp/
```

avec respectivement comme arguments :
— 9 : la taille du réseau (la racine)
— '1 3 5' : les temps de computations
— '0.3 0.7' : les poids
— 1000 :le nombre d'expérience qu'on utilise pour calculer nos statistiques
— a_send_20 : le scénario des expérience (a _send _20.stimulis doit de trouver sous le dossier stimulis/)
— temp/ : le dossier sous lequel les images vont être enregistrer
Cette commande va générer sous le dossier temp un ensemble de graphique représentant les 3 statistiques sur l'ensemble du graphe et sur la diagonale.

## 0.13 Résultats

On voit sur l'image ci contre que c'est avec un poids de 0.7 et un temps de computation de 3s qu'on obtient une courbe proche de ce que l'on cherche.
Le problème c'est que l'écart type est élevé comme on peut le voir sur cette image.

## 0.14 Réflexions

Afin de voir si ce modèle est fiable et pas trop aléatoire on se penche sur le calcul d'un intervalle de confiance autour de la moyenne. Le seul problème c'est que la distribution ne suit pas une loi de probabilité connu,comme on peut le voir sur cette image, du coup on pourrait s'orienter vers des méthodes bootstrap.

## Septième partie

# Probabilistic flooding et I/E

On rappelle que l'objectif est de trouver un modèle de réseau neural simulant le comportement de DNF tout en consommant moins de ressource. Pour cela on va utiliser le routage probabilitic flooding. Ce routage comme vu précédemment s'avère additif et suit une distribution exponentielle/gaussienne selon certains paramètre. On a donc le candidat idéal pour simuler le routage originel de DNF. On va donc mettre en place de couche de graphes utilisant le routage PF (probabilitic flooding). Une couche avec des signaux excitateurs et une autre avec des signaux inhibiteurs. On applique une transformation affine a ce deux couches puis on soustrait l'une et l'autre. On obtient à la fin la distribution du potentiel des neurones en fonction de leur éloignement aux stimulis. On veut que cette distribution ressemble à la différence de gaussienne du routage de DNF. Pour ça on doit trouver les paramètres des transformations affines ainsi que les poids et les temps de computations des deux graphes. C'est le script InhibiteurExcitateur.sce qui nous permet d'observer l'effet de tel ou tel paramètre sur le comportement voltaïque du réseau de neurones.

## 0.15 Utilisation du script

Le script InhibiteurExcitateur.sce permet d'observer la répartition voltaïque sur la diagonale d'un réseau de neurones. Comme expliquer précedement, on superpose deux couches de graphes suivant le routage probabilistic flooding puis on fait leur applique à chacun une transformation affine. Ce script va afficher la fonction

$$a_e * E(w_e, t_e) + b_e - (a_i * I(w_i, t_i) + b_i)$$

L'interface du script :

```
scilab -nw -f InhibiteurExcitateur.sce -args
taille configuration repetition wi ti ai bi we te ae be
```

avec respectivement comme arguments :
- taille : la taille du réseau (la racine)
- configuration : la configuration (asend, bsend ...)
- repetition : le nombre d'expérience qu'on utilise pour calculer nos statistiques
- wi : le poids pour le graphe d'inhibition
- ti : le temps de computation pour le graphe d'inhibition
- ai : le coefficient directeur de la fonction affine du graphe inhibiteur
- bi : la constante de la fonction affine du graphe inhibiteur
- we : le poids pour le graphe d'excitation
- te : le temps de computation pour le graphe d'excitation
- ae : le coefficient directeur de la fonction affine du graphe d'excitation
- be : la constante de la fonction affine du graphe d'excitation

## 0.16 Résultats

On veut obtenir une courbe de cette forme.On lance :

```
scilab -nw -f InhibiteurExcitateur.sce -args
9 a_send 1000 0.5 3 0.0428571 0.0 0.7 5 0.001 -0.5
```

Et on récupère une courbe de cette forme. La correspondance nous a semblé suffisante pour des tests en conditions réelles.

## 0.17 Application sur dnfsim

On va donc utiliser cette combinaison linéaire de Probabilistic flooding graphes afin de simuler DNF. Pour cela, on incorpore les différents graphes au logiciel dnfsim. On run un scénario avec deux bulles à des positions opposées qui tournent. Ça fonctionne très mal, la variance est très élevée, on a rarement le focus d'une bulle. Pour palier à ce problème de variance on s'oriente sur l'envoi de {20,40,60,80} spikes par activation de neurone pour voir ce que ça donne. On part de l'idée qu'envoyer plusieurs spikes en même temps permettrait la diminution de variance, en effet c'est comme si on répétait l'expérience plusieurs fois. Afin de voir si notre hypothèse est plausible, on va faire quelques simulations dans cette configuration.

## 0.18 Variance pour {20,40,60,80}_asend

On va supposer que notre scénario de départ permet l'envoi de {20,40,60,80} en A (en haut à gauche du réseau). Puis on va observer la répartition des spikes sur la diagonale du réseau (de A à B - en bas à droite - ). On en déduit une variance ainsi qu'un intervalle de confiance pour cette variance.

### 0.18.1 Utilisation du script

Il suffit de run

```
scilab -nw -f MultiBootstrap.sce
```

en ligne de commande. Ce script affiche la variance selon les configurations 20/40/60/80 ainsi que l'intervalle bootstrap à 95

### 0.18.2 Résultats

Pour commencer sous la contrainte de 5s de computation, on ne voit pas de grosses différences en terme de variance entre les 4 scénarios (20 40 60 80) comme on peut le constater sur cette image. Pour voir vraiment une différence, on voudrait augmenter le temps de computation. Le problème c'est que l'on

change totalement les paramètres de départ, du coup on pourrait perdre des propriétés. En l'occurrence, on perd la forme désirée de la courbe (une demi gaussienne). Afin de trouver un moyen de diminuer la variance du modèle, il faut définir quelques contraintes. Plus précisément on doit garder la propriété d'additivité ainsi que la même forme de courbe.

{TODO trouver un set de paramètre qui permettent de diminuer la variance.Si j'arrive pas avec paramètre jouer sur l'activation des neurones. Pourquoi on a ce comportement là (très varié) comment on peut l'améliorer ? Y réfléchir. Quand j'ai trouvé un bon set de paramètre, vérifier si il y a bien additivité.}

Pistes pour diminuer la variance du routage :
- wrapper les maps (essayer d'abord avec un stimulis en milieu de map)
- connexion 8-connexe
- superposer plusieurs maps E et plusieurs map I et faire la moyenne
- rendre certains le transfert des spike aux quatres premiers voisins
- rendre des connexions obsolètes avec une ou deux computations suite au transfert d'un spike

# Huitième partie

# Annexe

# Neuvième partie

# TODO

TODO expliquer les résultats pour avoir une meilleur compréhension des paramètres
TODO g1 additif,g2 additif => ? a*g1+b*g2+c additif =