

Table des matières

I	Introduction	3
II	Network	5
0.1	Objectif	6
0.2	Genericité	6
0.3	Le routage probabilistic flooding	6
0.4	Le modèle probabilistic flooding spike	6
III	Stimulus	8
0.5	Objectif	9
0.6	Utilisation	9
IV	Génération de données	12
0.7	Introduction	13
0.8	Spikes reçus jusqu'alors	13
V	Additivité	14
0.9	Théorie	15
0.10	Utilisation du script	16
0.11	Résultats	16
VI	Propagation des spikes	20
0.12	Utilisation du script	21
0.13	Résultats	21
0.14	Réflexions	21
VII	Probabilistic flooding et I/E	25
0.15	Utilisation du script	26
0.16	Résultats	27

0.17	Application sur dnfsim	27
0.18	Variance pour $\{20,40,60,80\}$ _asend	30
0.18.1	Utilisation du script	30
0.18.2	Résultats	30

VIII TODO 38

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 générale un nœud du graphe fera office de neurone et une arrête entre deux nœuds, une synapse.

Jusqu'alors nous travaillions 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 décharge en tension) aux neurones afférents. Cet envoi 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.

Afin d'alléger le modèle CNFT dans son ensemble, on va s'intéresser à des type de routage qui ne nécessitent 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 relatives à ces routages et comment les analyser. La mise en pratique sera faite avec le routage Probabilistic flooding. Ce routage connecte les neurones seulement voisins de manière 4-connexe. Avant l'envoi d'un spike, le neurone émetteur fait un test de Bernoulli avec un poids particulier (qu'on appellera *weight* 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 Chappellet 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 Chappellet 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 concentrés sur un framework de test pour ces réseaux. Ce dernier permet de générer un jeu de données 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 intégré 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 intuitives. Le programme se doit d'être évolutif. On s'oriente vers la création 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 (paquets) 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 paquets **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 manière 4 connexes i.e chaque nœud est lié à son voisin nord, sud, est et ouest. On associe à tous les nœuds 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 Bernoulli paramétrée 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 créé 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

particulier. 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'événements sur ce réseau. Ça consistera à envoyer des paquets sur le graphe 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'envoi de stimulus, i.e l'ajout de paquets 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 $t=0$:

```

1  asend.stimulis
2  <?xml version="1.0" encoding="UTF-8"?>
3  <programmation class="main.java.network.generic.packet.Spike">
4      <time t="0.0">
5          <add indice="0"/>
6      </time>
    </programmation>

```

On retrouve ici trois balises :

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

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

i.e on ne peut pas envoyer des stimulus 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 :

```

1  ipv4.stimulis
2  <?xml version="1.0" encoding="UTF-8"?>
3  <programmation class="main.java.network.generic.packet.IPv4Datagramme">

```

```

3      <time t="0.0">
4          <add indice="0">
5              <params> A </params>
6              <params> yes </params>
7              <params> ca fonctionne ! </params>
8          </add>
9      </time>
10     <time t="0.1">
11         <add indice="80">
12             <params> B </params>
13             <params> yes </params>
14             <params> ca fonctionne ! </params>
15         </add>
16     </time>
17 </programation>

```

On découvre la balise `<params>`. L'utilisation de cette balise est directement liée à la structure du constructeur de la classe spécifiée dans la balise `programmation`. 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
1 public class Packet {
2
3     private int size;
4
5     protected Packet(Object ... params){
6     }
7
8     //...

```

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
1 public final class Spike extends Packet {
2
3     public Spike(Object ... params) {
4         setSize(1);
5     }

```

```
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 noter 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 données conséquent concernant différentes caractéristiques du routage.

0.8 Spikes reçus jusqu'alors

Dans cette section, nous allons nous intéresser à la génération du jeu de données représentant la quantité de spikes reçus 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 PFMModel (probabilistic flooding model), elle envoie des stimuli 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 stimuli).

Un exemple d'appel de cette commande :

```
python generateData.py --weights 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 PFMModel pour le moment mais il est facilement extensible à d'autres 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** 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** $S_{p,n,t}$ est le placement du paquet 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** A_n est le **stimulis** $S_{s,n,0}$

Definition 3. (*Configuration*)

Une **configuration** C est un ensemble de **stimulis**. On parle d'initialisation I si c'est un ensemble d'activations.

Definition 4. (*Additivité*)

Soit A^0, A^1, \dots, A^K des activations où $A^i = A_{n_i} \forall i$.

On pose l'initialisation $I = \{A^k \mid n \in [0, K] \text{ et } K > 0\}$

On pose $I_k = \{A^k\} \forall n \in [0, K]$

Soit $M_{t,n,I}$ 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 I .

On pose $M_{t,I} = (M_{t,0,I}, M_{t,1,I}, \dots, M_{t,card(N),I})$ et $M_{t,I_i} = (M_{t,0,I_i}, M_{t,1,I_i}, \dots, M_{t,card(N),I_i})$

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

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

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 forcément une initialisation.

0.10 Utilisation du script

Afin de tester cette propriété d'additivité sur des résultats expérimentaux, nous avons créé un script en Scilab. Ce dernier lit un ensemble de données de plusieurs répétitions 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 veut tester l'additivité dans un milieu épuré. On prend une grille 9x9 de nœuds. On rajoute un spike en haut à gauche(a_send) et en bas à droite(b_send). On distingue les deux scénarios. On regarde la répartition des spikes au bout de 1,3 et 5s. On fait de même en rajoutant les spikes au deux endroits dès le début(ab_send).

On lance la commande suivante :

```
scilab -nw -f Additivite.sce -args  
'9' '1 3 5'  
'0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0'  
'10 50 100 500 1000' 'a_send b_send' ab_send 0
```

On obtient les trois figures suivantes :

- figure 1 sur la page 17.
- figure 2 sur la page 18.
- figure 3 sur la page 19.

On remarque 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).

On voit un optima autour du poids 0.6 à 5s de computations. Il faudrait voir après 5s de computations. Pourquoi trouve-t-on une singularité en 0.6 et pas en 0.0 ?

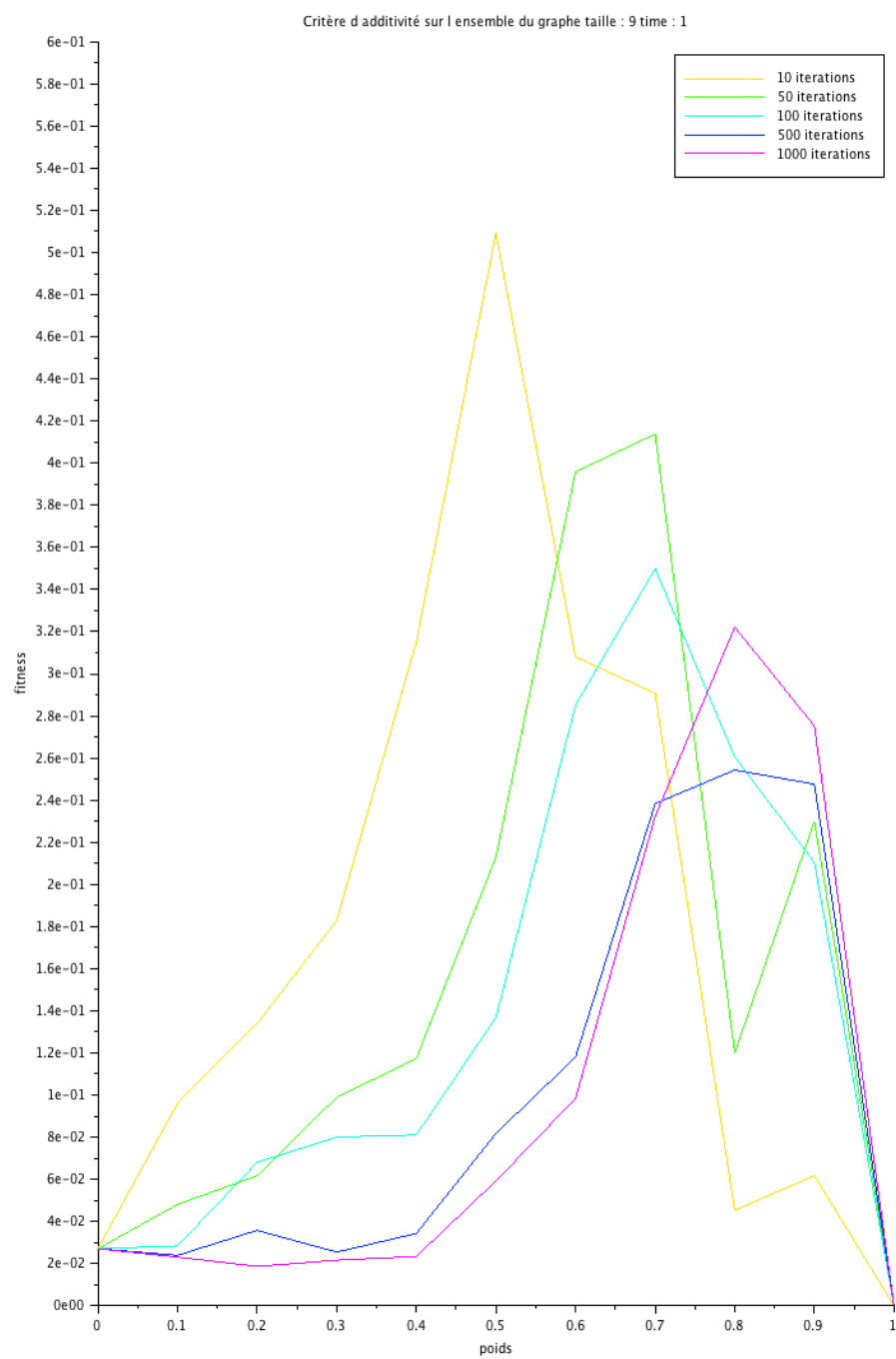


FIGURE 1 – Additivité sur l'ensemble du graphe, envoi d'un spike en A, 10 computations

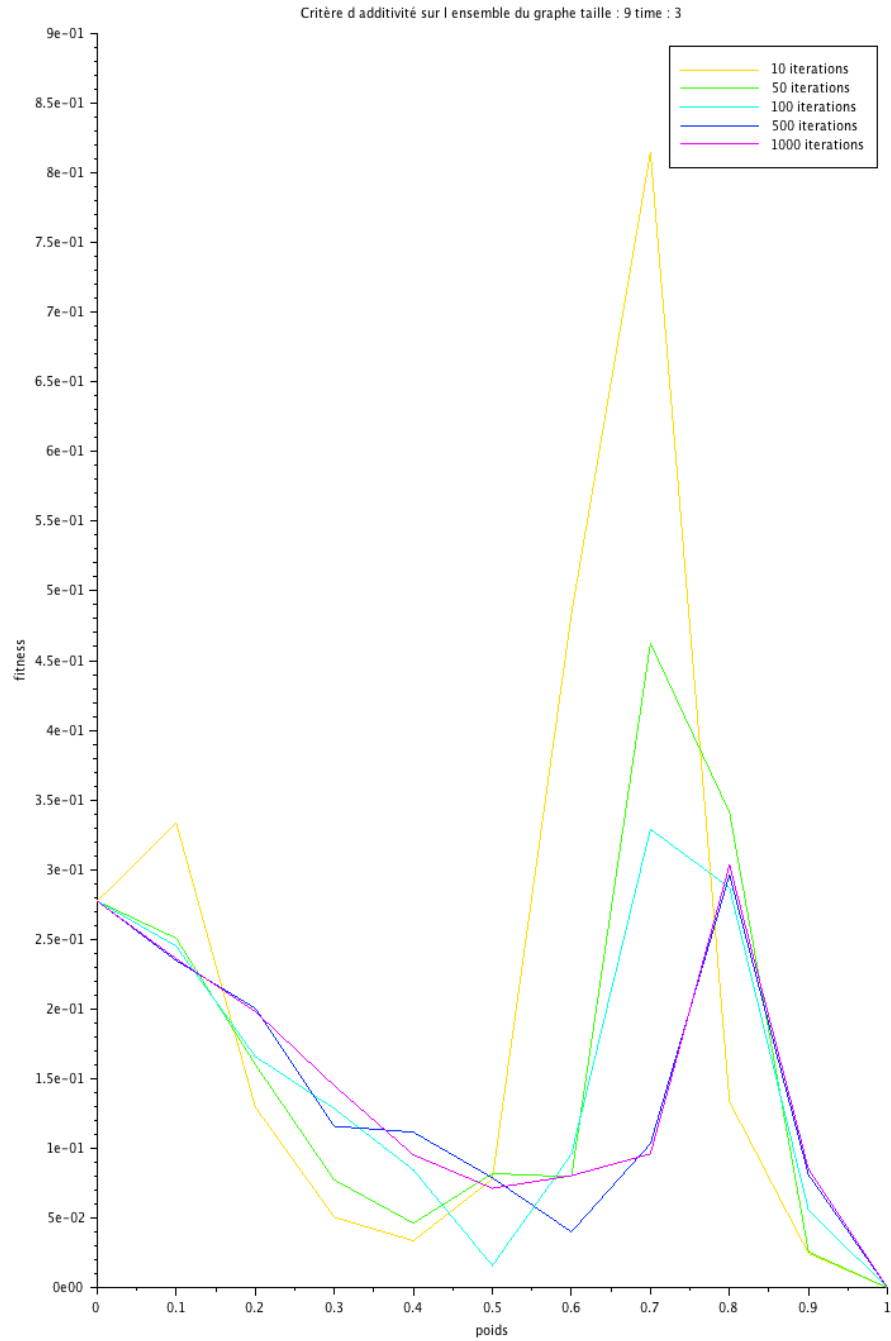


FIGURE 2 – Additivité sur l'ensemble du graphe, envoi d'un spike en A, 30 computations

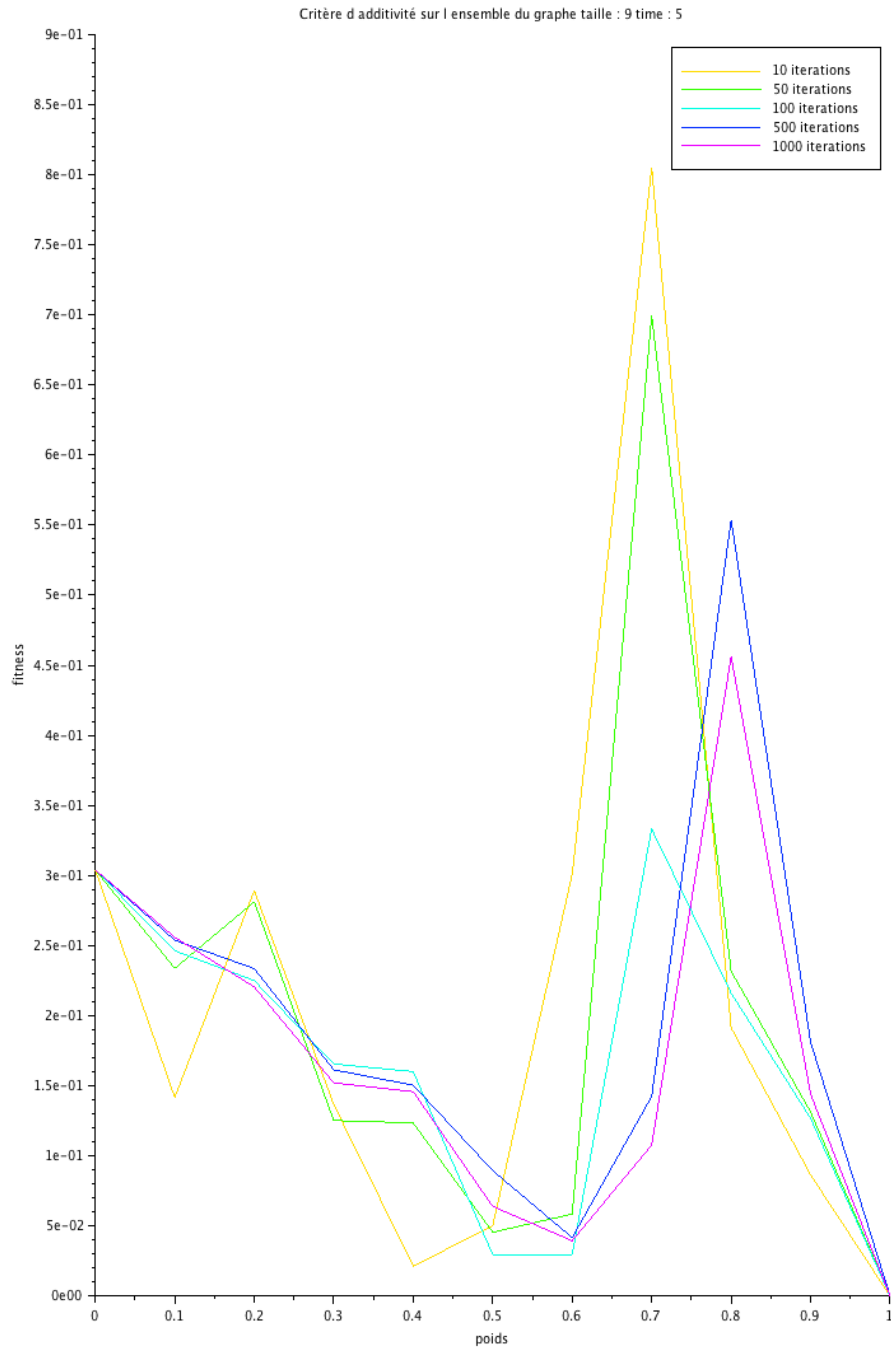


FIGURE 3 – Additivité sur l'ensemble du graphe, envoi d'un spike en A, 50 computations

Sixième partie

Propagation des spikes

Pour qu'un nouveau modèle soit viable, en plus d'avoir un comportement additif, il doit avoir 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'expériences 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'expériences qu'on utilise pour calculer nos statistiques
- a_send_20 : le scénario des expériences (a_send_20.stimulis doit de trouver sous le dossier stimulis/)
- temp/ : le dossier sous lequel les images vont être enregistrées

Cette commande va générer sous le dossier temp un ensemble de graphiques représentant les 3 statistiques sur l'ensemble du graphe et sur la diagonale.

0.13 Résultats

On voit sur l'image 4 de la page 22. 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 l'image 5 de la page 23.

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 l'image 6 de la page 24, du coup on pourrait s'orienter vers des méthodes bootstrap.

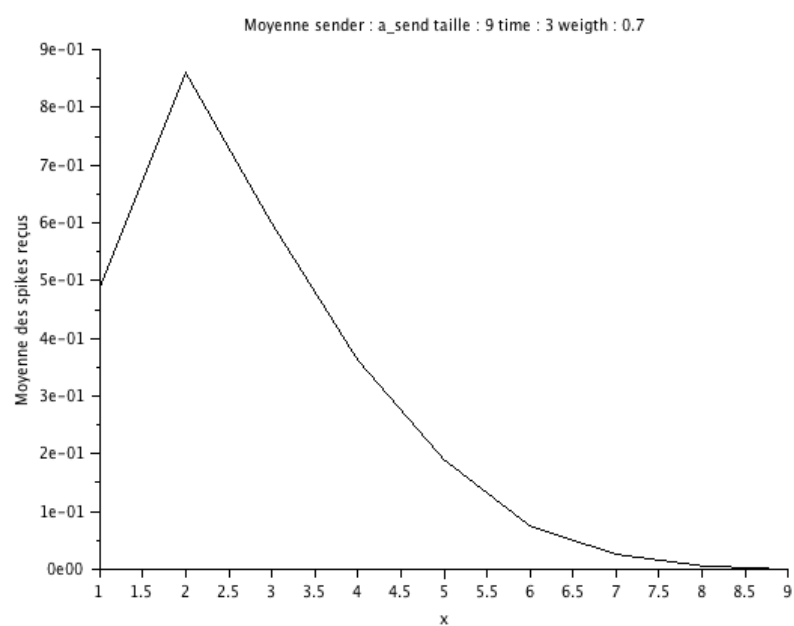


FIGURE 4 – Répartition des spikes en moyenne sur la diagonale, 30 computations, poids 0.7

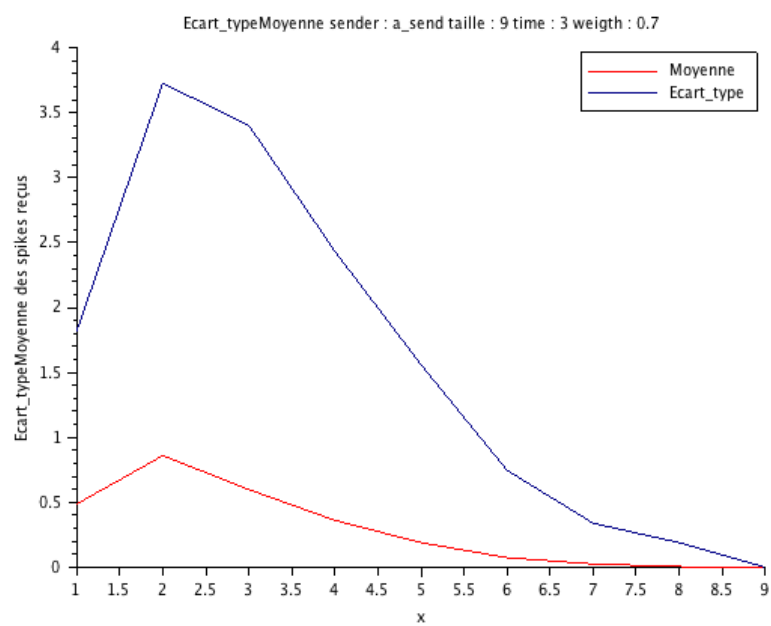


FIGURE 5 – Répartition des spikes en moyenne et écart type sur la diagonale, 30 computations, poids 0.7

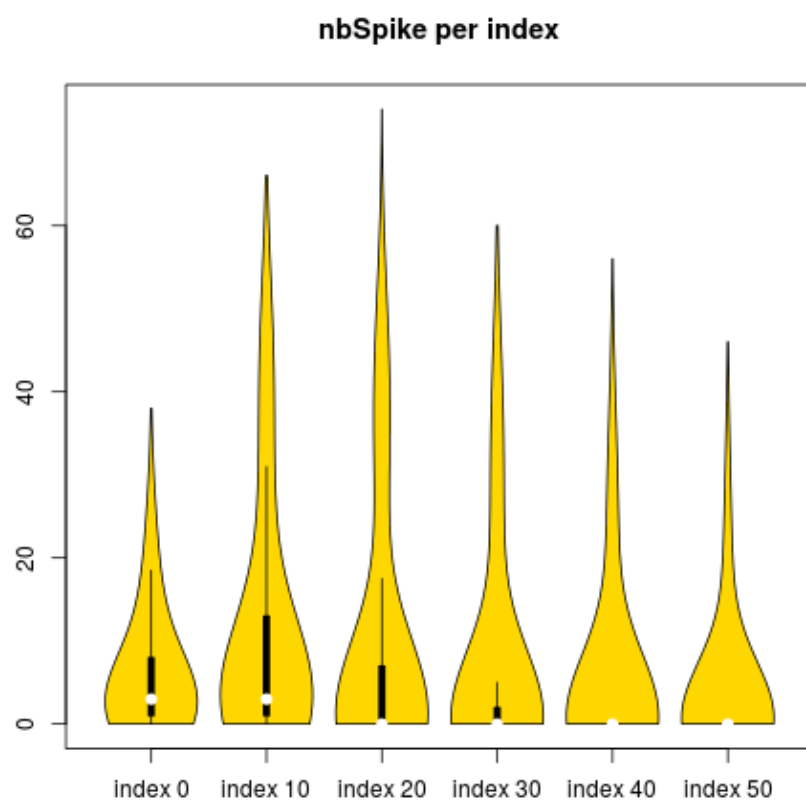


FIGURE 6 – Violins pour la répartition sur la diagonale

Septième partie

Probabilistic flooding et I/E

On rappelle que l'objectif est de trouver un modèle de réseau neuronal simulant le comportement de DNF tout en consommant moins de ressources. Pour cela on va utiliser le routage probabilistic flooding. Ce routage comme vu précédemment s'avère additif et suit une distribution exponentielle/gaussienne selon certains paramètres. On a donc le candidat idéal pour simuler le routage originel de DNF. On va donc mettre en place deux couches de graphes utilisant le routage PF (probabilistic flooding). Une couche avec des signaux excitateurs et une autre avec des signaux inhibiteurs. On applique une transformation affine à ces 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 stimuli. 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 expliqué précédemment, on superpose deux couches de graphes suivant le routage probabilistic flooding puis on 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`, `bseend` ...)
- `repetition` : le nombre d'expériences 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 : image 7 de la page 27 .On lance :

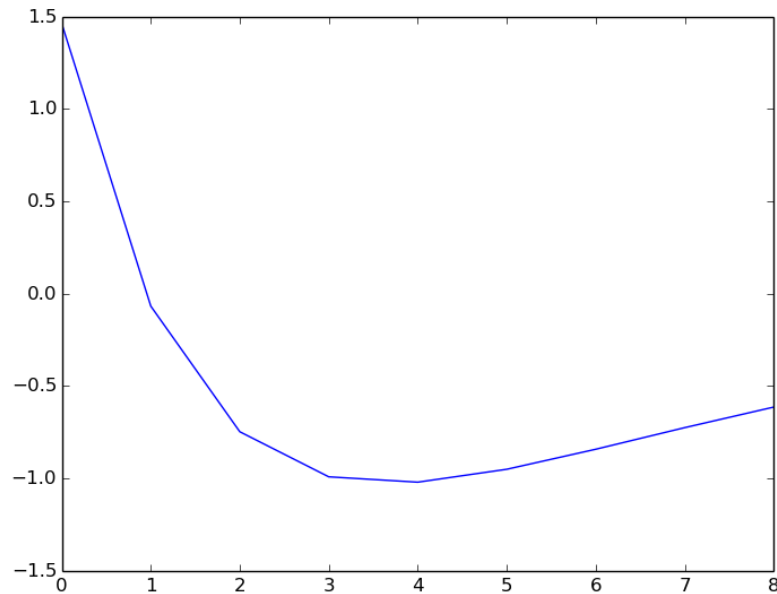


FIGURE 7 – Différence théorique de gaussiennes

```
scilab -nw -f InhibiteurExcitateur.sce -args  
9 a_send 1000 0.7 7 1.0 0.0 0.7 3.0 4.5 -0.9
```

Et on récupère une courbe de cette forme : image 8 de la page 28. La correspondance nous a semblé suffisante pour des tests en conditions réelles.

Par contre sur l'image 9 de la page 29, on voit déjà un écart type très élevé de la courbe.

0.17 Application sur dnfsim

On va donc utiliser cette combinaison linéaire de Probabilistic flooding graphs 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

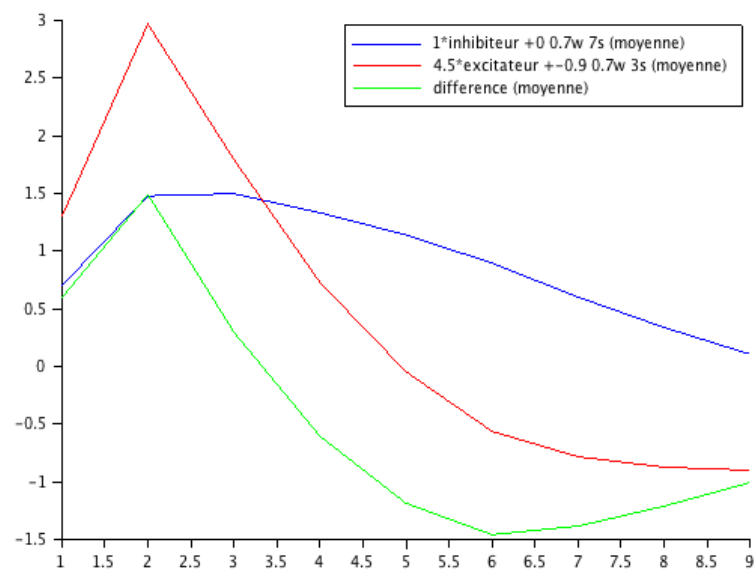


FIGURE 8 – Différence expérimentale de gaussiennes

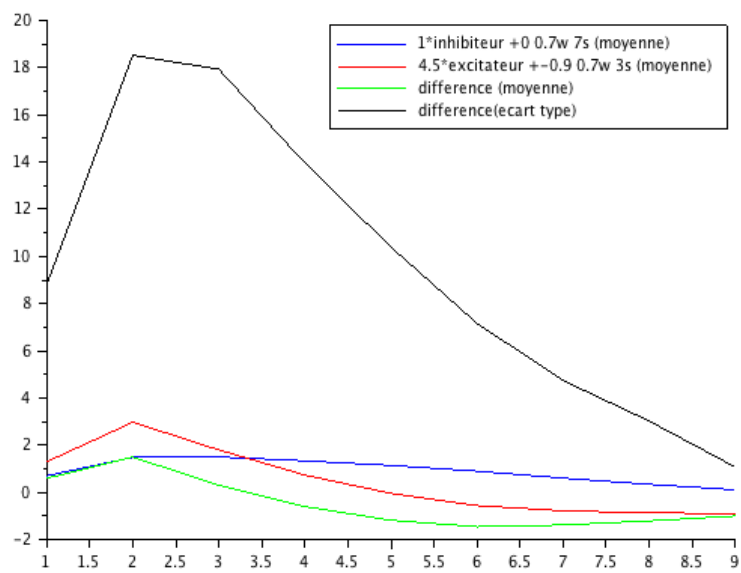


FIGURE 9 – Différence expérimentale de gaussiennes et écart type de cette différence

tournent. Ça fonctionne très mal, la variance est très élevée, on a rarement le focus d'une bulle. De toutes façons on aurait du s'en douter en voyant l'image 9 de la page 29. 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 se placer dans un scénario de test afin de voir si notre hypothèse est plausible et diminue en effet la variance. Pour ça 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 en ligne de commande.

```
scilab -nw -f MultiBootstrap.sce taille weigth time repetition
```

avec comme paramètres :

- taille : la taille du réseau (la racine)
- repetition : le nombre d'expériences qu'on utilise pour calculer nos statistiques
- weigth : le poids pour le graphe
- time : le temps de computation (avec $dt = 0.1$)

Ce script affiche la variance selon les configurations 1/20/40/60/80 ainsi que l'intervalle bootstrap à 95

0.18.2 Résultats

Afin de voir si on a de meilleurs résultats en terme de variance que les images 10 de la page 31 et 11 de la page 32

On lance

```
scilab -nw -f MultiBootstrap.sce 9 0.7 3 1000
```

et

```
scilab -nw -f MultiBootstrap.sce 9 0.7 7 1000
```

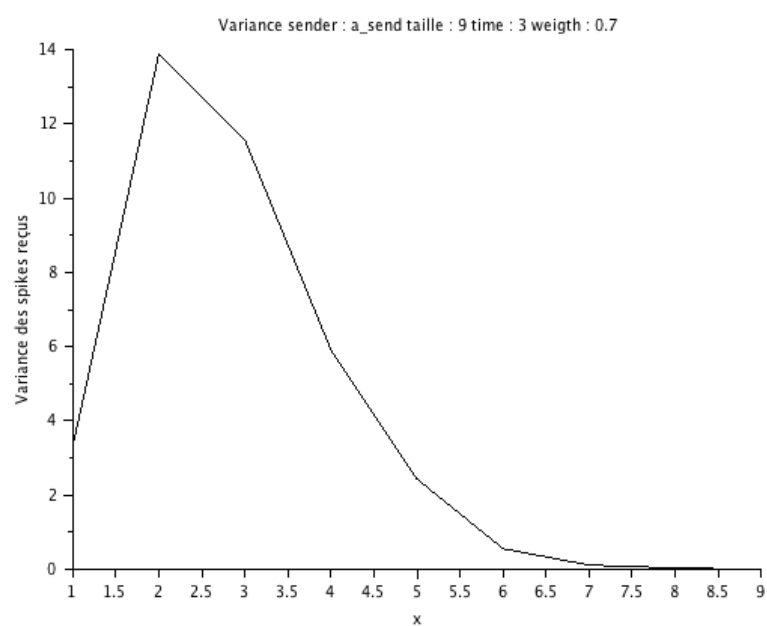


FIGURE 10 – Variance de la répartition des spikes sur la diagonale lors de l’envoi d’un stimulus en A. 30 computations, poids 0.7.

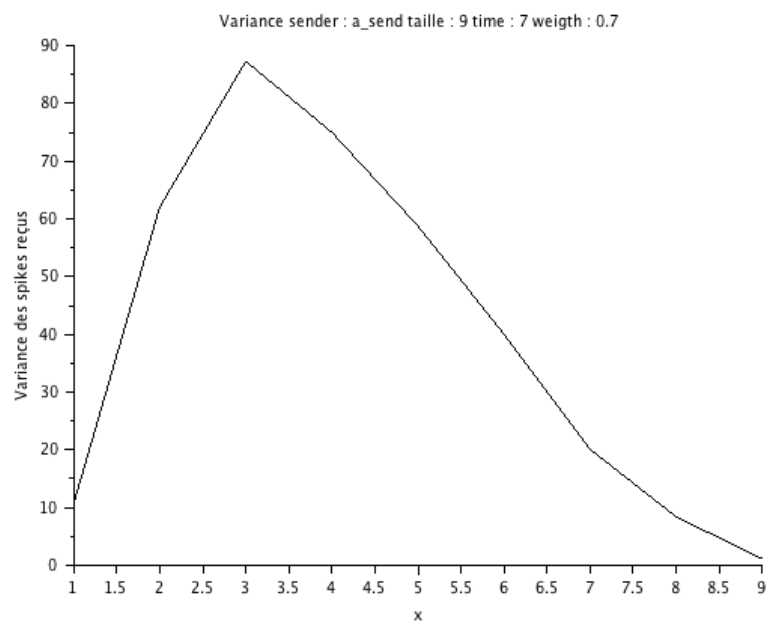


FIGURE 11 – Variance de la répartition des spikes sur la diagonale lors de l’envoi d’un stimulus en A. 70 computations, poids 0.7.

Voici les résultats :

- 12 de la page 33
- 13 de la page 34
- 14 de la page 35
- 15 de la page 36

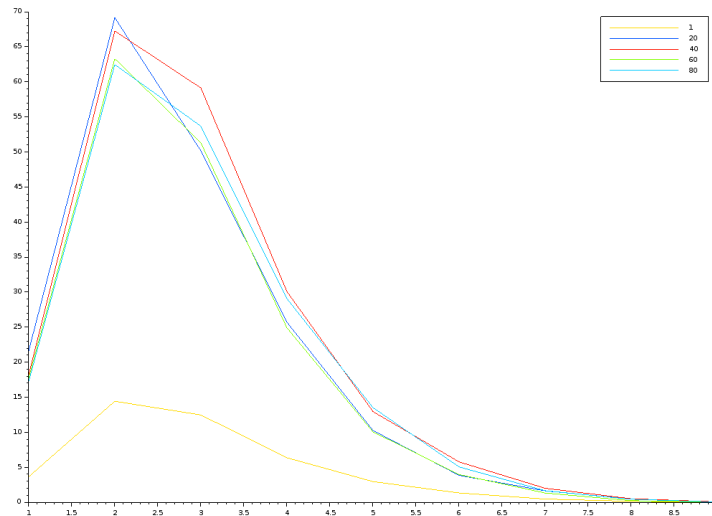


FIGURE 12 – Variances de la répartition des spikes sur la diagonales pour 1/20/40/60/80 envois de spikes en A. 30 computations, poids 0.7.

Comme on peut le constater, notre hypothèse se révèle être complètement fausse, on a beaucoup plus de variance en multipliant les spikes à l'origine. (problème de programmation ?) . A mon avis c'est normal. La variance est plus élevée car la moyenne est plus élevée. Il faudrait calculer la variance sur la même variable aléatoire (nombre de spikes reçus sur la diagonale) mais divisée par le nombre de spikes envoyés. Par contre, ça n'expliquerait pas pourquoi les variances avec 20/40/60/80 spikes sont similaires.

On a quelques pistes pour diminuer la variance du routage :

- wrapper les maps
- connexion 8-connexe
- superposer plusieurs maps E et plusieurs map I et faire la moyenne
- rendre certain le transfert des spike aux quatre premiers voisins
- rendre des connexions obsolètes après une ou deux computations suite

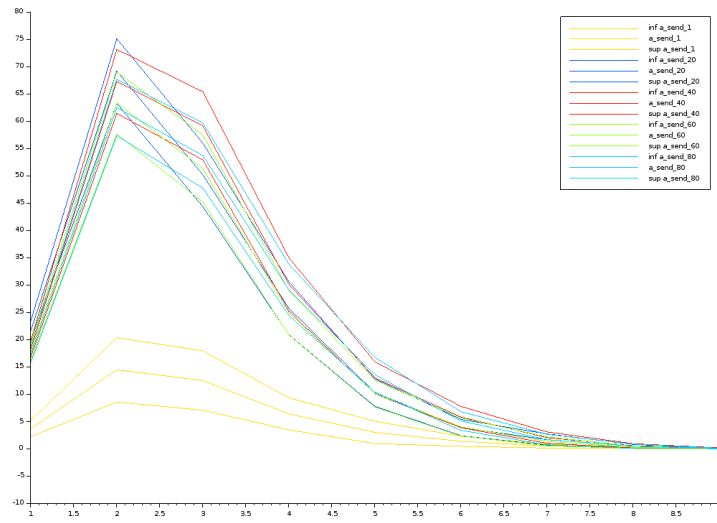


FIGURE 13 – Variances (et son intervalle de confiance) de la répartition des spikes sur la diagonales pour 1/20/40/60/80 envois de spikes en A. 30 computations, poids 0.7.

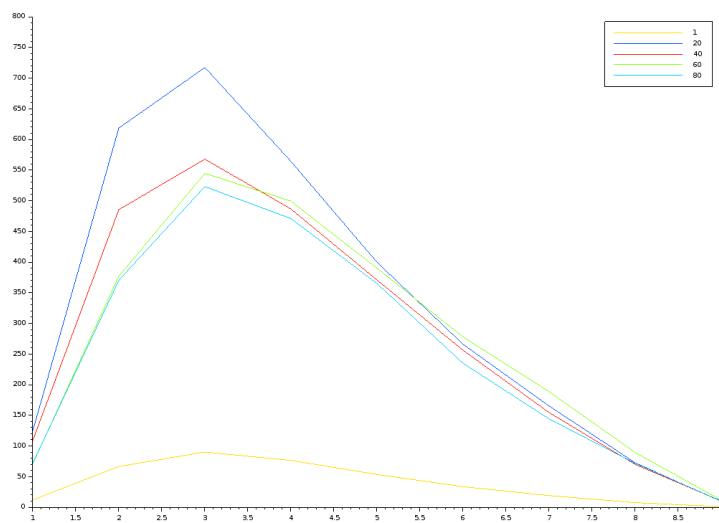


FIGURE 14 – Variances de la répartition des spikes sur la diagonales pour 1/20/40/60/80 envois de spikes en A. 70 computations, poids 0.7.

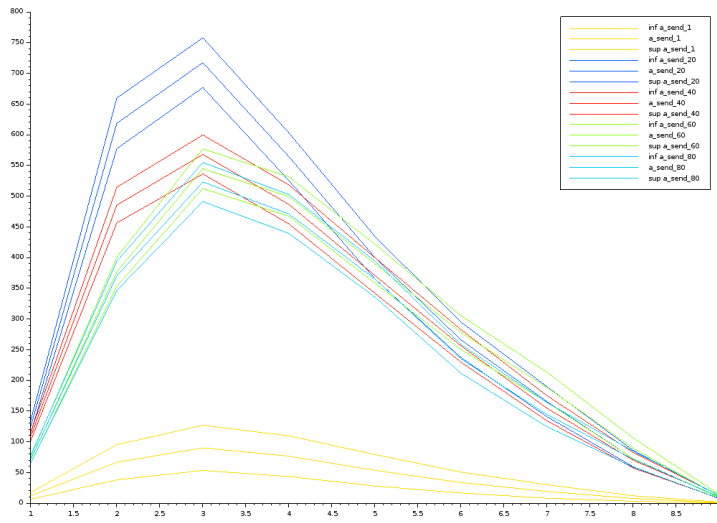


FIGURE 15 – Variances (et son intervalle de confiance) de la répartition des spikes sur la diagonales pour 1/20/40/60/80 envois de spikes en A. 70 computations, poids 0.7.

au transfert d'un spike

Huitième partie

TODO

- expliquer les résultats pour avoir une meilleur compréhension des paramètres
- g_1 additif, g_2 additif $\Rightarrow a \cdot g_1 + b \cdot g_2 + c$ additif?
- tous les scripts scilab travaillent avec le dossier `toric_F`, faire en sorte de choisir entre toric et non toric.