

Aix-Marseille Université
Mémoire de Recherche
présenté en vue de l'obtention du
MASTER de NEUROSCIENCES
(Spécialité: NIC)

**OB-V1 : un modèle de détection de
l'orientation dans l'aire visuelle primaire**



Par :	Fernand David Arbib
Responsable de Stage :	Laurent U. Perrinet http://invibe.net/LaurentPerrinet Laurent.Perrinet@univ-amu.fr
Laboratoire :	Équipe Inference in Visual Behaviour (InViBe) Institut de Neurosciences de la Timone UMR 7289, CNRS / Aix-Marseille Université 27, Bd. Jean Moulin, 13385 Marseille Cedex 5, France

Juin 2016

Table des matières

1	Introduction	3
1.1	Motivation	3
1.2	Contexte scientifique	4
1.2.1	Organisation des orientations	4
1.2.2	Réponse à une orientation	4
1.3	Plan du mémoire	5
2	Le réseau de neurones artificiels	7
2.1	Le neurone intègre et décharge	7
2.1.1	Implémentation sous BRIAN	8
2.1.2	Simulation d'un réseau de neurones avec BRIAN (réseau CUBA)	9
2.1.3	Implémentation sous Nest	11
2.1.4	Implémentation sous PyNN	13
2.1.5	Le processus de Poisson	16
2.1.6	Effet de variations de paramètres cellulaires et non-cellulaires	16
2.2	Le réseau récurrent aléatoire	17
2.2.1	Effet de variations de paramètres cellulaires	18
2.2.2	Choix de la dynamique de la synapse (fonction de decay)	19
2.2.3	Rôle du taux de décharge de l'entrée	23
2.2.4	Rôle du poids de l'entrée	23
2.2.5	Courbes de taux de décharge en fonction de l'activité et du poids de l'entrée	24
2.2.6	Effet d'une covariation de l'activité d'entrée et de son poids :	70
2.2.7	Rôle du poids global	70
2.2.8	Courbe de taux de décharge en fonction des poids synaptiques	82
2.2.9	Courbes de taux de décharge en fonction des paramètres de sparseness	83
2.3	Les états du réseau	89
2.3.1	Définir le coupling	89
2.3.2	Rôle du coupling	90
2.3.3	L'état balancé	91
2.3.4	Optimisation du coupling g	91

3	Le Ring	98
3.1	Le ring non accordé	99
3.1.1	Effet de la bandwidth d'entrée dans un réseau non récurrent	99
3.1.2	Effet de la bandwidth d'entrée	110
3.1.3	Effet des bandwidths des projections internes	111
3.1.4	Courbes d'accord d'un ring non accordé	111
3.2	Le ring accordé	118
3.2.1	connexion des couches suivant une topologit locale	118
3.2.2	Effet de la bandwidth d'entrée	122
3.2.3	Courbes d'accord d'un ring recurrent	147
4	Discussion et perspectives	181
4.1	Conclusion	181
4.1.1	Résultats obtenus	181
4.2	Projet de thèse	181
4.2.1	Motivation et contexte	181
4.2.2	Théorie	183
4.2.3	Méthodes	184
4.2.4	Résultats attendus	184

Chapitre 1

Introduction

1.1 Motivation

Une entité ne peut être vivante si elle n'interagit pas avec son environnement. La vie, réduit à son plus simple appareil, comporte du code génétique. Pour se perpétuer, ce code génétique doit permettre de produire des molécules qui vont former le matériel nécessaire à sa protection, à son alimentation et enfin à sa reproduction. La cellule remplit ces rôles, aussi, sa membrane et les protéines qui la constituent lui permettent de jouer un rôle d'interface entre le code génétique et l'environnement. Il apparaît que la notion d'interface soit importante dans le vivant. Ainsi, l'interface se conserve et se sophistique, au fil de nombreuses mutations du code génétique et de la sélection naturelle. Cette dernière va permettre l'émergence d'interfaces de plus en plus élaborées, qui vont offrir de multiples manières de filtrer l'environnement et des moyens d'explorer celui-ci.

Avec l'apparition des organismes pluricellulaires et des moyens de communications entre les cellules, les cellules se spécialisent, s'assemblent et constituent des tissus cellulaires, des organes et des systèmes qui vont permettre à l'ensemble de l'organisme d'interagir, d'une manière spécifique au système considéré, avec l'environnement. Le système nerveux en est un exemple. En effet, celui-ci permet de traiter des informations externes ou internes, et une partie de ce traitement comprend diverses fonctions que l'on regroupe dans le concept de perception.

Selon l'approche neurophysiologique de la vision, la perception visuelle est construite sur la photosensibilité de certaines cellules, les cônes et les bâtonnets. Ces cellules et d'autres comme, par exemple, les cellules ganglionnaires, forment la rétine tapissant le fond de l'oeil où, à tout instant, une image de l'environnement se projette. Les cellules photosensibles vont permettre d'encoder les variations locales de luminosité composant l'image en variations de potentiel membranaire. Ces variations de potentiel vont ensuite être codées en influx nerveux qui vont être transmis dans le réseau rétinien pour y être transformés. Les informations visuelles arrivent alors au niveau du cortex visuel primaire via les entrées thalamiques. L'organisation de ce cortex est rétinotopique. Ainsi, l'activité du cortex visuel primaire représente l'espace visuel, composé d'éléments locaux comme, par exemple, les bords orientés et les couleurs. Une question se pose alors lorsqu'on cherche à associer ce substrat biologique à la perception visuelle : comment le système

nerveux central réalise l'intégration de ces éléments locaux afin de constituer un percept global ?

Afin de répondre à cette large problématique, il convient de s'intéresser aux divers constituants de l'image composante par composante. Ainsi, le travail qui vous est présenté est dédié à l'étude de la détection d'orientations. Il s'inscrit dans un projet interdisciplinaire porté par l'équipe InViBe au sein de l'Institut de Neurosciences de la Timone. Ce projet prévoit, outre diverses expérimentations chez l'animal, une approche computationnelle des mécanismes impliqués dans cette détection d'orientations, qui constitue l'objet de ce travail.

1.2 Contexte scientifique

1.2.1 Organisation des orientations

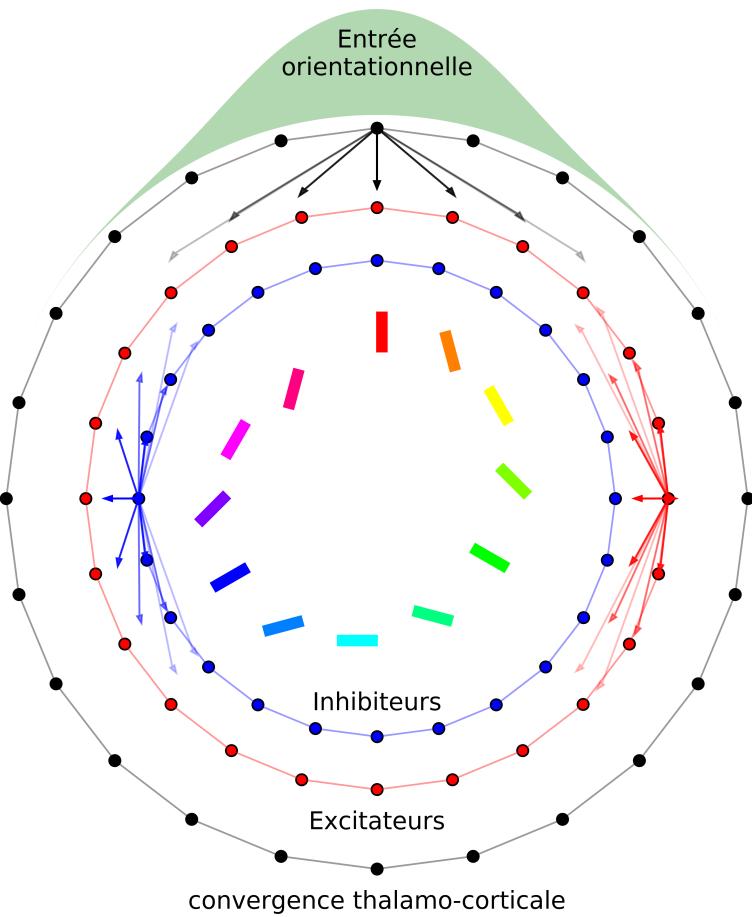
Des électrophysiologistes tels que Hubel et Wiesel ont mis en évidence, chez le chat, que des colonnes corticales du cortex visuel primaire ont une sensibilité préférentielle à une orientation possible de barres de contraste[?]. L'avancée des études sur la sélectivité à l'orientation des neurones corticaux, montrent que le cortex visuel primaire des mammifères carnivores et des primates est comme une carte, où les neurones de même sélectivité à l'orientation sont regroupés en domaines d'iso-orientation. L'organisation particulière de ces domaines ou îlots donne lieu à des propriétés remarquables. En effet, il existe différents voisinages d'un neurone se trouvant dans le cortex visuel primaire. Si celui-ci se trouve à l'intérieur des îlots, il est à proximité de neurones de même, ou proche, sélectivité à l'orientation. Si, en revanche, toutes les orientations sont codées par son voisinage, alors il est à l'intérieur des fractures (ou pinwheels), où la sélectivité à l'orientation varie rapidement [?] [?]. Dans le cortex visuel primaire du rongeur, il n'existe pas de domaines d'iso-orientation, le voisinage d'un neurone quelconque est alors de la deuxième espèce citée.

1.2.2 Réponse à une orientation

De telles organisations suscitent des hypothèses quant à l'intégration des informations sur l'orientation au sein des colonnes corticales du cortex visuel primaire. Une d'entre elles postule que la probabilité de connexion entre les neurones du cortex visuel primaire est exclusivement dépendante de leur distance anatomique [?]. Ce qui signifie que, dans le cas admis où il existe des connexions récurrentes et latérales au niveau cortical, les neurones à l'intérieur des domaines d'iso-orientation intègrent des informations provenant de neurones de même préférence à l'orientation. Ainsi, la réponse de ces neurones est fortement sélective et est robuste à la richesse en orientations d'un stimulus. Cela veut dire également qu'à proximité des fractures, et dans le cortex visuel primaire du rat, les neurones devraient avoir une faible sélectivité à l'orientation car ils intègrent les informations provenant de neurones sélectifs à différentes orientations. Ce n'est pourtant pas ce qui est montré expérimentalement. Une étude explique alors ce paradoxe. En effet, il a été théoriquement démontré que la réponse de ces neurones, supposés peu sélectifs, peut

être plus sélective à l'orientation si le réseau du cortex visuel primaire est dans un état balancé entre l'excitation et l'inhibition [?].

Nous nous proposons donc d'étudier la réponse d'un réseau de neurones artificiel, un ring, à différents stimuli visuels, des motion clouds [4], dont nous ferons alors varier la richesse en orientations. En effet, le contenu en orientations de chaque stimulus peut être défini de façon quantitative en modulant une certaine valeur de bandwidth B_θ caractérisant l'entrée visuelle. Le but est d'implémenter le réseau, de façon à ce qu'il possède des propriétés similaires à celles évoquées plus haut concernant le cortex visuel primaire. Nous comparerons alors la réponse de ce réseau à des données physiologiques.



Réseau de neurones organisé en “ring”.

1.3 Plan du mémoire

Dans un premier temps, nous traiterons des réseaux de neurones artificiels, de leur implémentation ainsi que de l'étude de leurs comportements. Nous y étudierons notamment le réseau récurrent aléatoire et tenterons de mettre en évidence certains de ces états,

notamment l'état balancé. Ensuite, nous étudierons le ring, un réseau récurrent aléatoire disposant d'une certaine organisation de neurones. Nous tenterons alors de montrer que ce réseau est un modèle satisfaisant du codage de l'orientation au sein du cortex visuel primaire.

Chapitre 2

Le réseau de neurones artificiels

Dans ce chapitre, nous développerons différents aspects de l'implémentation en Python de réseaux de neurones artificiels, en allant du plus simple au plus complexe. Nous commencerons donc par introduire le neurone “intègre et décharge” et traiterons de son implémentation dans un réseau simple. Cette implémentation nous permettra de tester et comparer différents simulateurs de réseaux neuronaux. Puis, nous traiterons du ‘random recurrent neural network’ (RRNN) et de l’exploration de ses régimes dynamiques. Enfin, nous étudierons différents états du RRNN, notamment de l’état dit “balancé”.

>>>

2.1 Le neurone intègre et décharge

Le neurone intègre et décharge, ou “integrate and fire”, est un modèle répandu du neurone biologique lorsque l’on désire simuler le fonctionnement de réseaux de neurones. Contrairement à d’autres modèles plus réalistes comme celui d’Hodgkin et Huxley [?], il néglige l’effet des courants ioniques sur le potentiel membranaire. Mais cette approximation permet de gagner en temps de calcul, en facteur d’ordre 2 [?] et offre la possibilité de simuler un réseau de neurones en un temps raisonnable. Sa variante la plus répandue est le neurone “leaky integrate and fire” (LIF), qui permet de prendre en considération des courants de fuite, qui sont regroupés en un terme de résistance membranaire.

La dynamique du potentiel de membrane V d’un neurone LIF est définie par :

$$\tau_m \frac{d}{dt} V = E_L - V + R_m I_e$$

avec

- τ_m : constante de temps membranaire
- E_L : potentiel de repos
- R_m : résistance membranaire
- I_e : courant entrant

Un neurone LIF génère un potentiel d’action quand V dépasse une valeur seuil $V_{threshold}$. Une fois ce seuil dépassé V est réinitialisé à une valeur V_{reset} .

Différents modèles de neurones LIF ont été développés. Certains de ces modèles tentent de concilier l'efficacité en temps de calcul du neurone LIF et le bioréalisme du modèle de Hodgkin et Huxley. Ainsi, il est possible de distinguer les neurones LIF conductance-based (COBA) des neurones LIF current-based (CUBA). Le modèle CUBA considère les entrées synaptiques comme des courants entrants qui vont, comme l'équation au dessus l'indique, faire évoluer le potentiel membranaire. Le modèle COBA, en revanche, traduit les entrées synaptiques par des changements de conductance, de telle sorte que des entrées vont amplifier les effets d'autres entrées sur le potentiel de membrane. Le modèle COBA est donc un peu plus bioréaliste que le modèle CUBA mais aussi plus complexe à étudier [3].

Ce projet porte sur la modélisation du cortex visuel primaire et nous pousse donc à choisir notre modèle neuronal parmi ceux disposant d'un minimum de réalisme. C'est pourquoi notre choix se porte sur le modèle COBA.

>>>

2.1.1 Implémentation sous BRIAN

Différents simulateurs sont testés afin d'évaluer leurs souplesse d'utilisation ainsi que leur efficacité. Nous commençons donc par tester Brian. (<http://brian-simulator.org>). La particularité de Brian est qu'il permet à son utilisateur de créer un modèle en rentrant l'équation différentielle de sa dynamique. Il permet donc une souplesse maximale quant au choix du modèle neuronal, tout en prenant en charge la mise en réseau des neurones. Afin de se familiariser avec cet outil, nous implementons un simple neurone intègre et décharge. (voir https://brian.readthedocs.org/en/latest/tutorial1_basic_concepts.html#tutorial-1-basic-concepts)

```
>>> from brian import *
... tau = 20 * msecond
... Vt = -50 * mvolt
... Vr = -60 * mvolt
... El = -49 * mvolt
... psp = 0.5 * mvolt
...
... G = NeuronGroup(N=40, model='dV/dt = -(V-El)/tau : volt', threshold=vt,
...
... C = Connection(G, G, sparseness=0.1, weight=psp)
...
... M = StateMonitor(G, 'V', record=0)
...
... G.V = Vr + rand(40) * (Vt - Vr)
...
... run(200 * msecond)
```

```

-----
ImportError                                     Traceback (most recent call last)

<ipython-input-2-f91817fbf293> in <module>()
----> 1 from brian import *
      2 tau = 20 * msecond
      3 Vt = -50 * mvolt
      4 Vr = -60 * mvolt
      5 El = -49 * mvolt

ImportError: No module named 'brian'

>>> figure(figsize=(15,5))
... plot(M.times/ms, M[0]/mV)
... xlabel('Time (in ms)')
... ylabel('Membrane potential (in mV)')
... title('Membrane potential for neuron 0')
... show()

```

2.1.2 Simulation d'un réseau de neurones avec BRIAN (réseau CUBA)

Nous implémentons également un réseau de neurones CUBA.

https://brian.readthedocs.org/en/latest/tutorial2_connections.html#tutorial-2-connections

```

>>> taum = 20 * ms
... taue = 5 * ms
... taui = 10 * ms
... Vt = -50 * mV
... Vr = -60 * mV
... El = -49 * mV
... we = (60 * 0.27 / 10) * mV
... wi = (20 * 4.5 / 10) * mV

>>> eqs = Equations('''
...     dV/dt  = (ge-gi-(V-El))/taum : volt
...     dge/dt = -ge/taue                : volt
...     dgi/dt = -gi/taui                : volt
... ''')

>>> G = NeuronGroup(N=4000, model=eqs, threshold=Vt, reset=Vr)

```

```

...
... Ge = G.subgroup(3200)
... Gi = G.subgroup(800)
...
... Ce = Connection(Ge, G, 'ge', sparseness = 0.02, weight=we)
... Ci = Connection(Gi, G, 'gi', sparseness = 0.02, weight=wi)

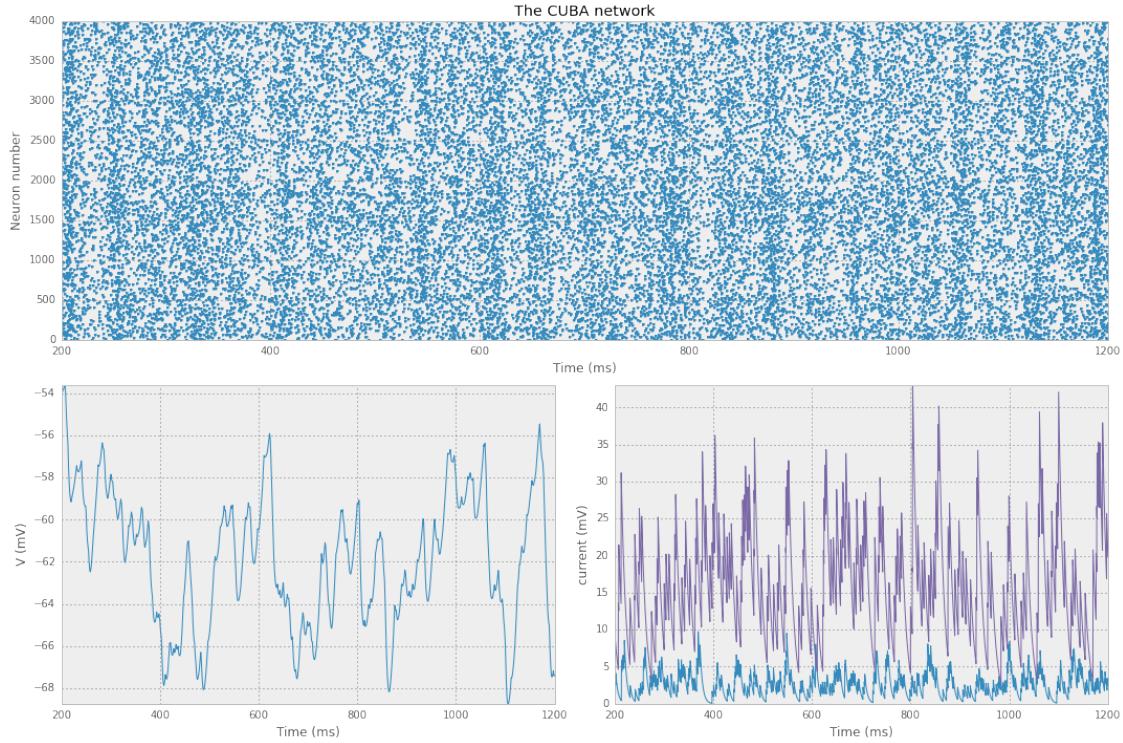
>>> M = SpikeMonitor(G)
... Mv = StateMonitor(G, 'V', record=0)
... Mge = StateMonitor(G, 'ge', record=0)
... Mgi = StateMonitor(G, 'gi', record=0)

>>> G.V = Vr + (Vt - Vr) * rand(len(G))

>>> run(1000 * ms)

>>> figure(figsize=(15, 10))
... subplot(211)
... raster_plot(M, title='The CUBA network', newfigure=False)
... axis('tight')
... subplot(223)
... plot(Mv.times/ms, Mv[0]/mV)
... xlabel('Time (ms)')
... ylabel('V (mV)')
... axis('tight')
... subplot(224)
... plot(Mge.times / ms, Mge[0] / mV, label='ge')
... plot(Mgi.times / ms, Mgi[0] / mV, label='gi')
... xlabel('Time (ms)')
... ylabel('current (mV)')
... axis('tight')
... # legend('upper right')
... tight_layout()
... show()

```



2.1.3 Implémentation sous Nest

Nest (<http://www.nest-simulator.org/>) est un autre simulateur de réseaux de neurones spécialisé dans la modélisation de larges réseaux de neurones simplifiés. Celui-ci, en revanche, ne permet pas de définir un modèle neuronal fin en saisissant explicitement des équations différentielles. Le choix est fait d'optimiser plutôt ces modèles en les compilant de façon efficace. Comme avec Brian, nous implementons une simulation d'un simple modèle "integrate and fire" en suivant la même formalisation.

```
>>> import nest
... import matplotlib.pyplot as plt
... neuron = nest.Create("iaf_neuron")
...
... nest.GetStatus(neuron)
...
... nest.GetStatus(neuron, "I_e")
... nest.GetStatus(neuron, ["V_reset", "V_th"])
...
... nest.SetStatus(neuron, {"I_e": 376.0})
...
... nest.GetStatus(neuron, "I_e")
```

```
(376.0,)
```

```
>>> spikedeceptor = nest.Create("spike_detector",
...                                 params={"withgid":True, "withtime":True})
...
... multimeter = nest.Create("multimeter")
... nest.SetStatus(multimeter, {"withtime":True, "record_from": ["V_m"]})
...
... noise = nest.Create("poisson_generator", 2)
... nest.SetStatus(noise, [{"rate": 8000.0}, {"rate": 15000.0}])
... nest.SetStatus(neuron, {"I_e": 0.0})
...
... syn_dict_ex = {"weight": 1.2}
... syn_dict_in = {"weight": -2.0}
... nest.Connect([noise[0]], neuron, syn_spec=syn_dict_ex)
... nest.Connect([noise[1]], neuron, syn_spec=syn_dict_in)
...
... nest.Connect(multimeter, neuron)
... nest.Connect(neuron, spikedeceptor)
```

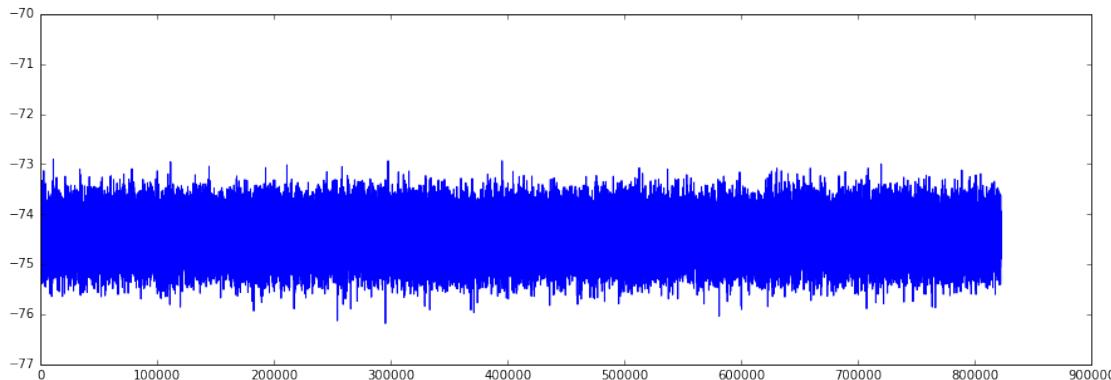
```
>>> %%timeit
```

```
... nest.Simulate(200.0)
```

```
1000 loops, best of 3: 911 µs per loop
```

```
>>> dmm = nest.GetStatus(multimeter)[0]
... Vms = dmm["events"]["V_m"]
... ts = dmm["events"]["times"]
...
... plt.figure(figsize=(15,5))
... plt.plot(ts, Vms)
```

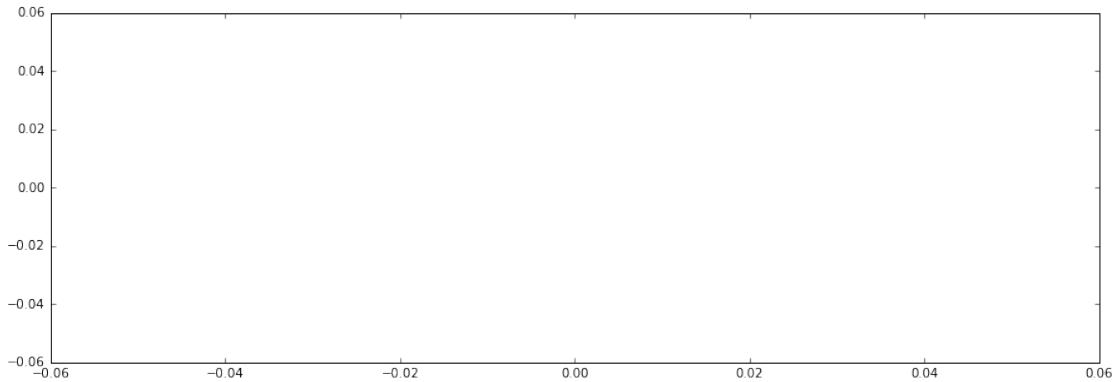
```
[<matplotlib.lines.Line2D at 0x10d369f28>]
```



```

>>> dSD = nest.GetStatus(spikedetector, keys='events')[0]
... evs = dSD["senders"]
... ts = dSD["times"]
... plt.figure(figsize=(15,5))
... plt.plot(ts, evs, ".")
... plt.show()

```



2.1.4 Implémentation sous PyNN

Nous avons vu qu'une implémentation d'un même modèle peut radicalement changer de forme selon le simulateur utilisé. Il peut être utile, afin de ne pas passer trop de temps sur l'apprentissage des différents simulateurs, de trouver un moyen d'harmoniser l'écriture des modèles. PyNN (<http://neuralensemble.org/PyNN>) n'est pas un simulateur mais une API, une interface de description (i.e. programmation), qui permet de réaliser des simulations pouvant s'exécuter sur différents simulateurs, comme Nest ou Brian, sans changer le code d'implémentation. Elle permet également de générer divers graphiques aisément [?].

C'est avec cette interface que nous allons construire des réseaux neuronaux et programmer différents outils pour leur étude. Nous implementons tout d'abord un réseau très simple afin de comparer l'efficacité de Nest et de Brian. Ces simulations permettent de vérifier que les implementations de NEST et Brian fournissent des résultats quasi-identiques au niveau de l'implémentation des équations différentielles des modèles neuronaux. Il s'avère aussi que la simulation avec Nest se fait en un temps plus court que celle effectuée avec Brian. C'est pourquoi désormais nous utiliserons Nest pour la simulation de réseaux de neurones.

```

>>> import pyNN.nest as sim
... import numpy

```

```

...
... from pyNN.utility import get_simulator, init_logging, normalized_filenam
... from pyNN.parameters import Sequence
... from pyNN.random import RandomDistribution as rnd
... from pyNN.utility.plotting import Figure, Panel
...
...
...
... # === Define parameters =====
...
...
... n = 10      # Number of cells
... w = 0.2    # synaptic weight ( $\mu$ S)
... coba_params = {
...     'tau_m'        : 20.0,    # (ms)
...     'tau_syn_E'    : 2.0,     # (ms)
...     'tau_syn_I'    : 4.0,     # (ms)
...     'e_rev_E'      : 0.0,     # (mV)
...     'e_rev_I'      : -70.0,   # (mV)
...     'tau_refrac'   : 2.0,     # (ms)
...     'v_rest'       : -60.0,   # (mV)
...     'v_reset'      : -70.0,   # (mV)
...     'v_thresh'     : -50.0,   # (mV)
...     'cm'           : 0.5}     # (nF)
...
...
... cuba_params = {
...     'tau_m'        : 20.0,    # (ms)
...     'tau_syn_E'    : 2.0,     # (ms)
...     'tau_syn_I'    : 4.0,     # (ms)
...     'tau_refrac'   : 2.0,     # (ms)
...     'v_rest'       : -60.0,   # (mV)
...     'v_reset'      : -70.0,   # (mV)
...     'v_thresh'     : -50.0,   # (mV)
...     'cm'           : 0.5,     # (nF)
...     'i_offset'     : 0.0}
...
... dt          = 0.1          # (ms)
... syn_delay   = 1.0          # (ms)
... input_rate  = 10.0         # (Hz)
... simtime     = 1000.0        # (ms)
...
...
... # === Build the network =====
...
...
... sim.setup()
...
...
... coba = sim.Population(n, sim.IF_cond_alpha(**coba_params),
...                         initial_values={'v': rnd('uniform', (-60.0, -50.0)),
... ...

```

```

...
... cuba = sim.Population(n, sim.IF_curr_alpha(**cuba_params),
...                         initial_values={'v': rnd('uniform', (-60.0, -50.0)),
...                                         'label="cuba")}
...
...
...
... number = int(2*simtime*input_rate/1000.0)
... numpy.random.seed(26278342)
...
...
...
... def generate_spike_times(i):
...     gen = lambda: Sequence(numpy.add.accumulate(numpy.random.exponential,
...                                                 scale=1.0))
...     if hasattr(i, "__len__"):
...         return [gen() for j in i]
...     else:
...         return gen()
... #assert generate_spike_times(0).max() > simtime
...
...
... spike_source = sim.Population(n, sim.SpikeSourceArray(spike_times=generate_spike_times))
...
...
...
... input_conn1 = sim.Projection(spike_source, coba, sim.FixedProbabilityConnector(1.0))
... input_conn2 = sim.Projection(spike_source, cuba, sim.FixedProbabilityConnector(1.0))
...
...
... #----- Recording -----
... spike_source.record('spikes')
...
...
... coba.record('spikes')
... coba[0:n].record(('v', 'gsyn_exc'))
...
...
... cuba.record('spikes')
... cuba[0:n].record(('v'))
...
...
... # === Run simulation =====
...
...
... sim.run(simtime)
...
...
... print("COBA Mean firing rate: ", coba.mean_spike_count()*1000.0/simtime)
... print("CUBA Mean firing rate: ", cuba.mean_spike_count()*1000.0/simtime)
...
...
... # === Clean up and quit =====
...
...
... sim.end()

```

COBA Mean firing rate: 7.2 Hz

CUBA Mean firing rate: 6.5 Hz

2.1.5 Le processus de Poisson

Afin de simuler une entrée bruitée nous utilisons un processus de Poisson. Le processus de Poisson est un processus stochastique, qui permet la génération d'une séquence de potentiels d'action, au cours d'un certain laps de temps et, en fonction d'une probabilité d'occurrence donnée. Il est dit homogène dans le cas où la probabilité d'occurrence d'un potentiel d'action pour tout temps t est la même, et où elle ne dépend donc pas des potentiels d'action survenus à $t - n\Delta t$, n étant un naturel non nul et Δt , le pas de temps [1]. Une telle entrée nous servira ainsi de source d'activité pour les réseaux que nous implémenterons.

2.1.6 Effet de variations de paramètres cellulaires et non-cellulaires

Ici, nous implémentons un réseau composé de deux populations, une population A dite source et une population B. L'activité de la population A est définie par un processus de poisson homogène et la population B comporte des neurones COBA. Les deux populations sont composées chacune de deux neurones et sont connectées entre elles par une projection de type "all to all". C'est à dire que chacun des neurones de la population A est connecté à tous les neurones de la population B. Une première solution est de résoudre analytiquement les équation différentielles [2]. Une exploration numérique de différents paramètres est ici privilégiée. Cette étude permet de tester l'effet de ces paramètres sur le taux de décharge mesuré en sortie du réseau, c'est à dire l'activité de la population B. En effet, la variation de ces paramètres sur un petit réseau de neurones permet d'observer rapidement les phénomènes qu'elle provoque. Les paramètres par défaut sont inspirés de modèles canoniques (<http://neuralensemble.org/trac/PyNN/wiki/StandardModels>) :

Pour chaque paramètre étudié, plusieurs simulations du modèle sont lancées avec différentes valeurs du paramètre. Et pour chaque simulation, le taux de décharge neuronal moyen de la population B est récupéré. Les résultats sont alors affichés dans une courbe de taux de décharge en fonction d'une variation d'un paramètre. Les résultats de ces simulations peuvent être interprétés à partir des courbes de réponse entrée-sortie (I-F) sur les figures et de façon qualitative concernant le taux de décharge de la population B, nous observons sur une augmentation de la valeur des paramètres respectifs :

- taux de décharge de la source : une augmentation quasi-linéaire du taux de décharge,
- poids de la source : une augmentation quasi-linéaire également,
- délai synaptique : pas de variation,
- tau_m : une augmentation logarithmique,
- tau_syn_E : une augmentation linéaire,
- tau_syn_I : pas de variation,
- e_rev_E : une augmentation par paliers,
- e_rev_I : pas de variation,

- v_rest : une augmentation quasi linéaire,
- v_thresh : une diminution en fonction puissance,
- v_reset : une légère augmentation,
- tau_refrac : une légère diminution,
- cm : une diminution exponentielle

Ces observations confirment bien la résolution analytique des équations différentielles et permettent de contrôler que l'on est dans un bon régime [2].

```
>>> import pandas as ps
... import numpy as np
... import matplotlib.pyplot as plt
...
... import pyNN.nest as sim
... #import pyNN.brian as sim
...
... from pyNN.parameters import Sequence
... from pyNN.random import RandomDistribution as rnd
...
... import os

CSAConnector: libneurosim support not available in NEST.
Falling back on PyNN's default CSAConnector.
Please re-compile NEST using --with-libneurosim=PATH
```

2.2 Le réseau récurrent aléatoire

Afin de ne pas avoir à faire face à de nombreuses difficultés à la fois, il convient de développer et d'étudier le modèle de détection d'orientation étape par étape. Ainsi, avant de traiter le réseau en Ring, une généralisation de celui-ci, le réseau récurrent aléatoire ou “random recurrent neural network” (RRNN), est d'abord développée et explorée. Cette étape permet, dans un premier temps, de négliger la topologie du réseau pour se concentrer sur la connectique ainsi que sur la recherche et démonstration d'un état d'équilibre de celui-ci.

Le RRNN utilisé est un réseau comportant mille neurones propres au réseau et cinq cents neurones pour la source, et est constitué de trois populations : une population source, une population excitatrice et une population inhibitrice. La population source mise à part, le réseau contient des neurones du même type que ceux utilisés dans le réseau feedforward étudié jusqu'à présent.

Dès à présent, nous parlons de “connexion” pour désigner une connexion synaptique entre neurones, et de “projection” lorsque nous faisons référence à l'ensemble des connexions des neurones d'une population A aux neurones d'une autre population B. Le poids d'une projection, est alors le poids de toutes les connexions synaptiques de cette projection.

La population source comporte des neurones qui déchargent selon un processus de Poisson homogène. La population excitatrice, E, reçoit l'activité émise par la source via une projection de type "one to one". C'est-à-dire que chaque neurone de la population source est connecté à un seul neurone de la population E. L'activité transformée par cette dernière excite la population inhibitrice.

Enfin, la population inhibitrice, I, dont les neurones possèdent, hormis leur faculté d'inhibition, les mêmes propriétés que les neurones excitateurs, inhibe E. En outre, E et I possèdent des connexions récurrentes et de façon arbitraire (car on peut régler les poids d'interaction), les populations E et I possèdent le même nombre de neurones [?].

```
>>> net = RRNN()

>>> net.model()

(   angle_input  b_exc_exc  b_exc_inh  b_inh_exc  b_inh_inh  b_input  \
0          90        inf        inf        inf        inf        inf

      b_input_exc  c_exc_exc  c_exc_inh  c_inh_exc      ...      v_init_max
0         inf       0.015     0.015     0.015      ...      -49.5

      v_init_min  v_reset  v_rest  v_thresh  w_exc_exc  w_exc_inh  w_inh_exc
0       -53       -70      -60      -50      0.06      0.06      0.6

      w_inh_inh  w_input_exc
0        0.6       0.015

[1 rows x 43 columns],
Segment with 540 spike trains
name: 'segment000'
description: u'Population "NE"\n      Structure : Line\n      y: 0.0\n
Segment with 540 spike trains
name: 'segment000'
description: u'Population "NI"\n      Structure : Line\n      y: 0.0\n

>>>
```

2.2.1 Effet de variations de paramètres cellulaires

Il est primordial de bien choisir les valeurs des paramètres cellulaires pour éviter que l'activité du réseau soit trop basse ou trop élevée. Cette expérience contrôle est nécessaire pour rester dans un régime d'activité où la manipulation des paramètres non cellulaires aura un effet observable sur l'activité du réseau.

Pour ce faire, le même procédé d'exploration utilisé pour le réseau feed-forward est effectué ici. Ainsi, l'effet de la variation de chacun des paramètres cellulaires sur le taux de

décharge peut être observé. Nous rappelons que pour chaque paramètre étudié, plusieurs simulations du modèle sont lancés avec différentes valeurs du paramètre. Et, pour chaque simulation, le taux de décharge neuronal moyen de la population B est récupéré. Les résultats sont alors affichés dans une courbe du taux de décharge mesuré en fonction d'une variation d'un paramètre.

Les résultats obtenus vérifient bien le fait que les paramètres cellulaires sont bien choisis.

2.2.2 Choix de la dynamique de la synapse (fonction de decay)

Les modèles de neurones IFcondexp et IFcondalpha sont ici à nouveau comparés, dans le cadre du RRNN cette fois. Le but est d'éviter d'avoir un réseau qui sature trop facilement afin que les manipulations effectuées aient un effet observable. Les paramètres de taux de décharge d'entrée et de poids de l'entrée sont manipulés. Comme cela a pu être effectué précédemment, pour chacun de ces paramètres, le taux de décharge moyen des neurones du réseau est récupéré et une courbe est générée.

Le rapport F/I est plus important pour un decay en alpha function. Afin d'éviter des saturations de l'activité du réseau et compte tenu des résultats obtenus avec les deux types de neurones, je choisis de conserver l'IF cond exp.

```
>>> import numpy as np
... from RRNN import RRNN
...
...
... net_exp = RRNN()
... net_alpha = RRNN(n_model='cond_alpha')
... datapath_exp = '/tmp/OB-V1_data/cond_exp' + tag
... datapath_alpha = '/tmp/OB-V1_data/cond_alpha' + tag
...
...
... n_sim_each = 20
...
...
... sim_list = [
...     ('input_rate', net_exp.sim_params['input_rate']*np.logspace
...     ('w_input_exc', net_exp.sim_params['w_input_exc']*np.logspace
...
... ]
...
... net_exp.paramRole(sim_list, datapath=datapath_exp)
...
... net_alpha.paramRole(sim_list, datapath=datapath_alpha)
```

CSAConnector: libneurosim support not available in NEST.

Falling back on PyNN's default CSAConnector.

Please re-compile NEST using --with-libneurosim=PATH

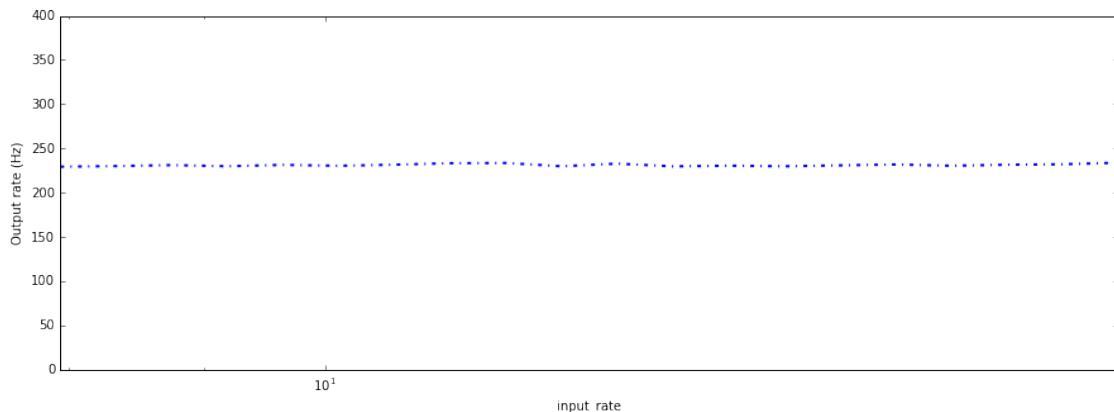
/usr/local/lib/python3.5/site-packages/matplotlib/__init__.py:1350: UserWarning:

because the backend has already been chosen;

matplotlib.use() must be called *before* pylab, matplotlib.pyplot,

or matplotlib.backends is imported for the first time.

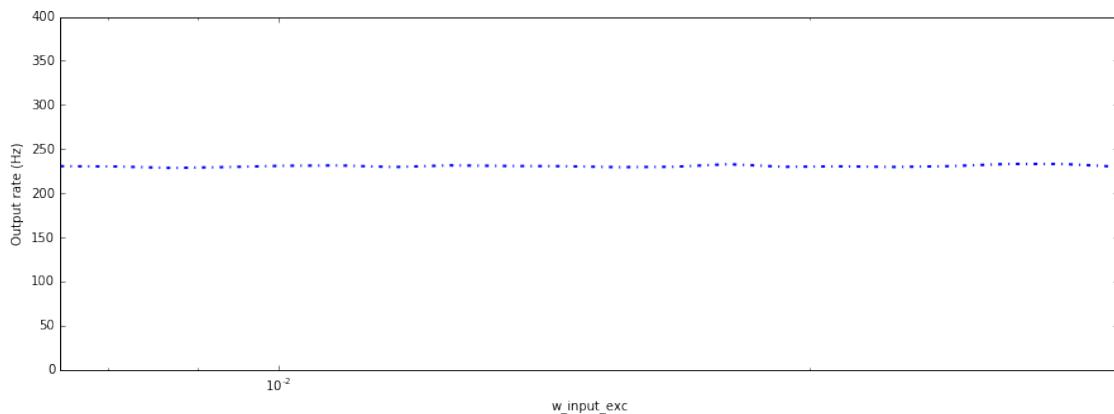
```
----- input_rate -----  
0    7.943282  
1    8.337822  
2    8.751959  
3    9.186665  
4    9.642964  
5    10.121926  
6    10.624678  
7    11.152402  
8    11.706338  
9    12.287787  
10   12.898117  
11   13.538762  
12   14.211227  
13   14.917093  
14   15.658020  
15   16.435748  
16   17.252105  
17   18.109011  
18   19.008479  
19   19.952623  
Name: input_rate, dtype: float64
```



```
----- w_input_exc -----  
0    0.007518  
1    0.008085  
2    0.008695
```

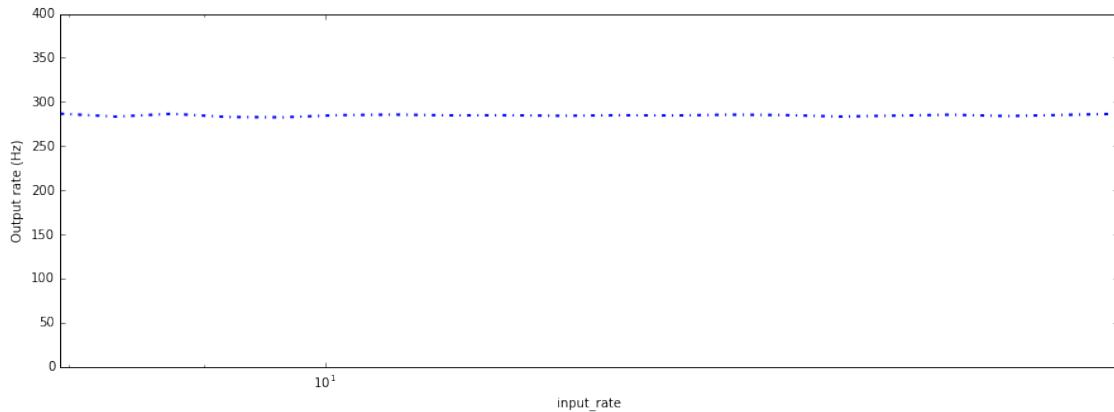
```
3    0.009350
4    0.010056
5    0.010814
6    0.011630
7    0.012507
8    0.013450
9    0.014464
10   0.015555
11   0.016729
12   0.017990
13   0.019347
14   0.020806
15   0.022376
16   0.024063
17   0.025878
18   0.027830
19   0.029929
```

Name: w_input_exc, dtype: float64



```
----- input_rate -----
0    7.943282
1    8.337822
2    8.751959
3    9.186665
4    9.642964
5    10.121926
6    10.624678
7    11.152402
8    11.706338
```

```
9      12.287787
10     12.898117
11     13.538762
12     14.211227
13     14.917093
14     15.658020
15     16.435748
16     17.252105
17     18.109011
18     19.008479
19     19.952623
Name: input_rate, dtype: float64
```

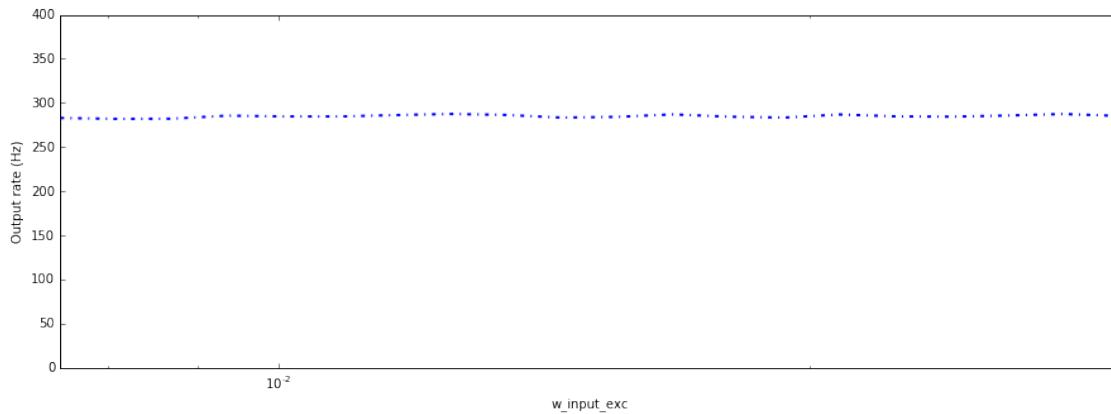


```
----- w_input_exc -----
0      0.007518
1      0.008085
2      0.008695
3      0.009350
4      0.010056
5      0.010814
6      0.011630
7      0.012507
8      0.013450
9      0.014464
10     0.015555
11     0.016729
12     0.017990
13     0.019347
14     0.020806
```

```

15    0.022376
16    0.024063
17    0.025878
18    0.027830
19    0.029929
Name: w_input_exc, dtype: float64

```



2.2.3 Rôle du taux de décharge de l'entrée

L'activité neuronale de la population source est définie par un processus de Poisson homogène. L'activité de la population source est l'énergie apportée au RRNN. Il est alors important d'observer l'effet d'une manipulation de cette activité sur le comportement du réseau.

Pour chaque valeur de taux de décharge de la population source une simulation du RRNN est exécutée et des rasterplots des populations source, exciatrice (E) et inhibitrice (I) sont affichés.

Il est remarqué qu'une augmentation suffisante de l'activité de la population source induit une augmentation d'activité dans les populations E et I. Cette discontinuité suggère l'existence d'un filtre entre la population source et les deux autres.

2.2.4 Rôle du poids de l'entrée

Les rasterplots générés sur une manipulation de l'activité de la population source montrent l'existence de filtres entre l'activité de la source et l'activité du réseau. Le poids de l'entrée est un de ces filtres. Ce paramètre détermine les valeurs des poids des connexions synaptiques entre les neurones de la source et les neurones de la population E. Ici, son effet sur le comportement du réseau est étudié.

Pour cela, un rasterplot des trois populations est généré par valeur du poids de l'entrée

Il s'avère que l'augmentation du poids de l'entrée induit une augmentation de l'activité du réseau.

2.2.5 Courbes de taux de décharge en fonction de l'activité et du poids de l'entrée

Les rasterplots permet d'avoir un aperçu qualitatif de l'activité d'une population neuronale. Cependant, il est nécessaire de pouvoir observer de façon quantitative cette activité. Ici, les effets de la manipulation de l'activité de la population source ainsi que du poids de celle-ci sur l'activité du réseau sont réexaminés.

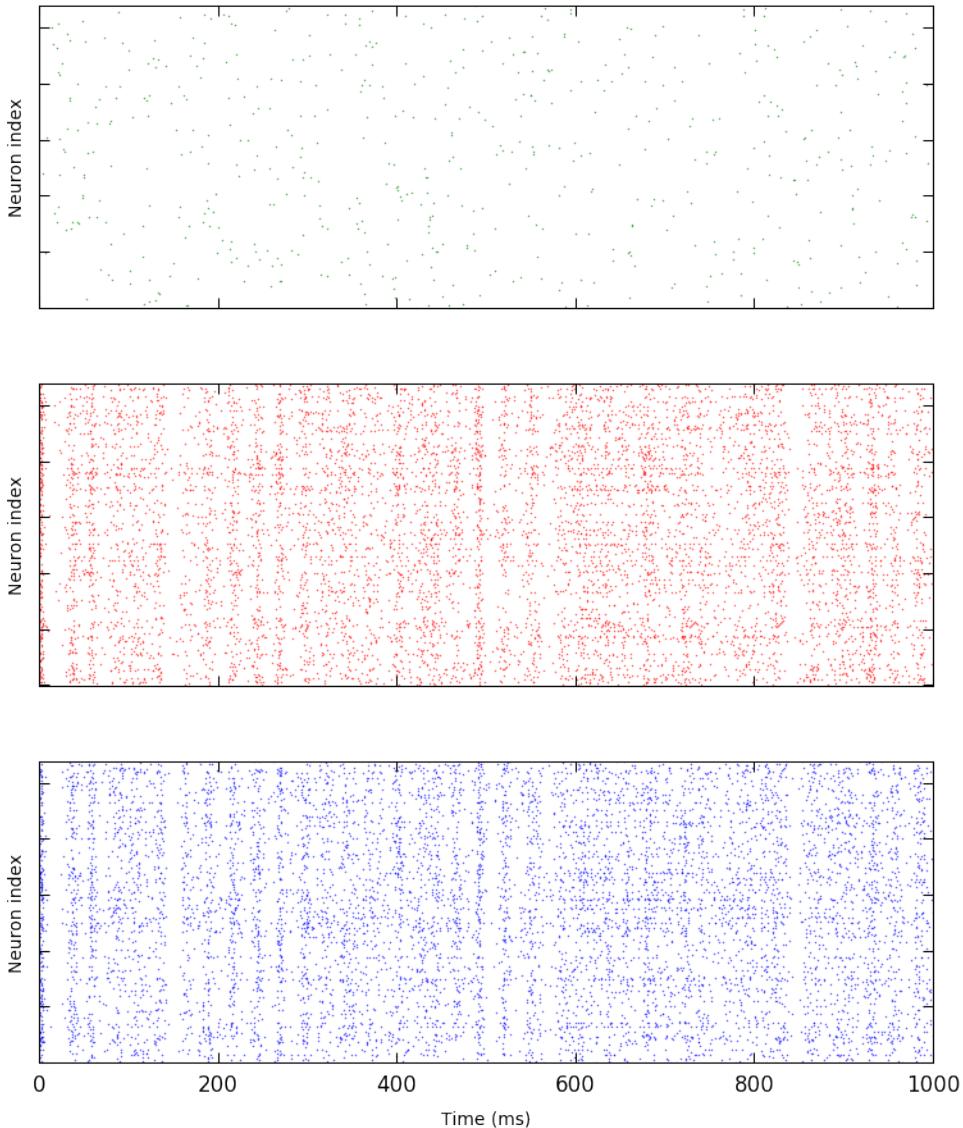
Pour chacun des deux paramètres, une simulation du modèle est exécutée par valeur prise par le paramètre étudié et le taux de décharge neuronal moyen des populations E et I est récupéré. Chaque point d'une courbe étant défini par le couple (valeur du paramètre, taux de décharge), une courbe représente donc la variation de taux de décharge en fonction de la manipulation d'un paramètre.

Il s'avère que le taux de décharge augmente bien quand l'activité de la population source ou le poids de l'entrée augmentent.

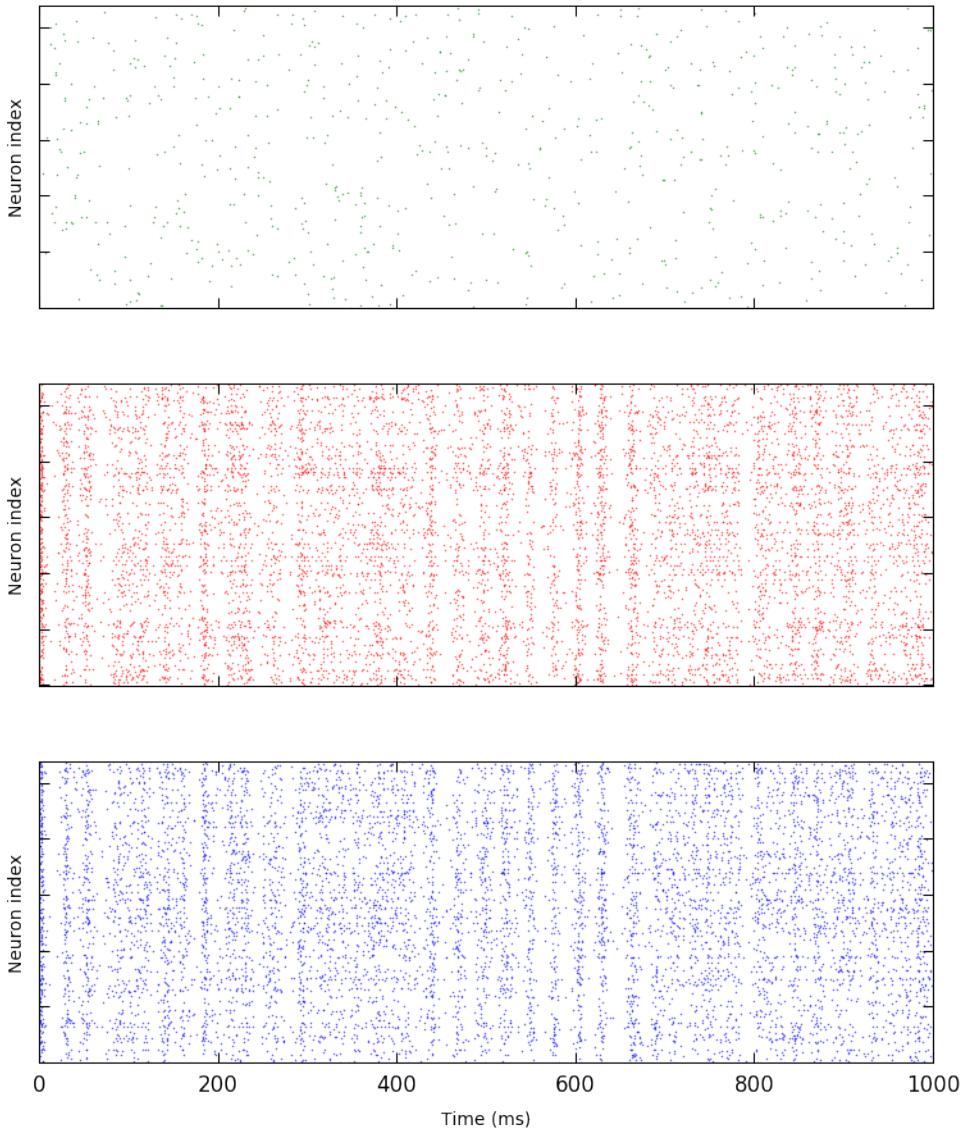
```
>>> from RRNN import RRNN
... import numpy as np
...
...
...
... n_sim_each, time = 5, 100
... n_sim_each, time = 25, 1000
...
... net = RRNN(time=time)
... _ = net.variationRaster('input_rate', net.sim_params['input_rate'] * np
... #_ = net.variationRaster('input_rate', net.sim_params['input_rate'] * np

CSAConnector: libneurosim support not available in NEST.
Falling back on PyNN's default CSAConnector.
Please re-compile NEST using --with-libneurosim=PATH
/usr/local/lib/python3.5/site-packages/matplotlib/__init__.py:1350: UserWarning:
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.
```

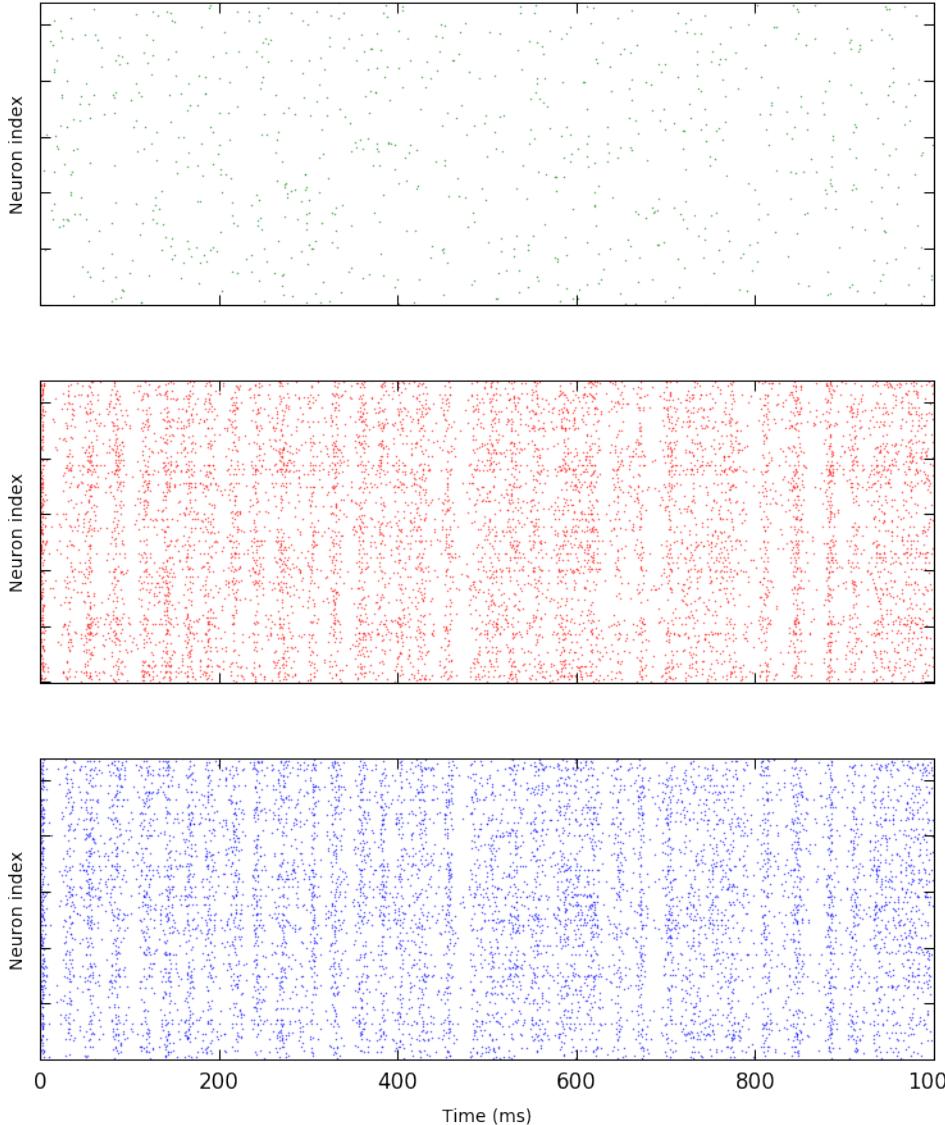
----- input_rate = 1.0 -----



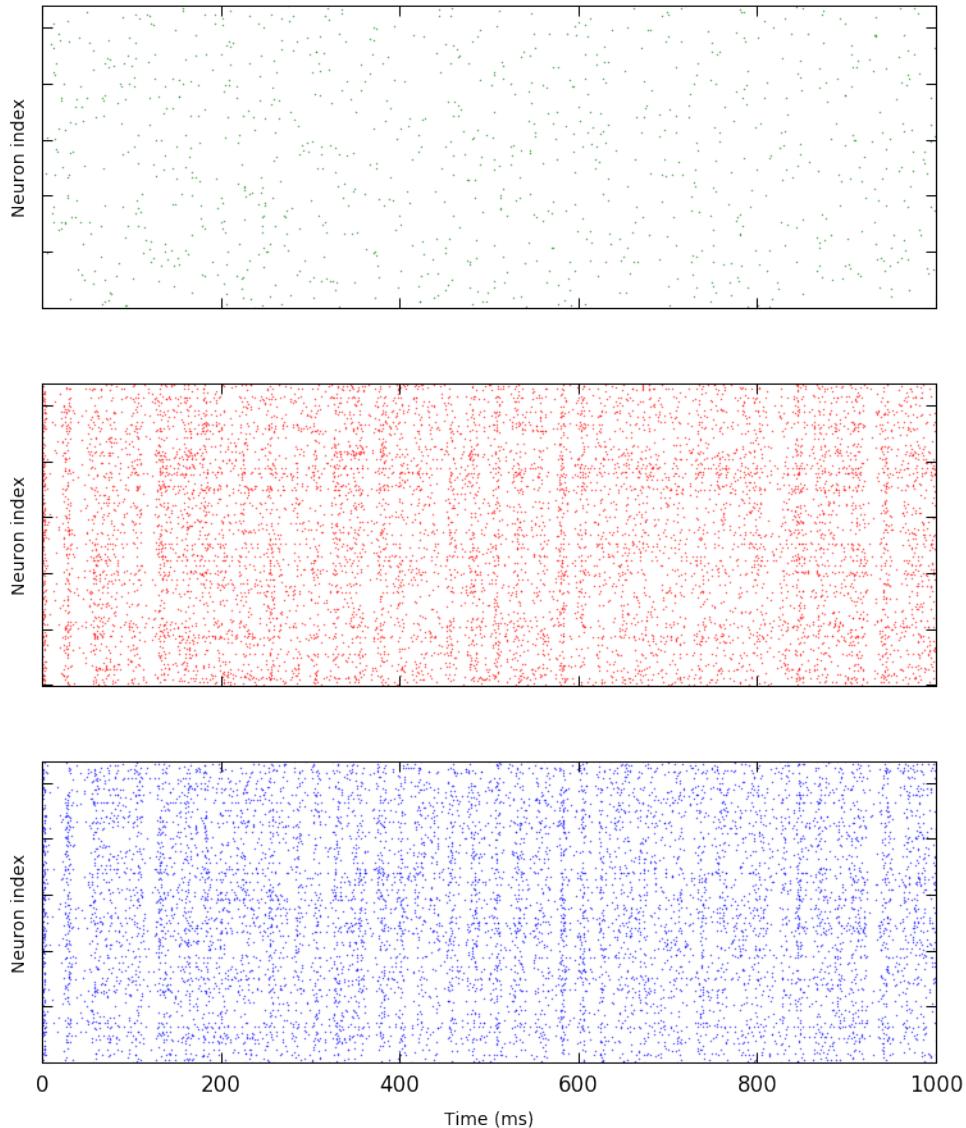
----- input_rate = 1.2115276586285886 -----



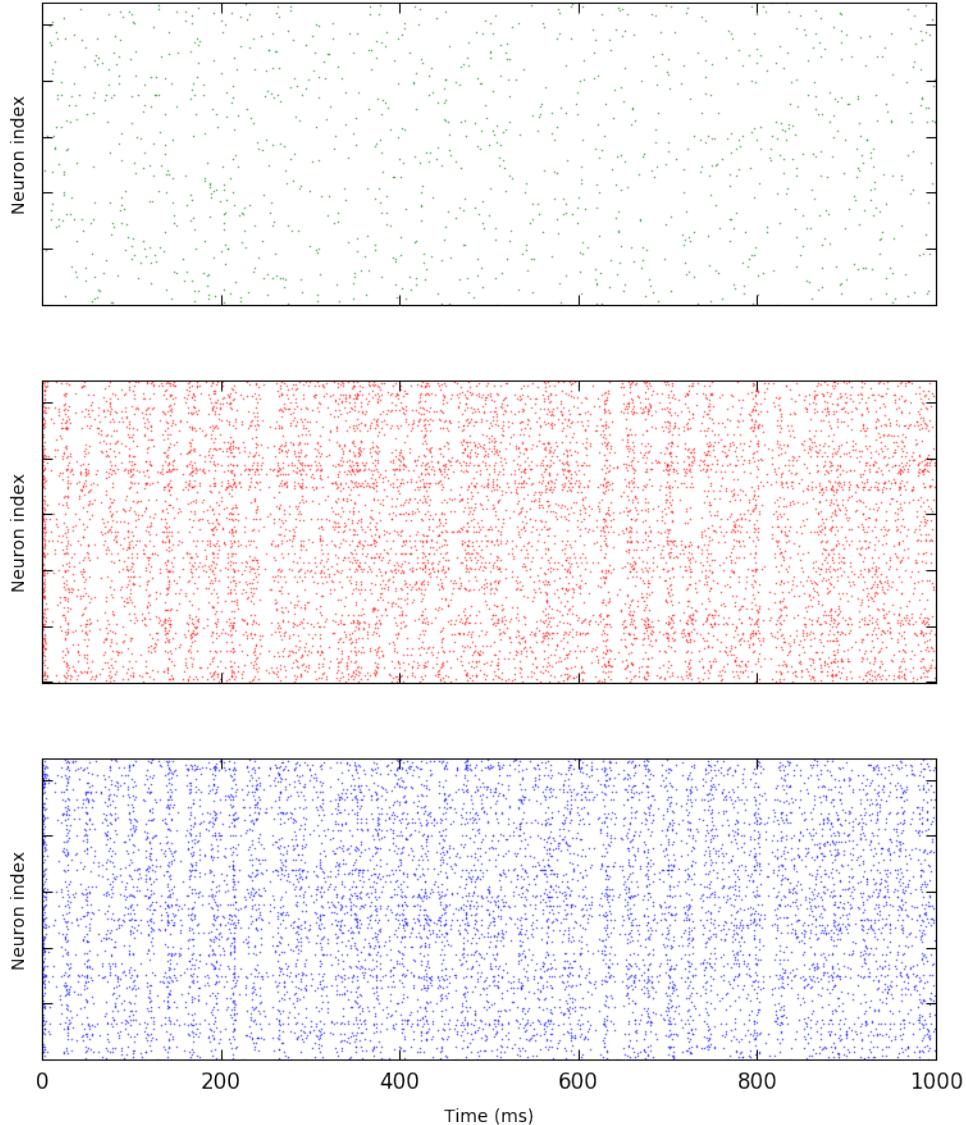
----- input_rate = 1.4677992676220695 -----



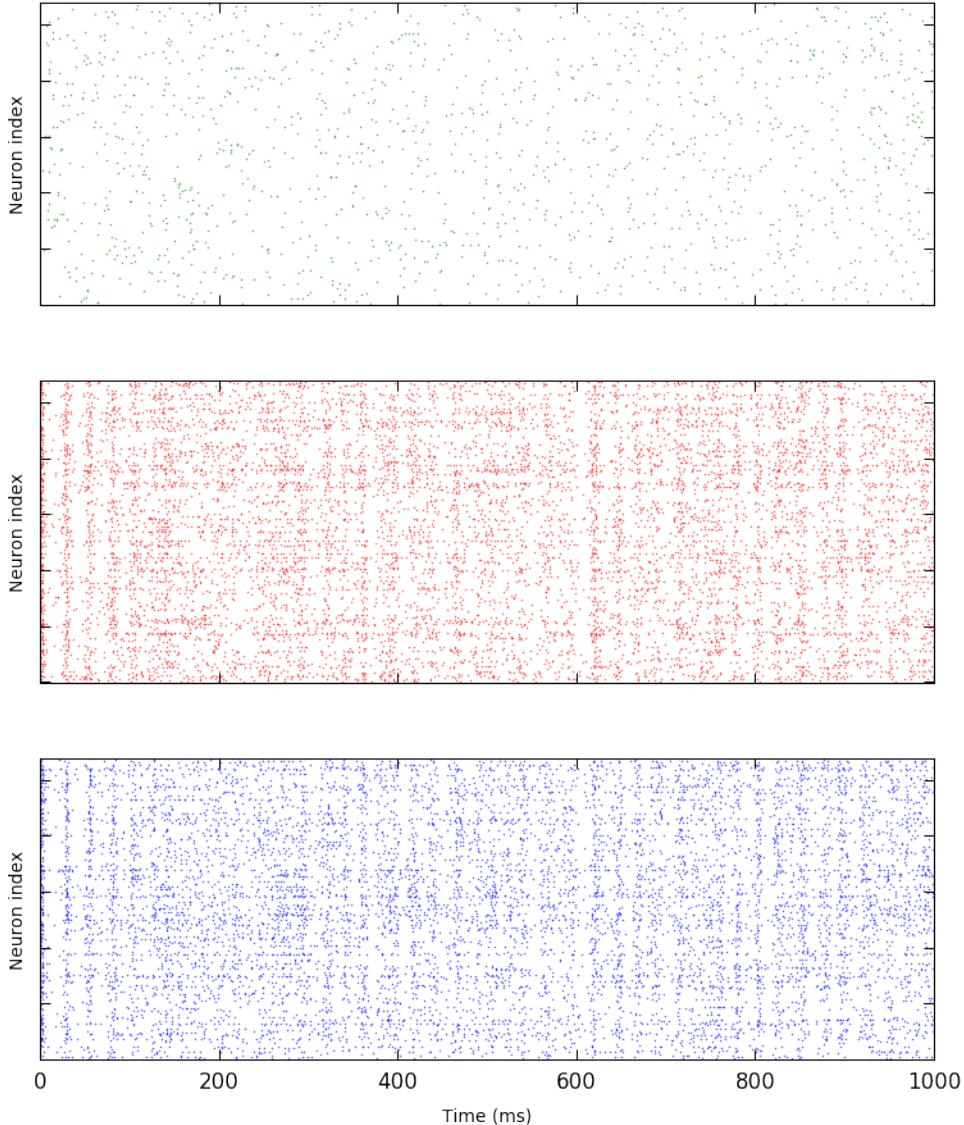
----- input_rate = 1.778279410038923 -----



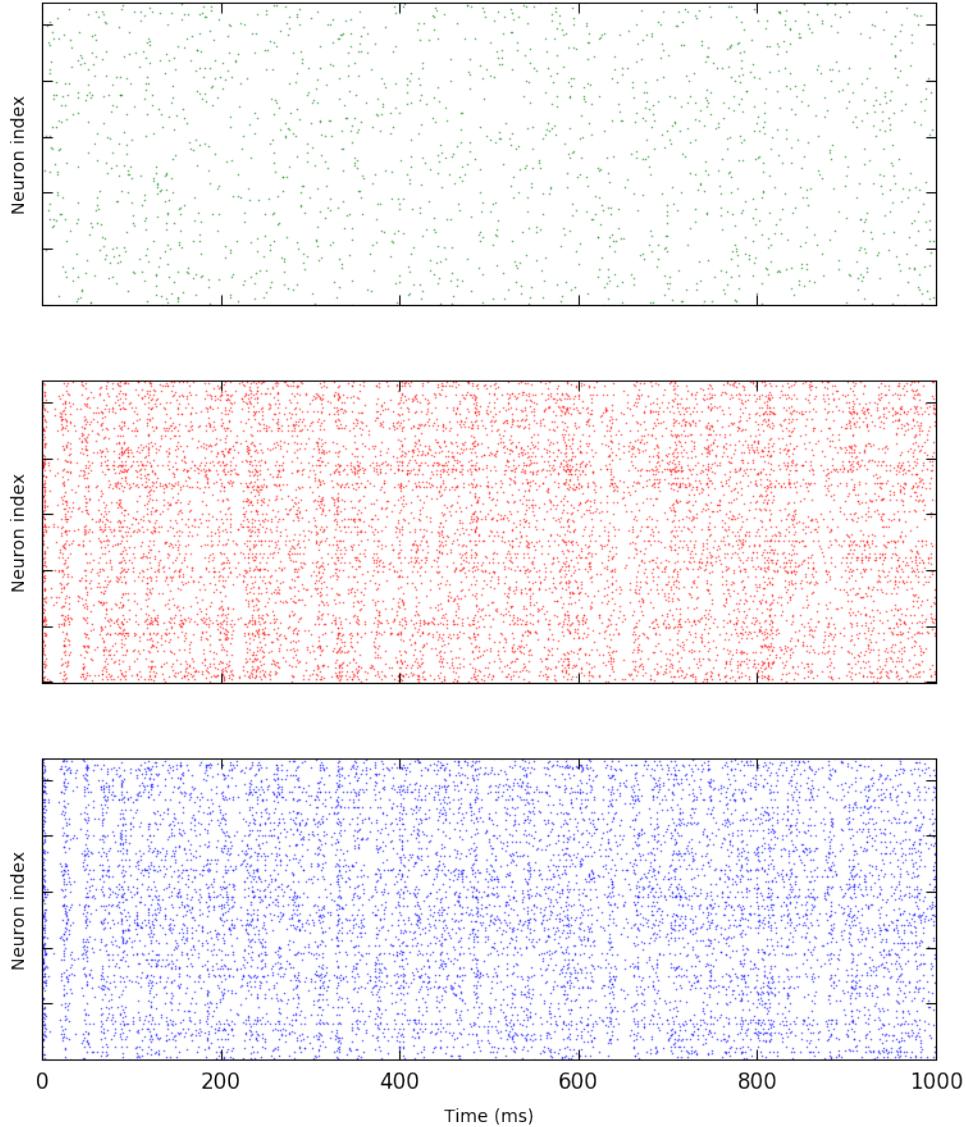
----- input_rate = 2.1544346900318834 -----



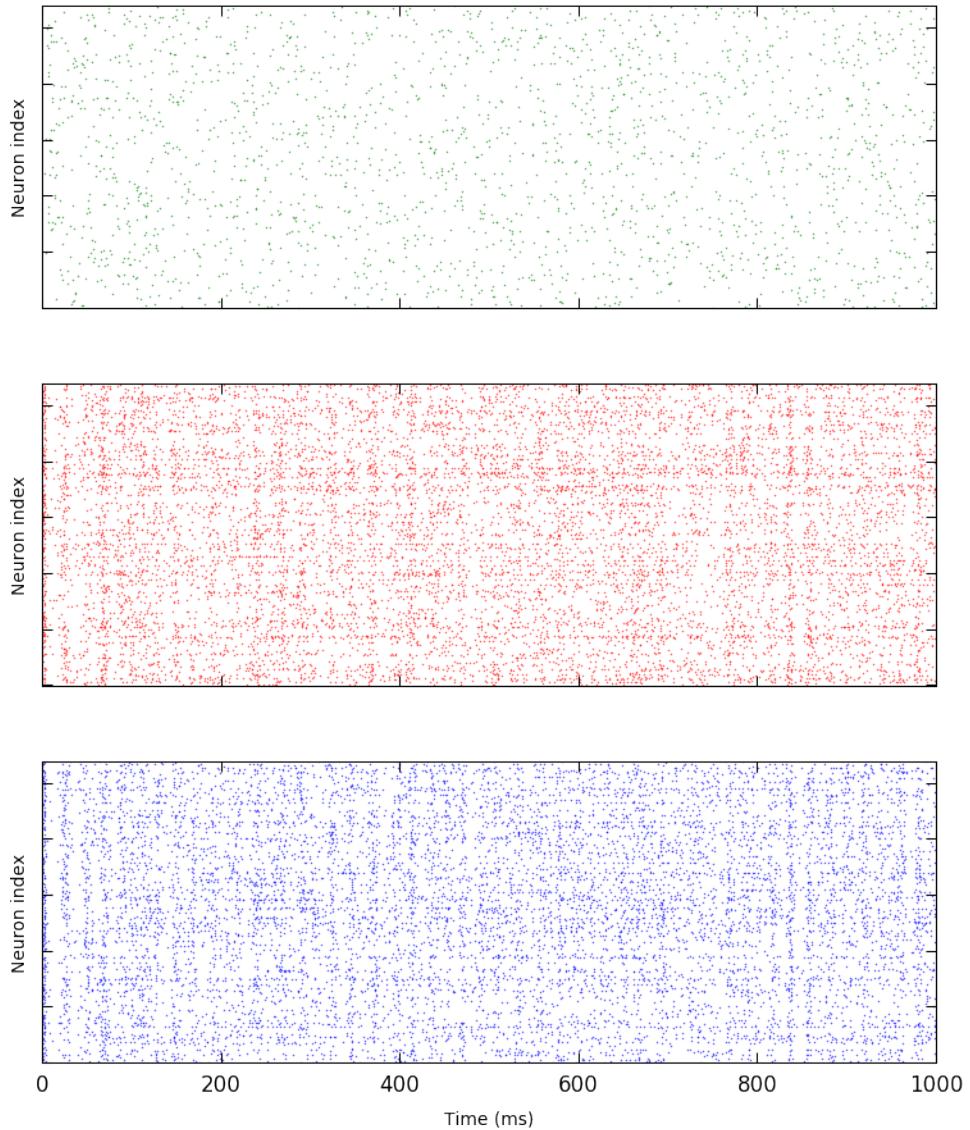
----- input_rate = 2.6101572156825363 -----



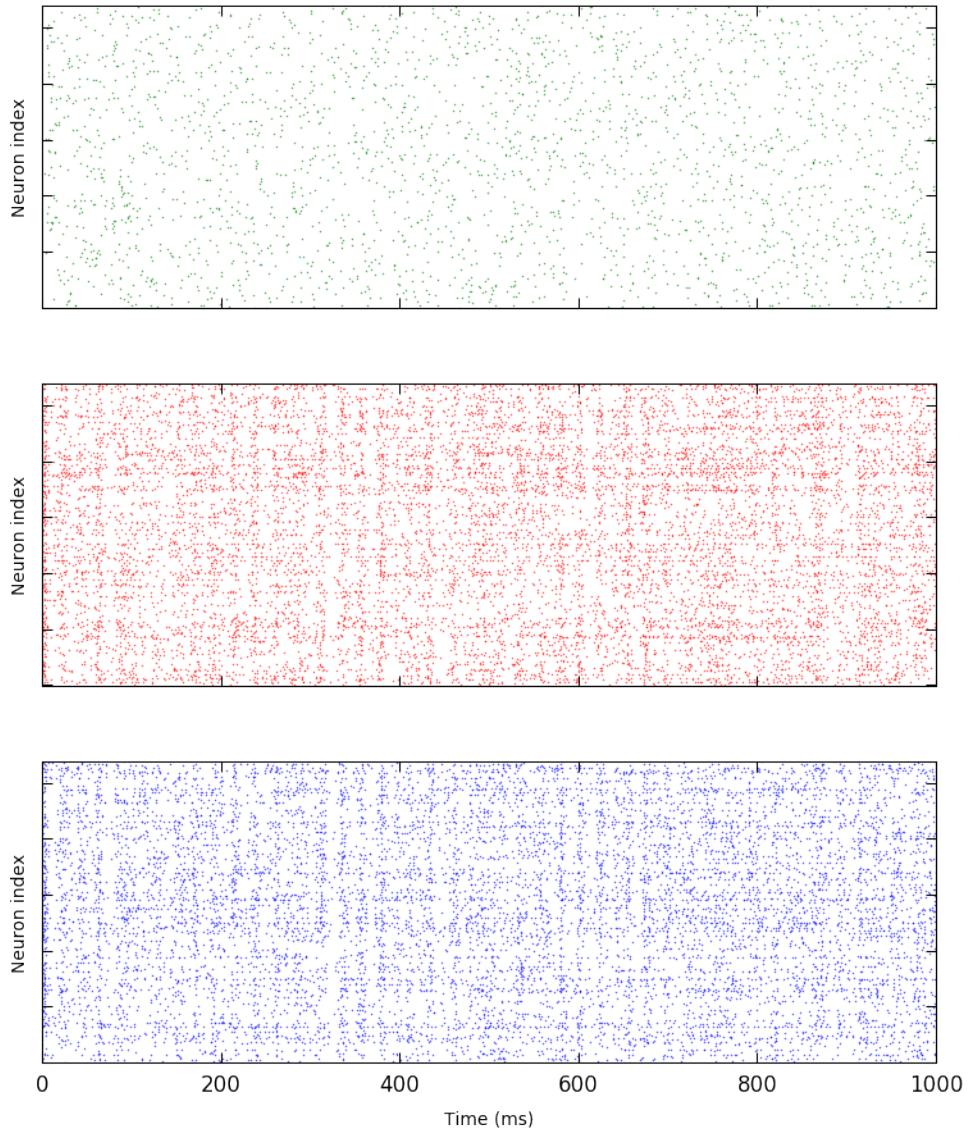
----- input_rate = 3.1622776601683795 -----



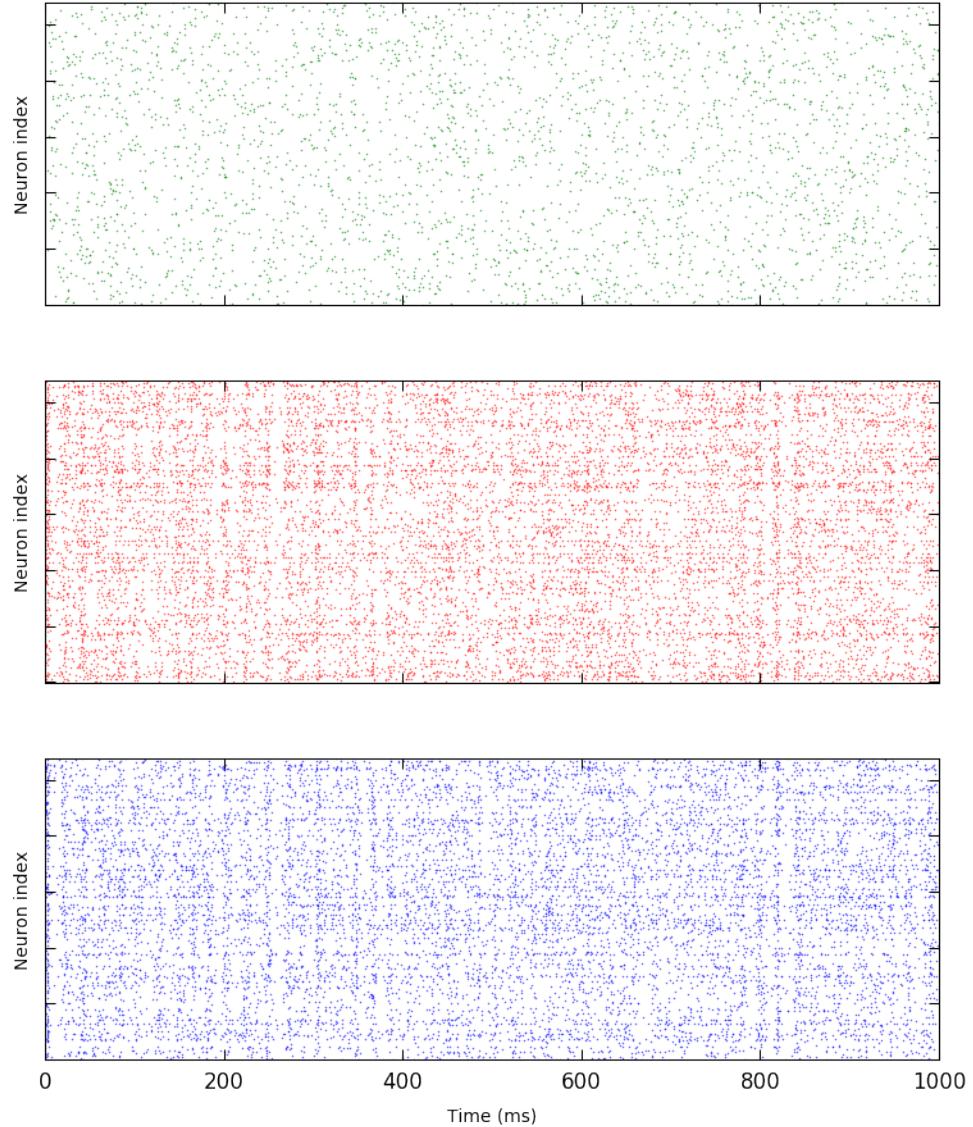
----- input_rate = 3.831186849557287 -----



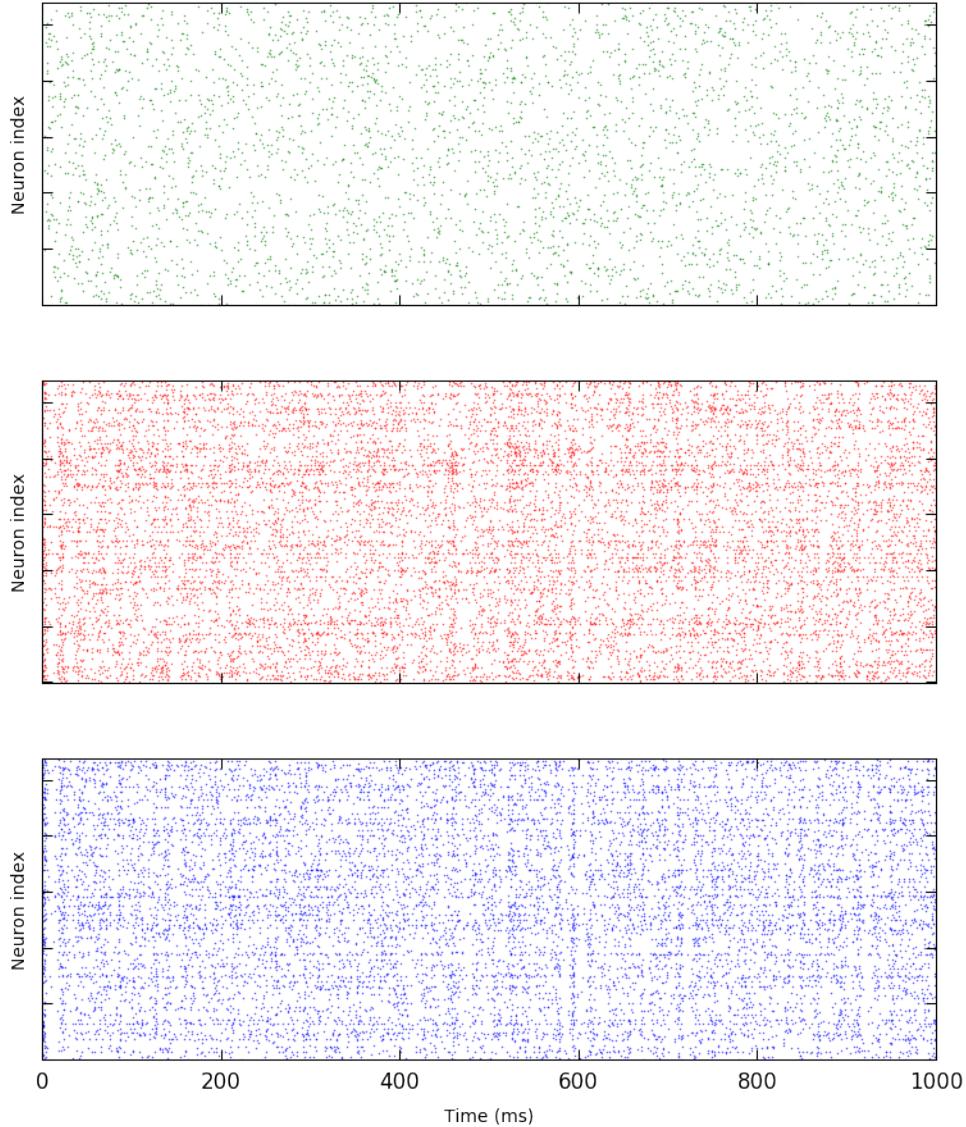
----- input_rate = 4.641588833612778 -----



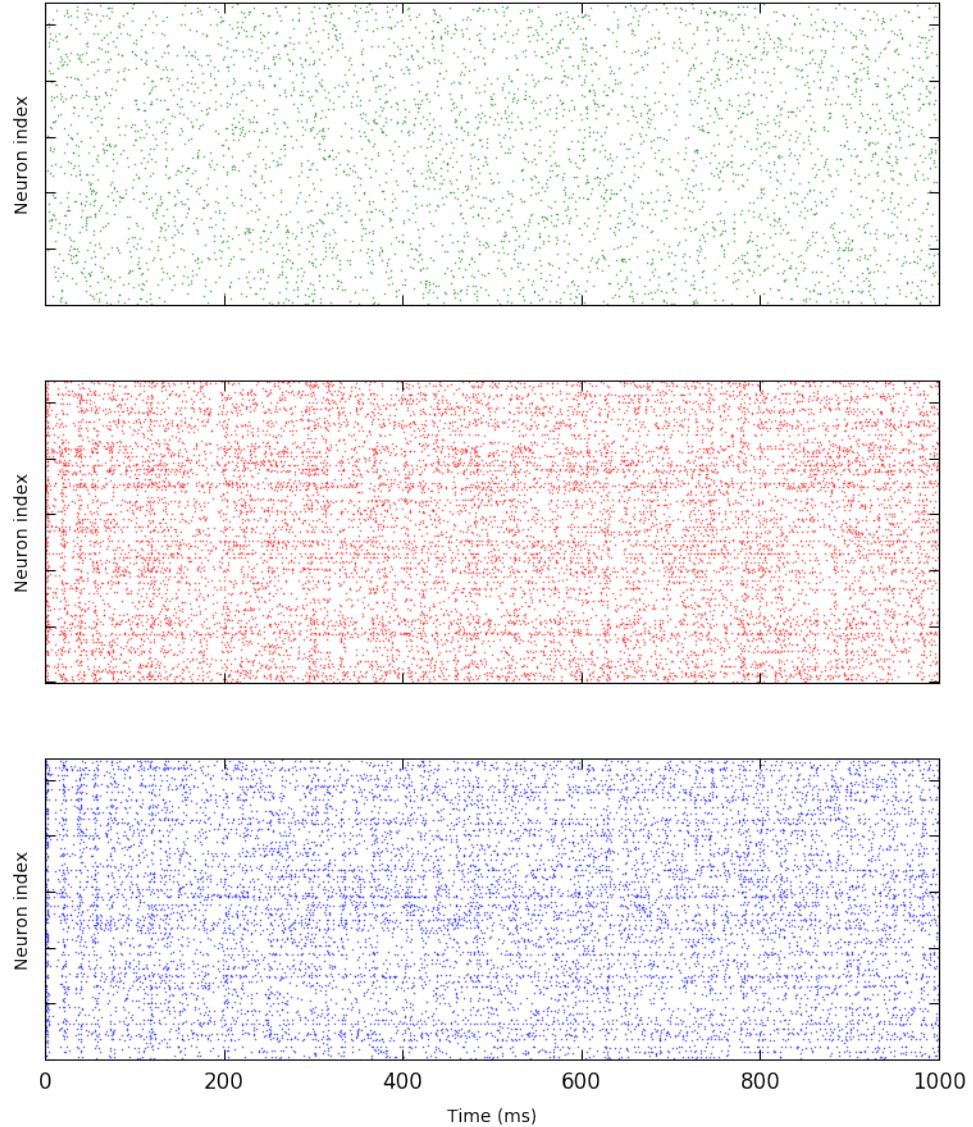
----- input_rate = 5.623413251903491 -----



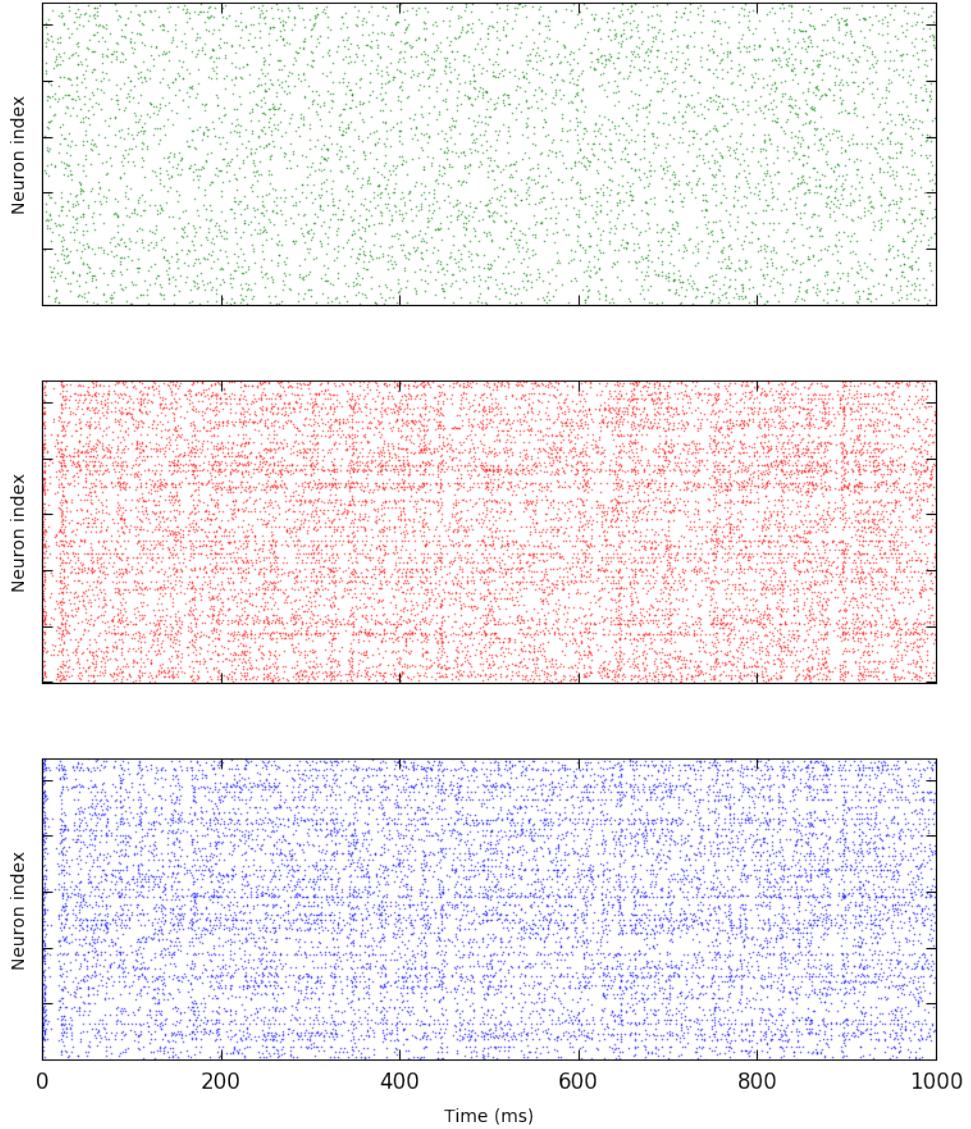
----- input_rate = 6.812920690579611 -----



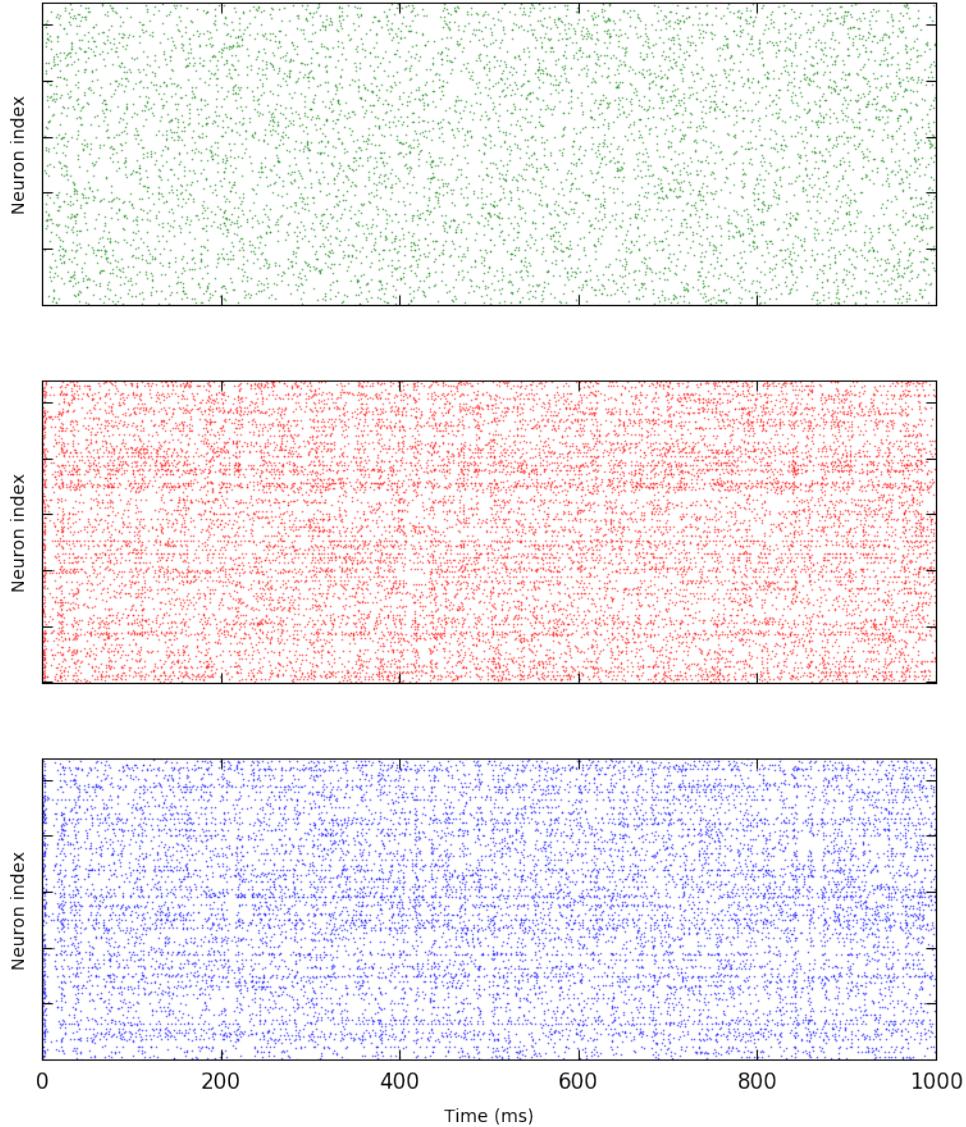
----- input_rate = 8.254041852680183 -----



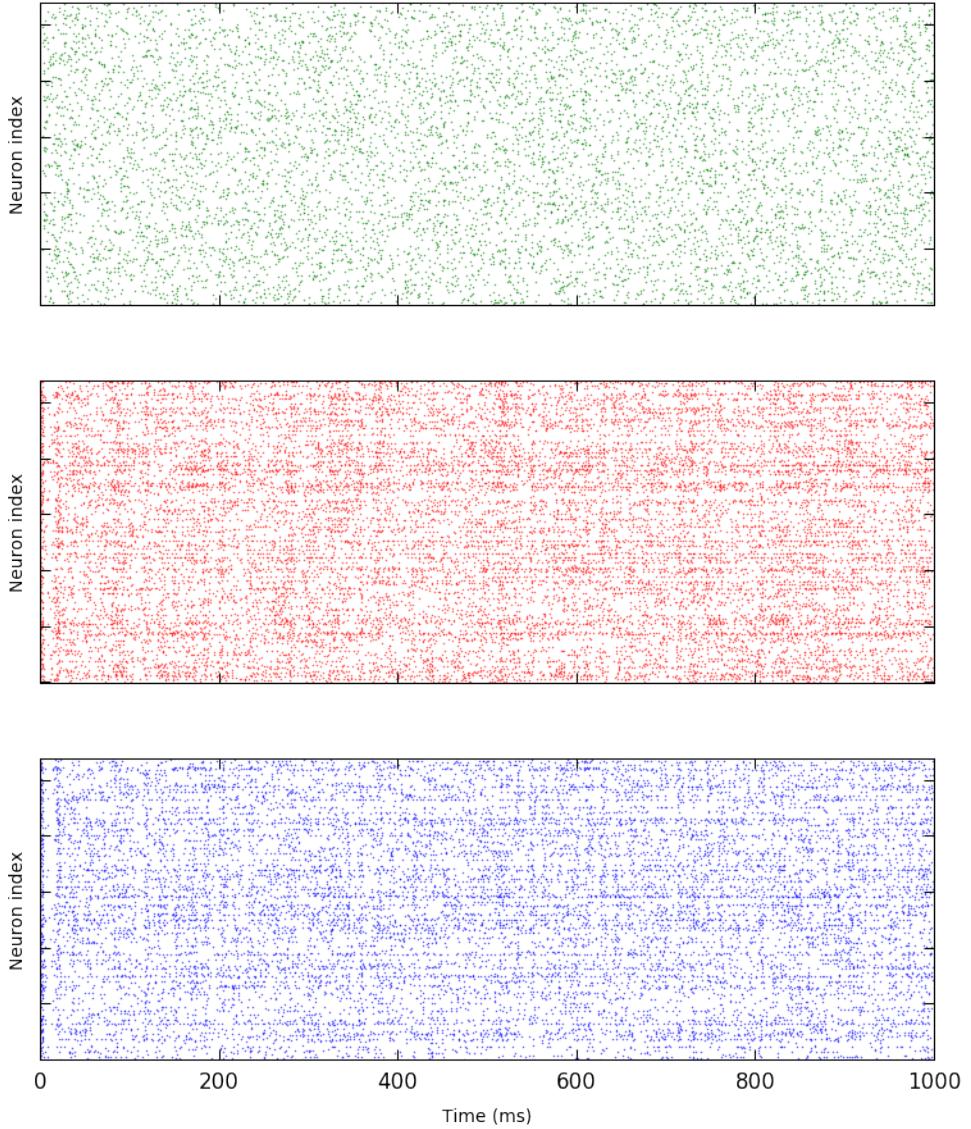
----- input_rate = 10.0 -----



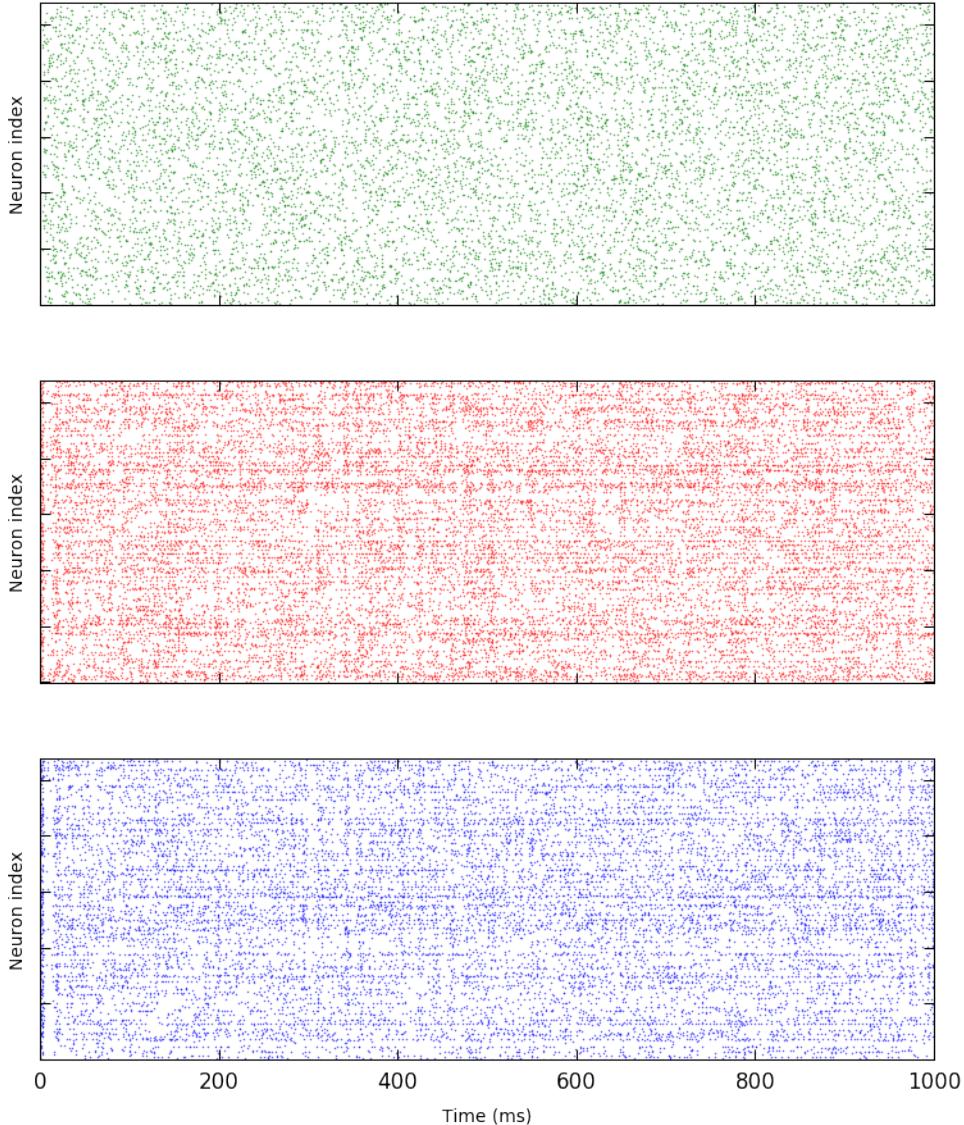
----- input_rate = 12.115276586285882 -----



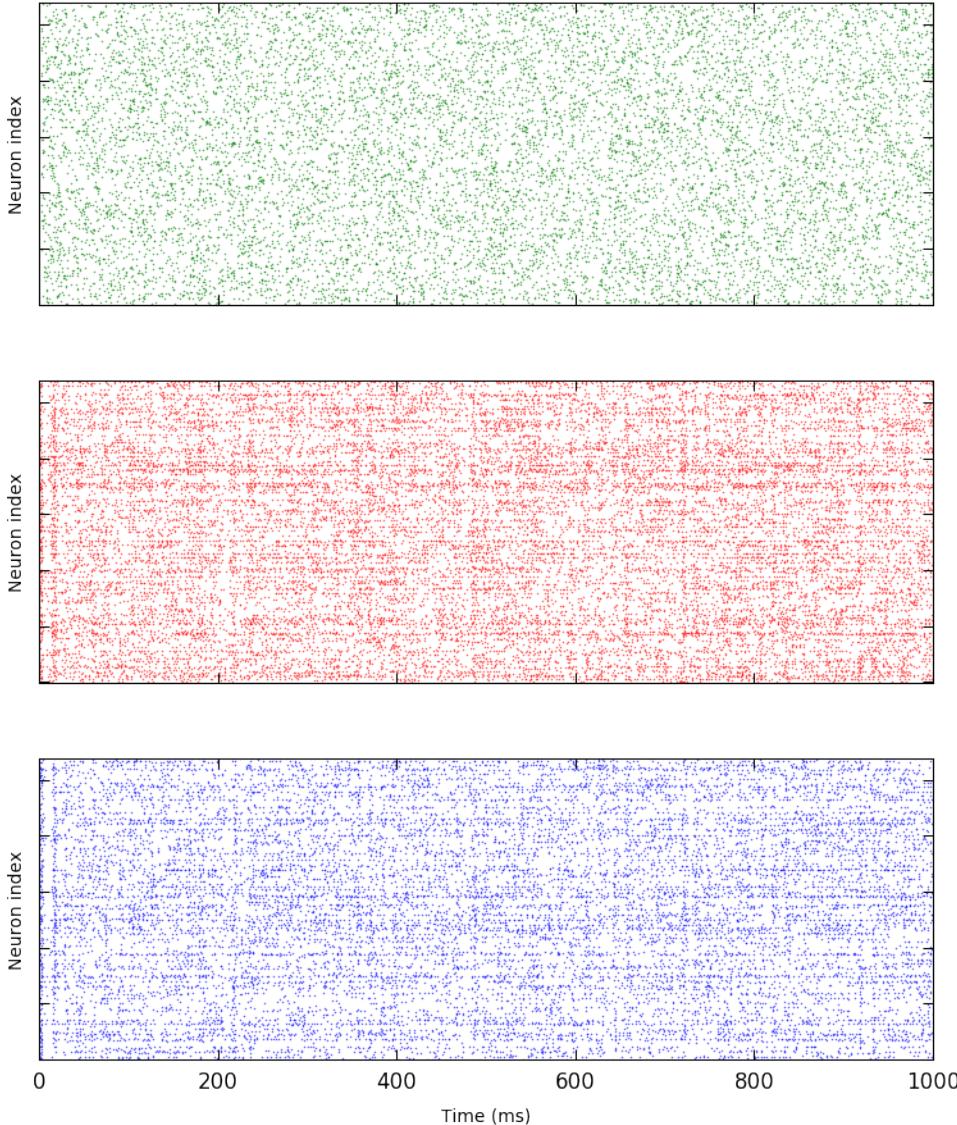
----- input_rate = 14.67799267622069 -----



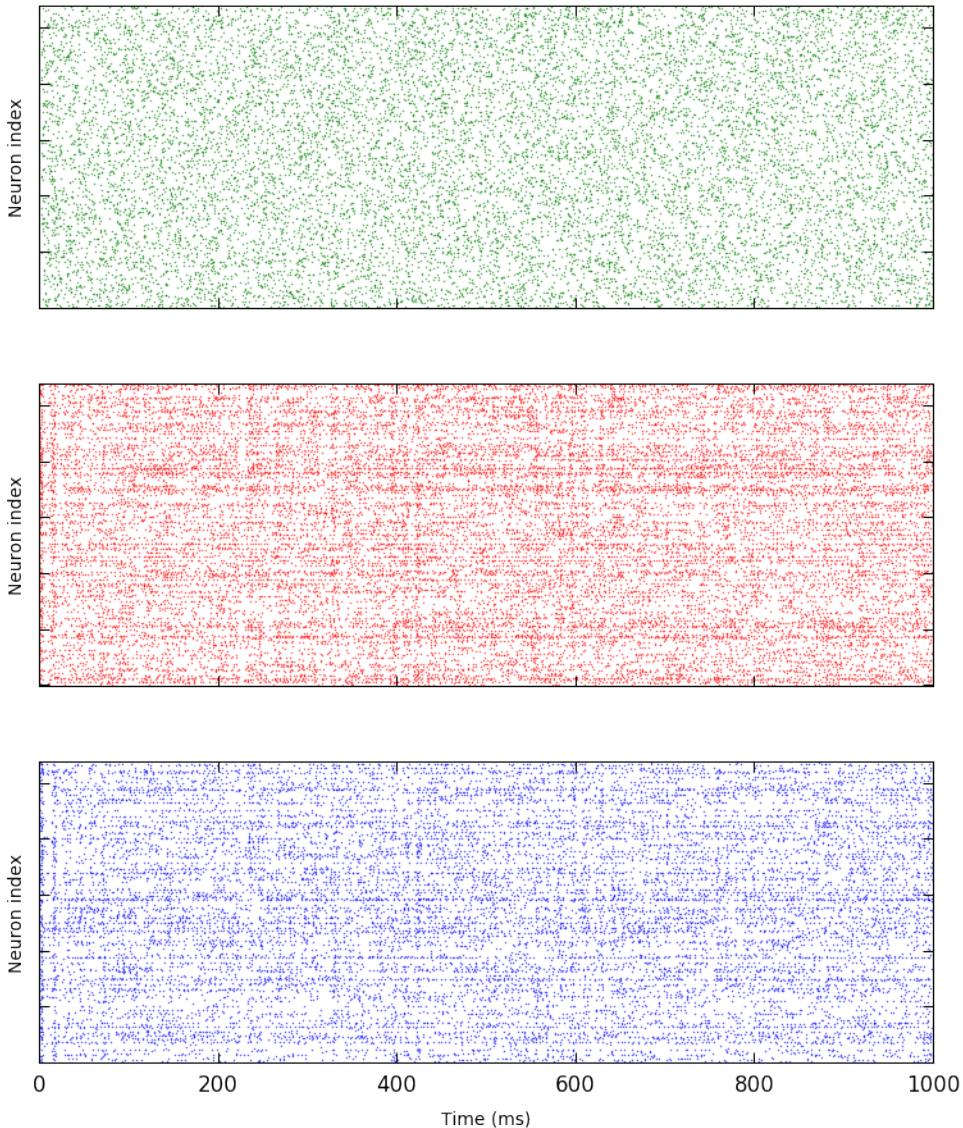
----- input_rate = 17.78279410038923 -----



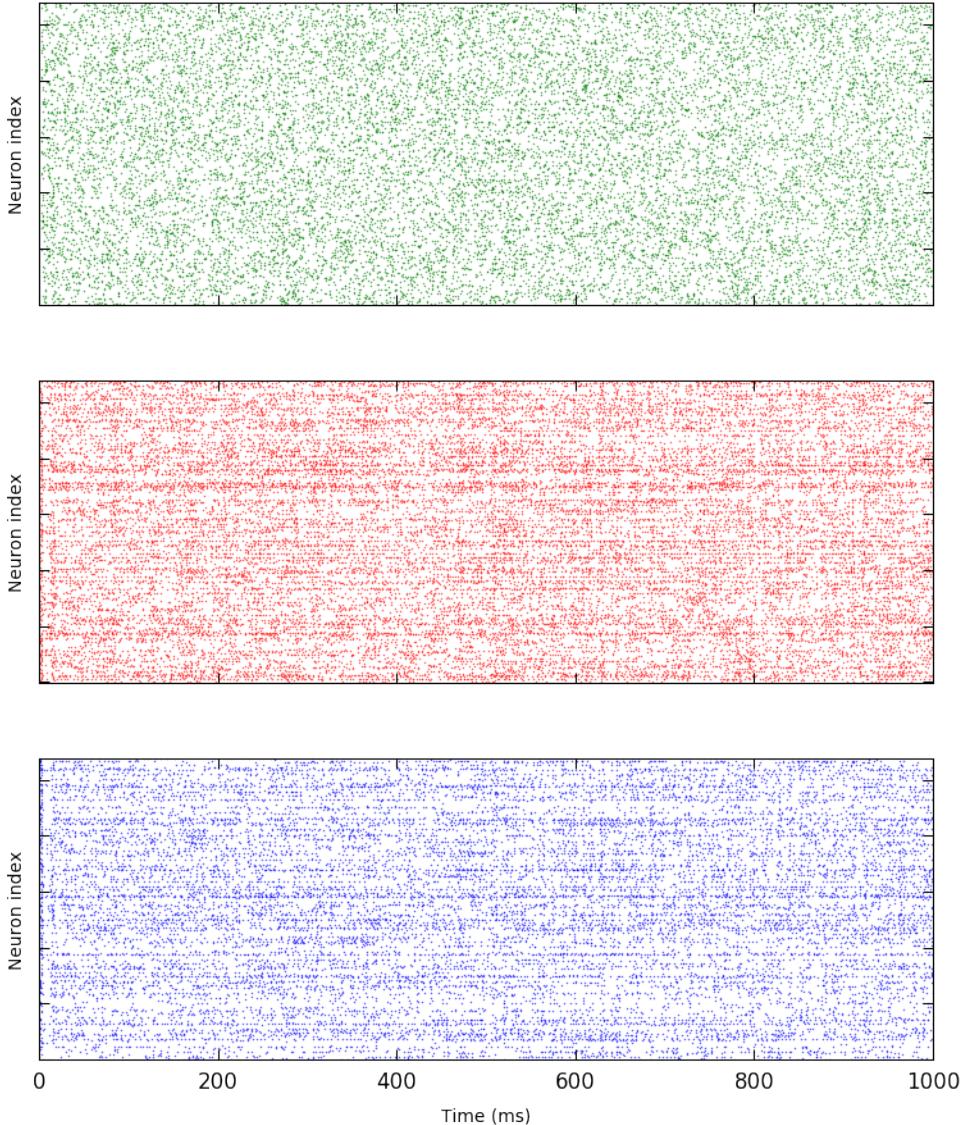
----- input_rate = 21.544346900318835 -----



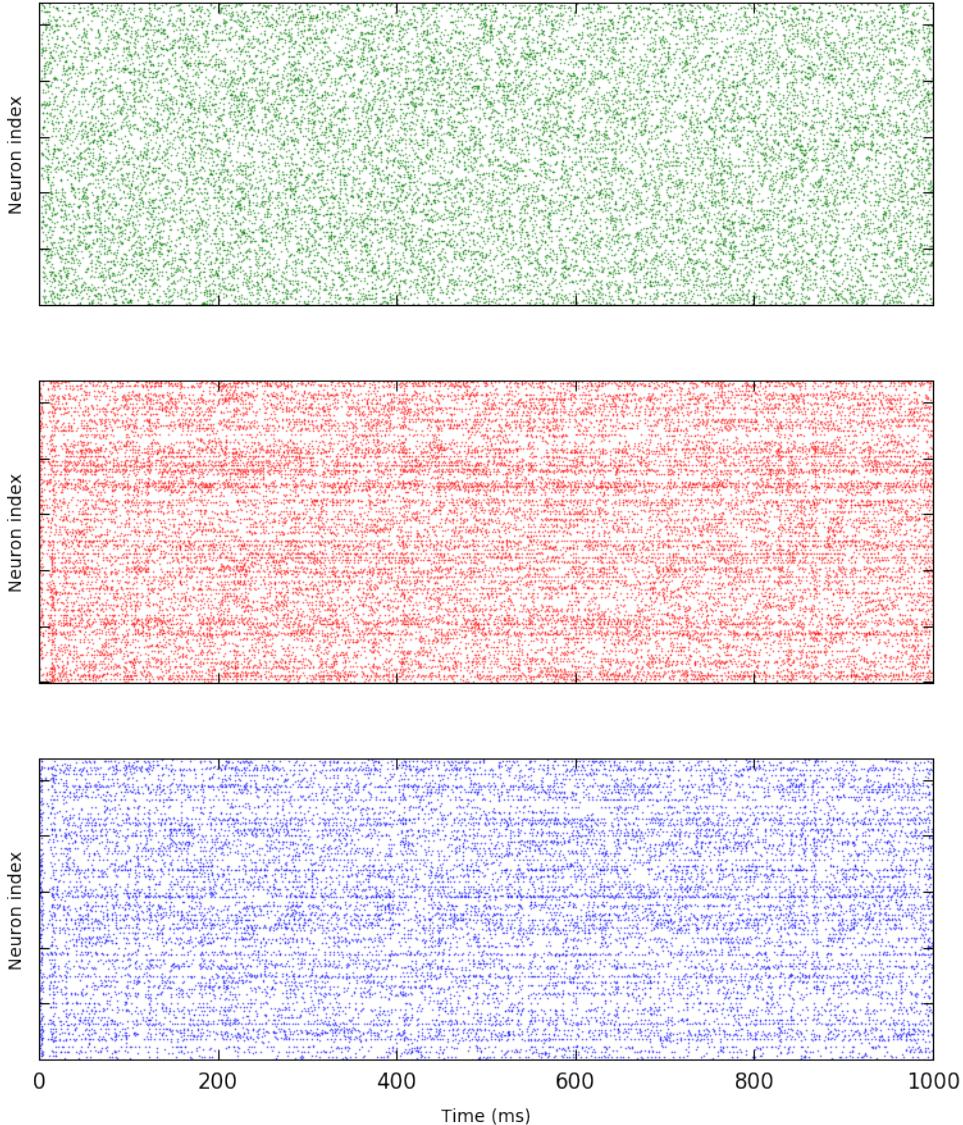
----- input_rate = 26.10157215682536 -----



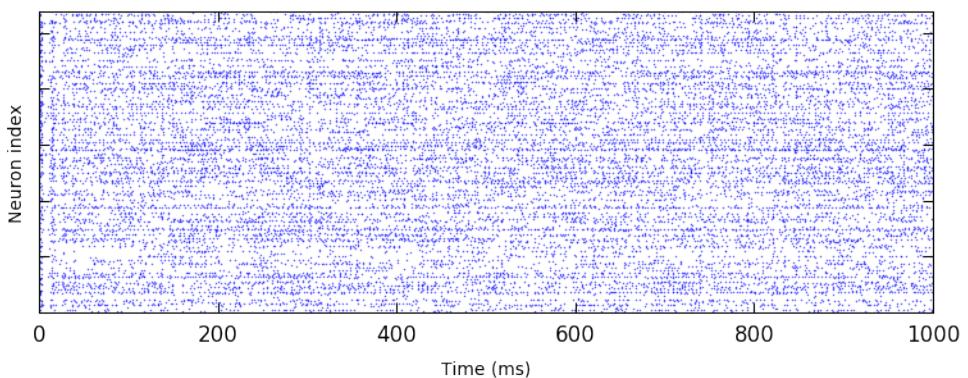
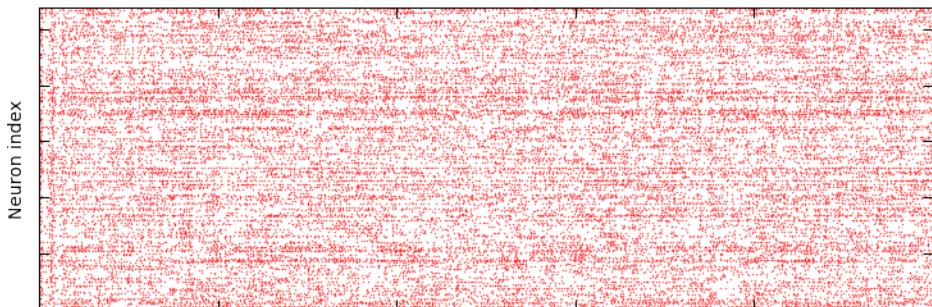
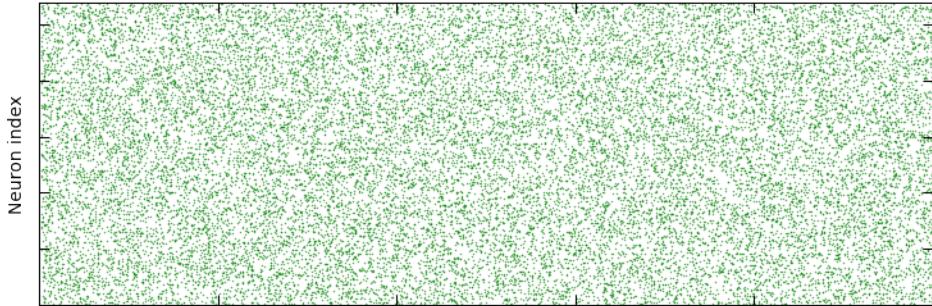
----- input_rate = 31.622776601683796 -----



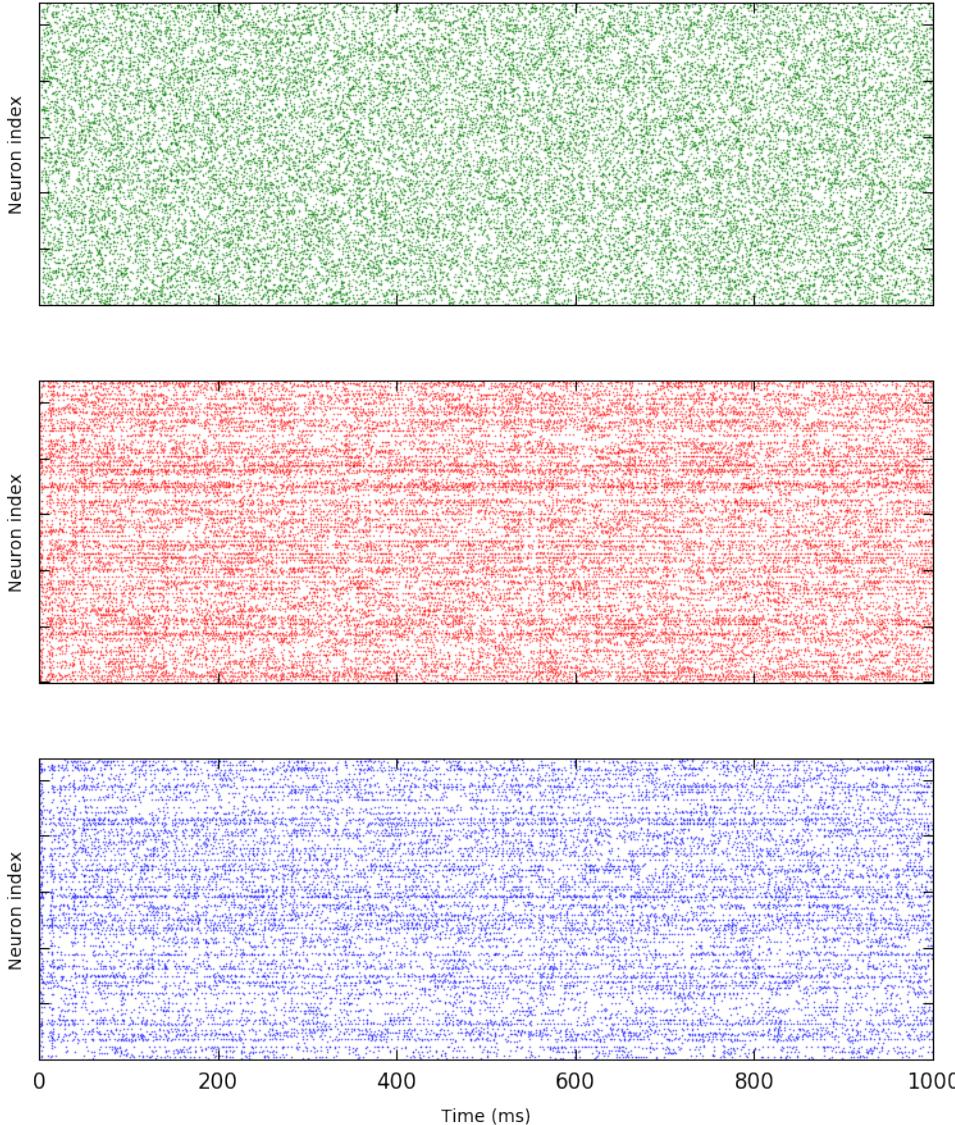
----- input_rate = 38.31186849557287 -----



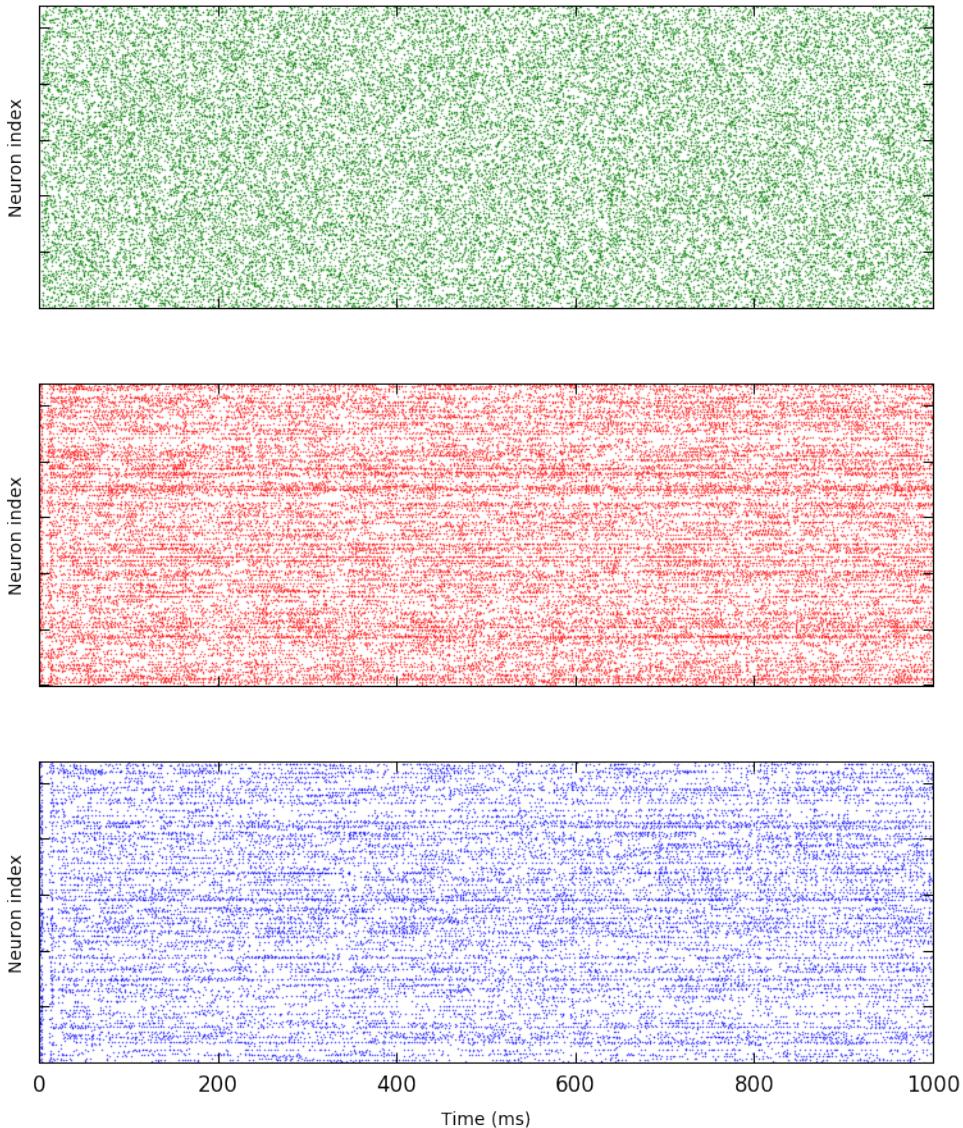
----- input_rate = 46.41588833612778 -----



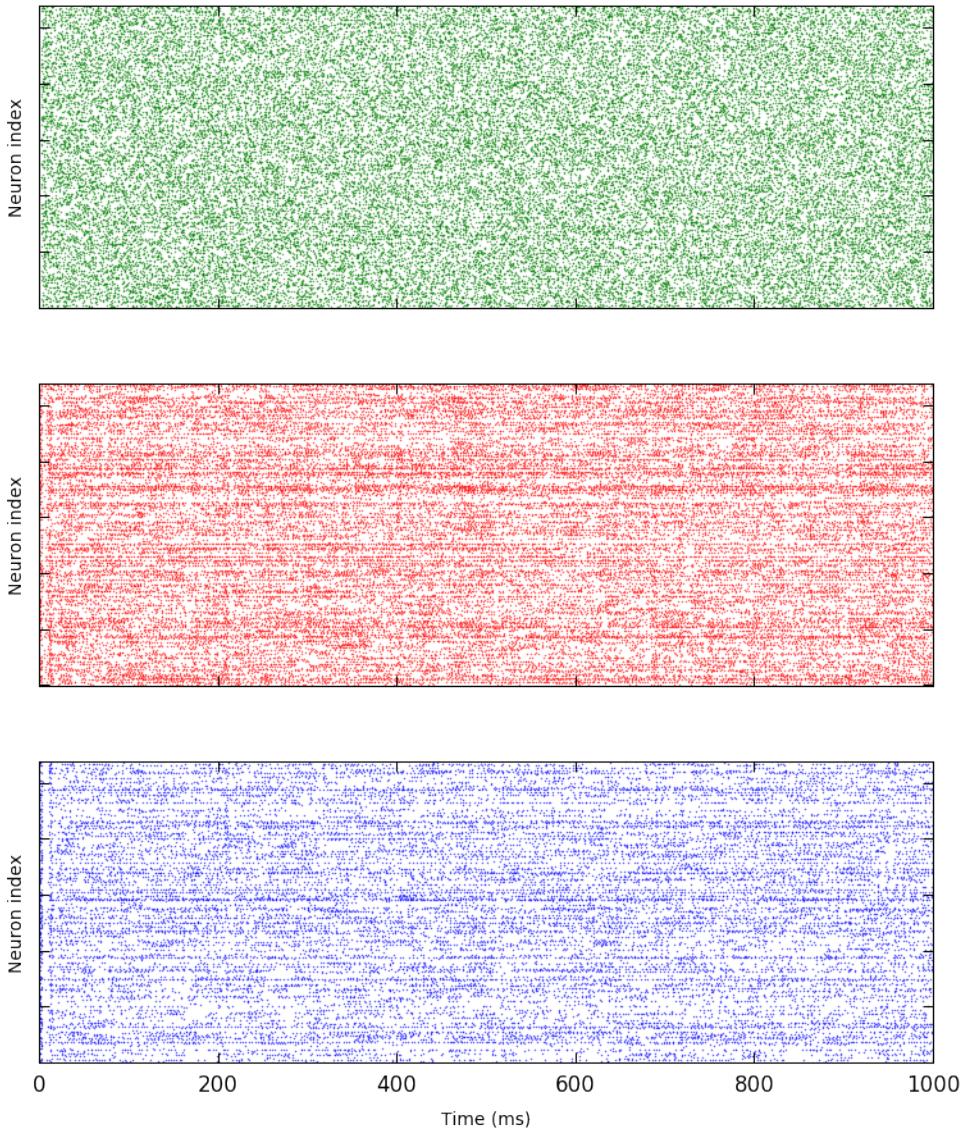
----- input_rate = 56.234132519034915 -----



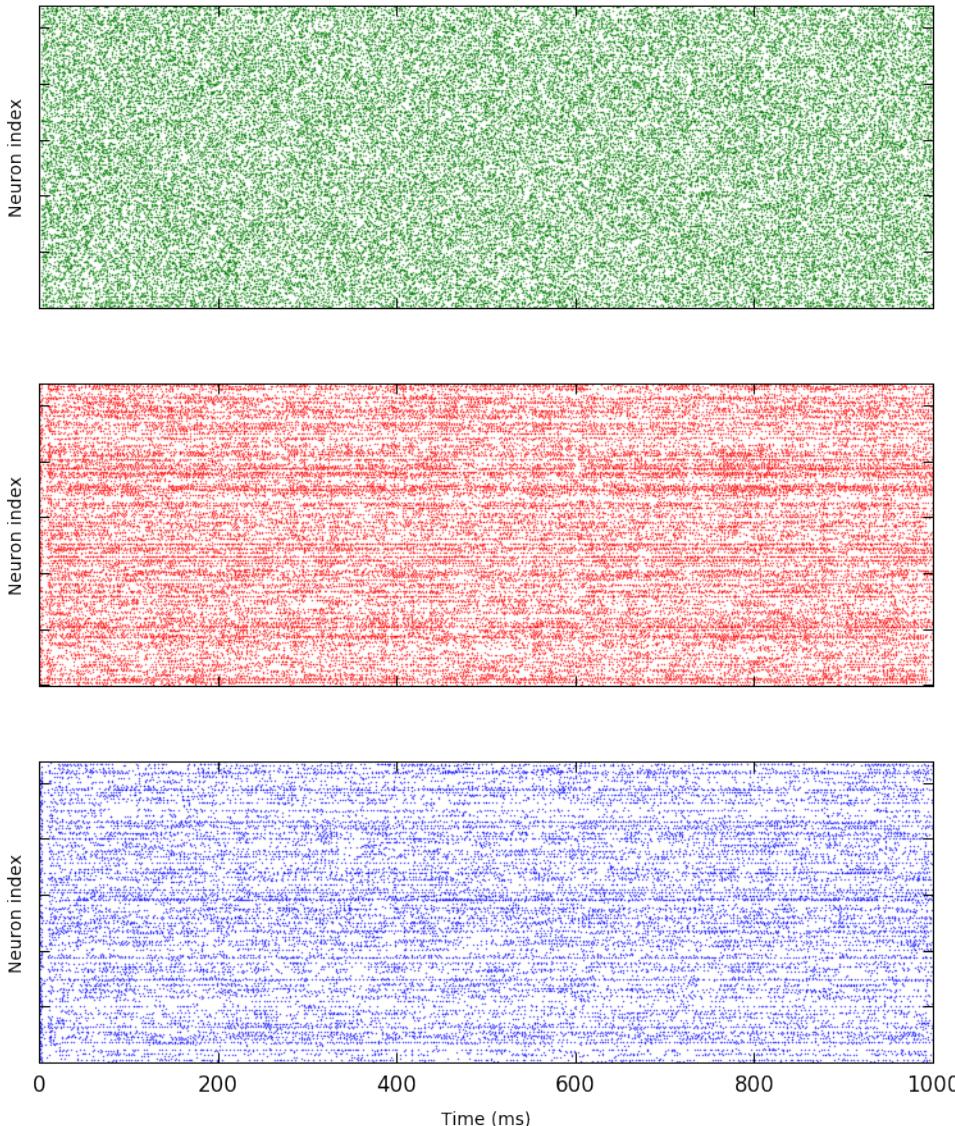
----- input_rate = 68.12920690579611 -----



----- input_rate = 82.54041852680182 -----

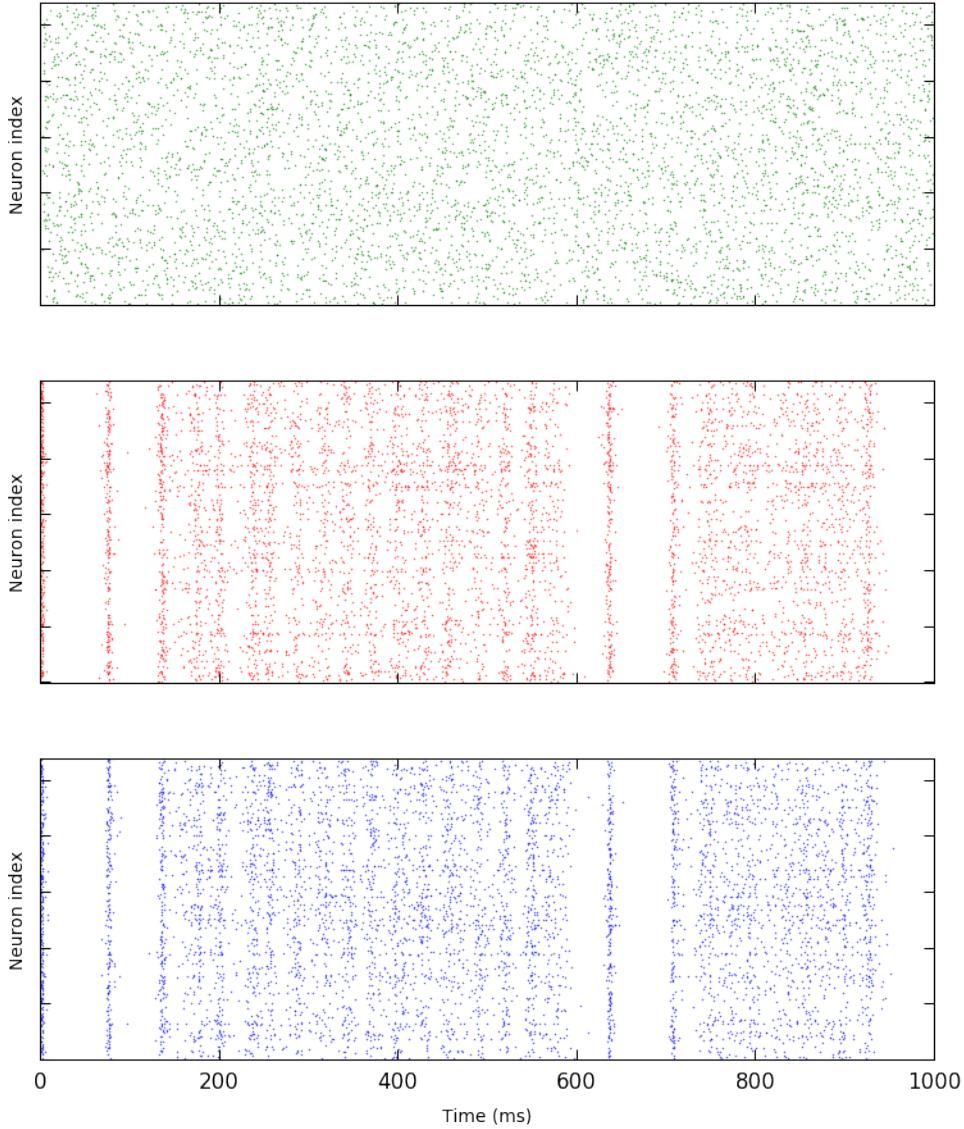


----- input_rate = 100.0 -----

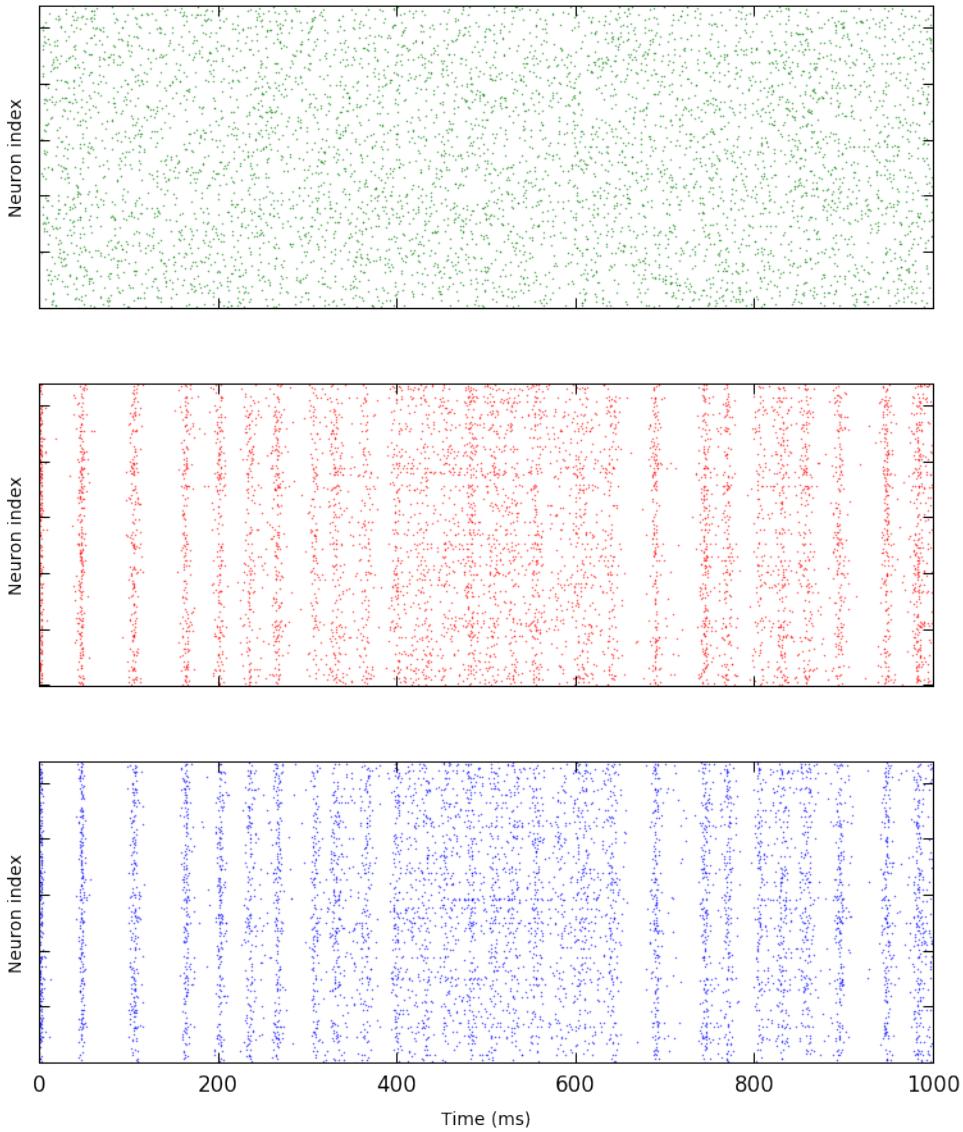


```
>>> _ = net.variationRaster('w_input_exc', net.sim_params['w_input_exc']*np
```

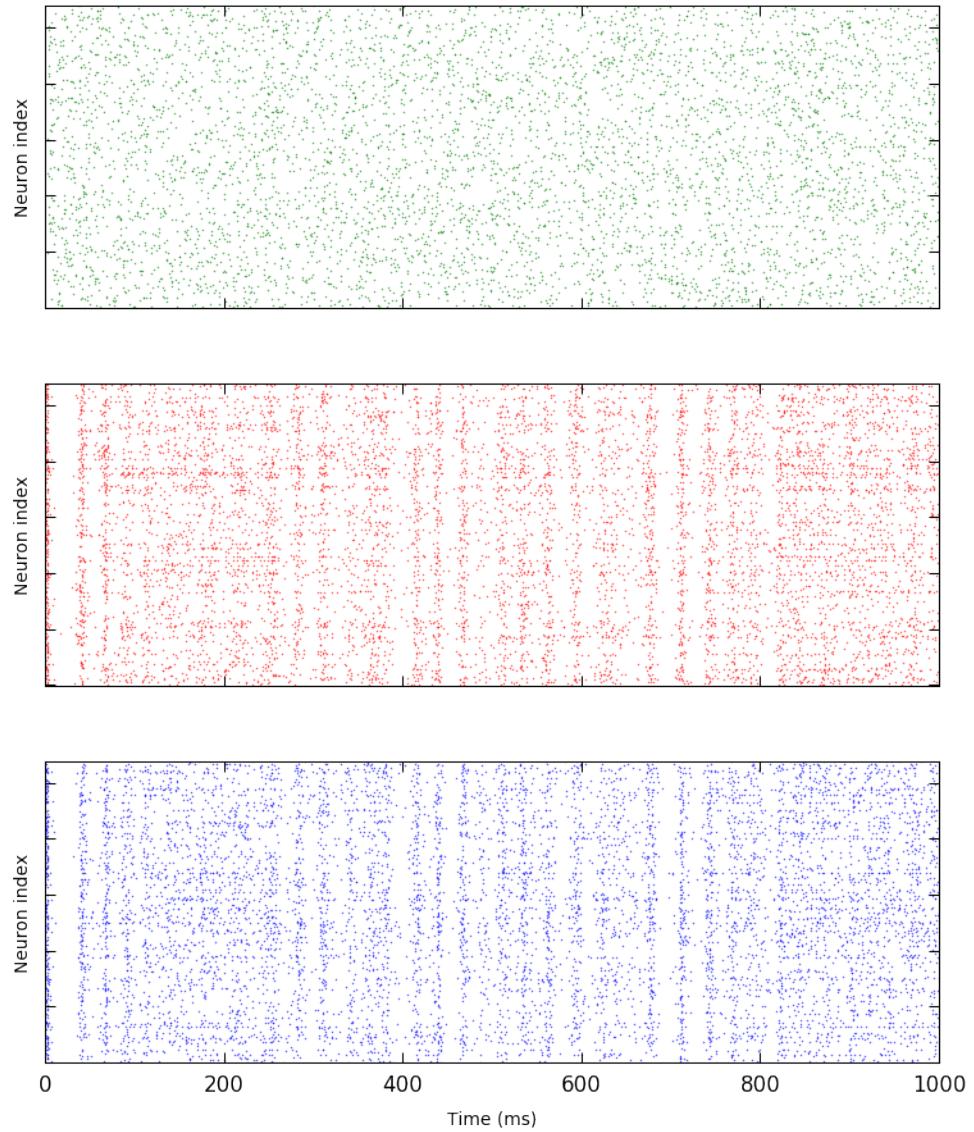
----- w_input_exc = 0.025 -----



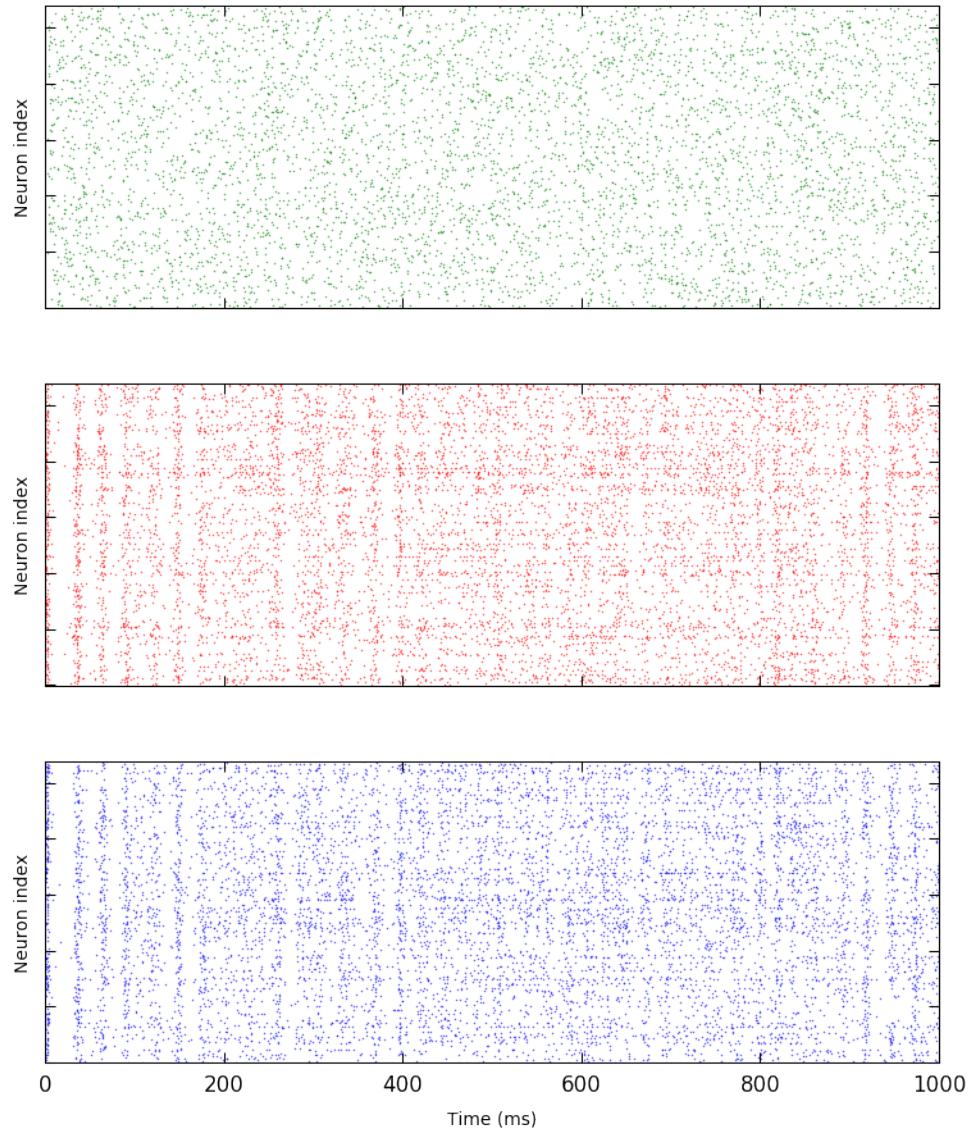
----- w_input_exc = 0.031856874642578345 -----



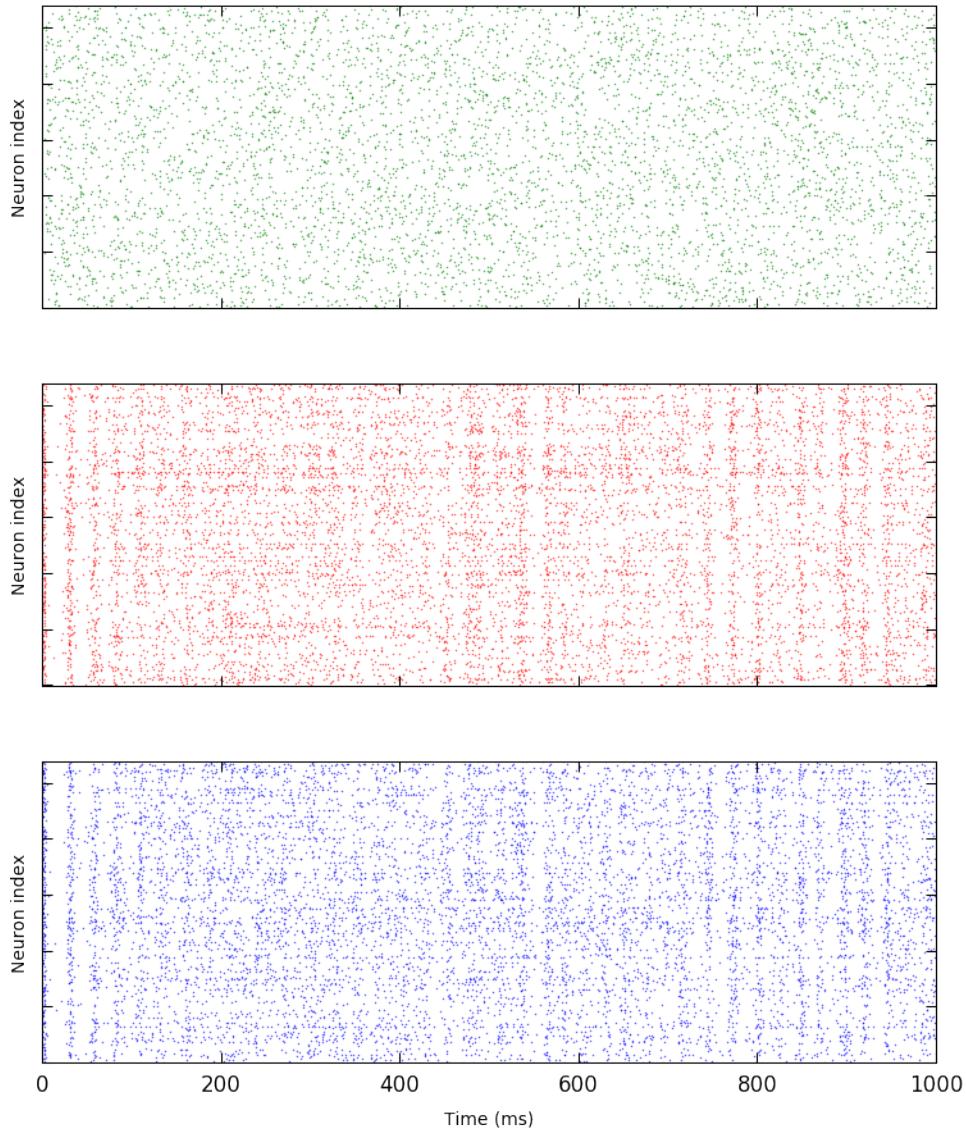
----- w_input_exc = 0.04059441847971804 -----



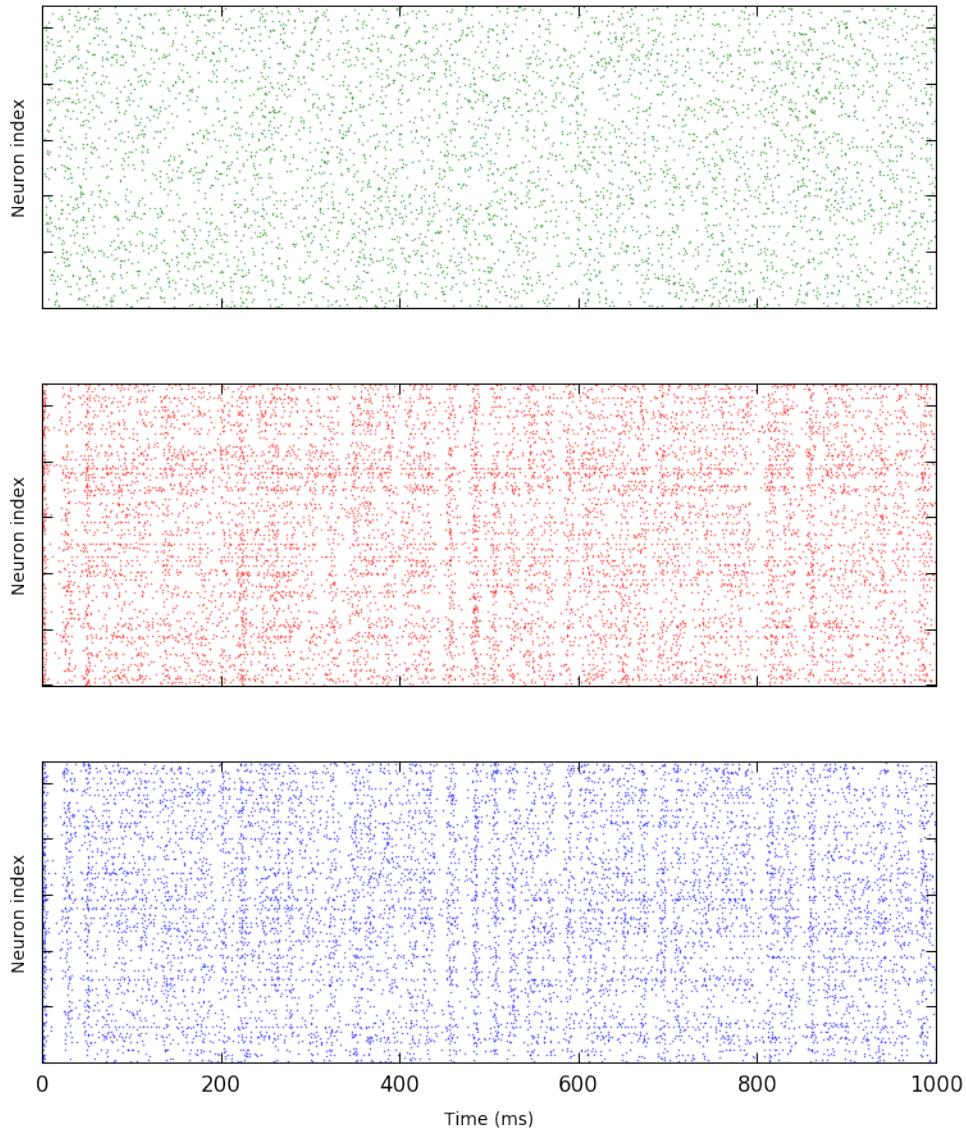
----- w_input_exc = 0.05172845202786974 -----



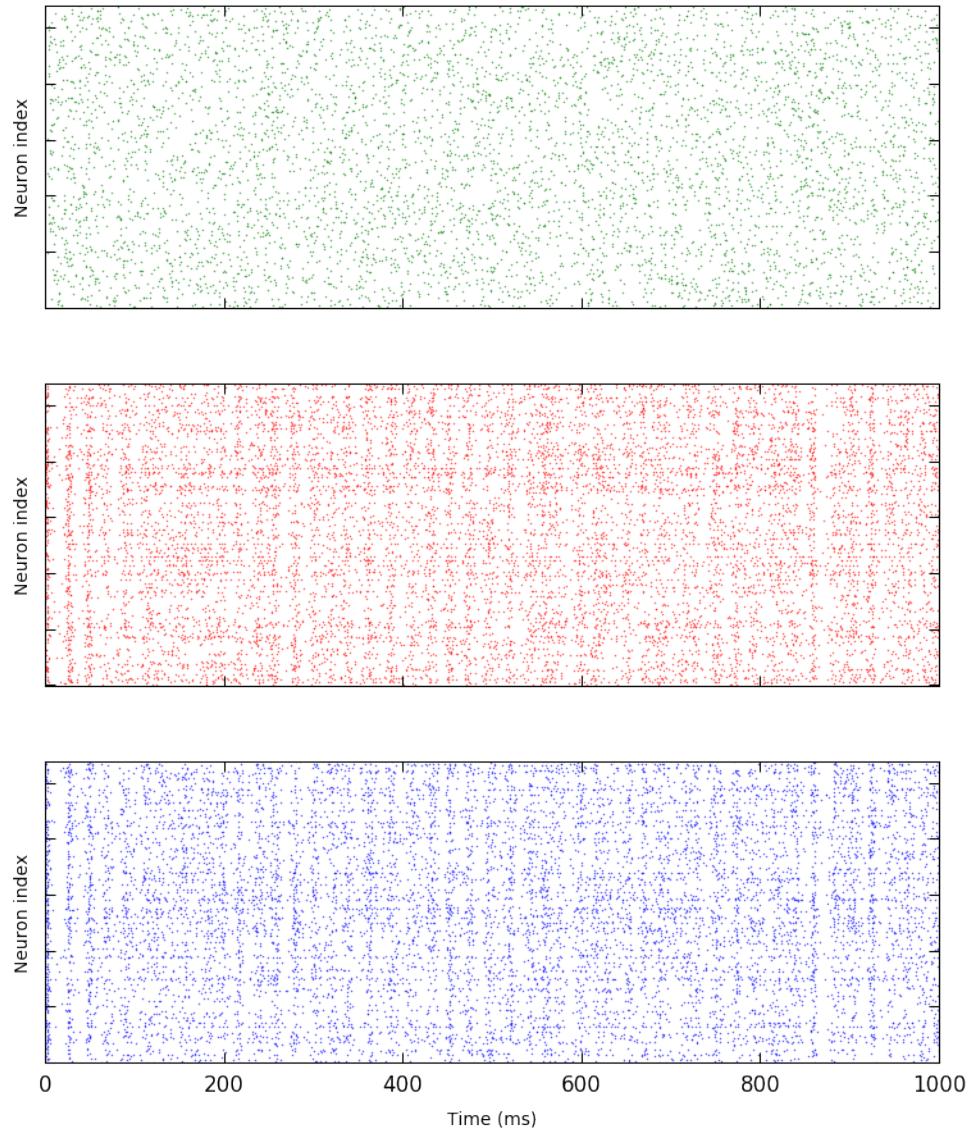
----- w_input_exc = 0.06591627246825896 -----



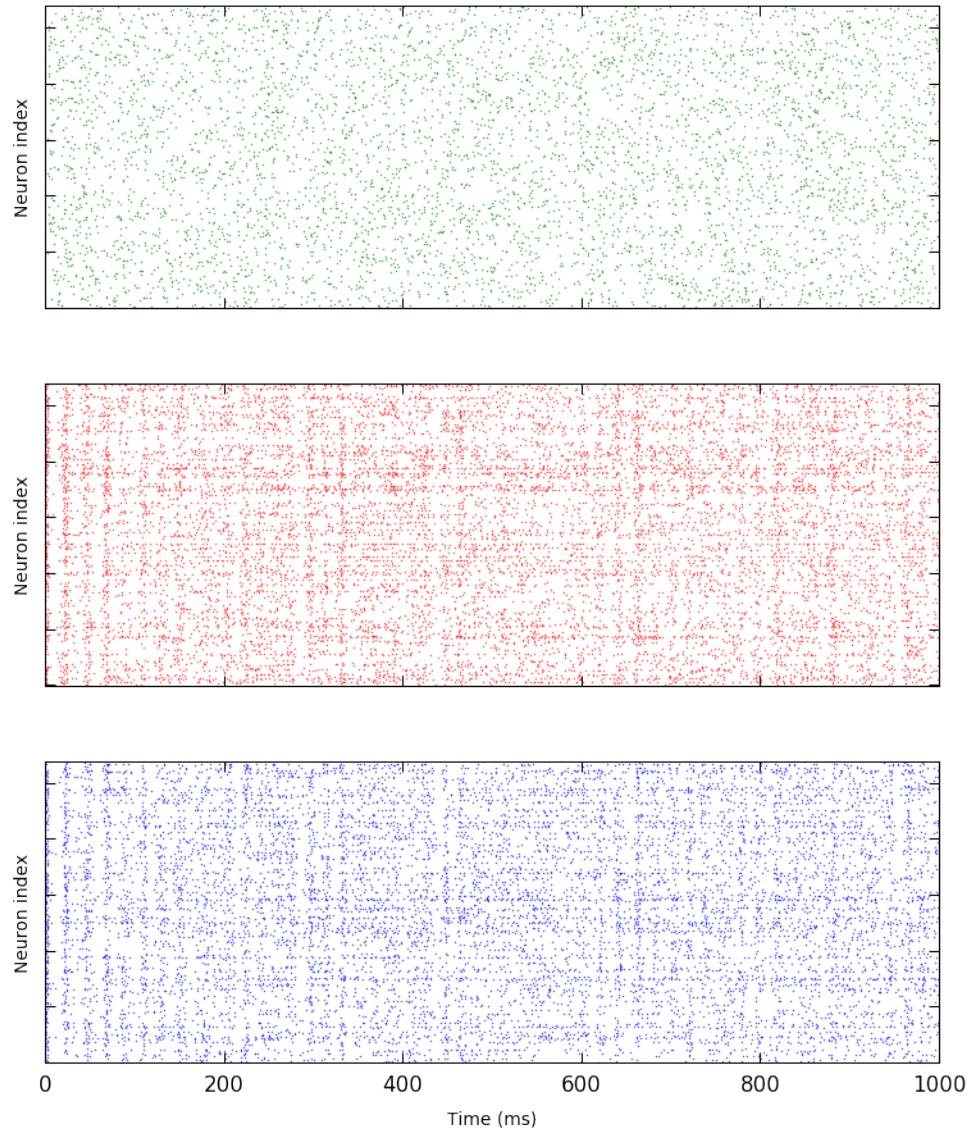
----- w_input_exc = 0.08399545715709454 -----



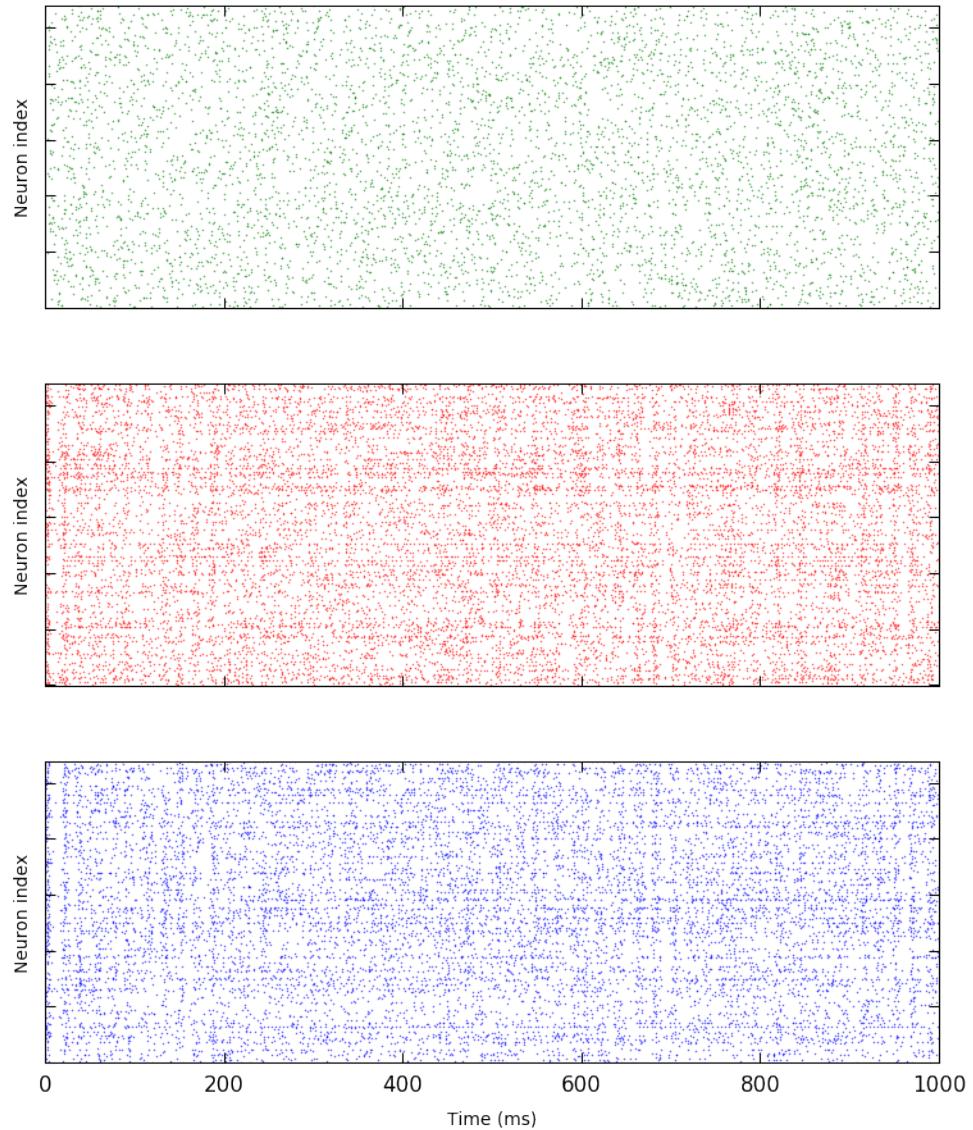
----- w_input_exc = 0.10703330996798484 -----



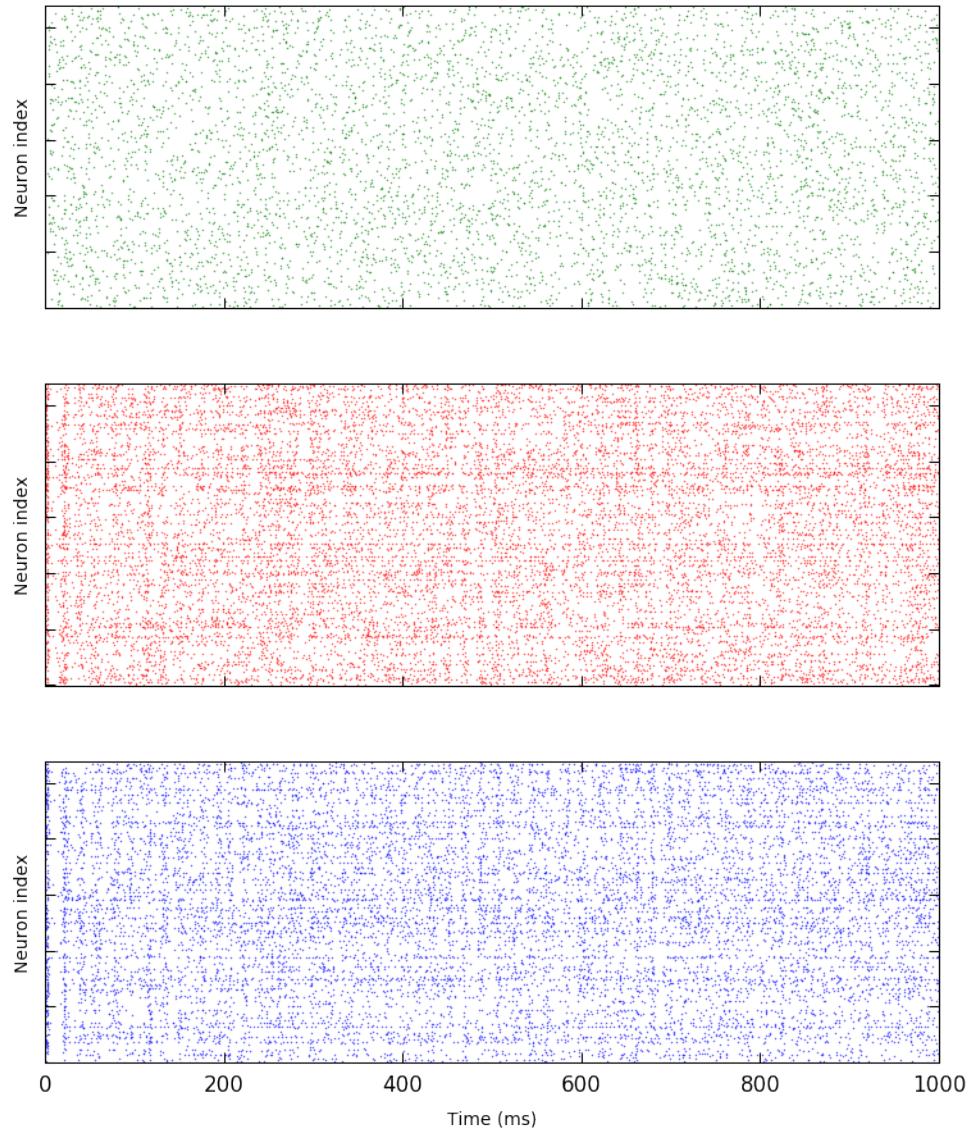
----- w_input_exc = 0.13638986952921298 -----



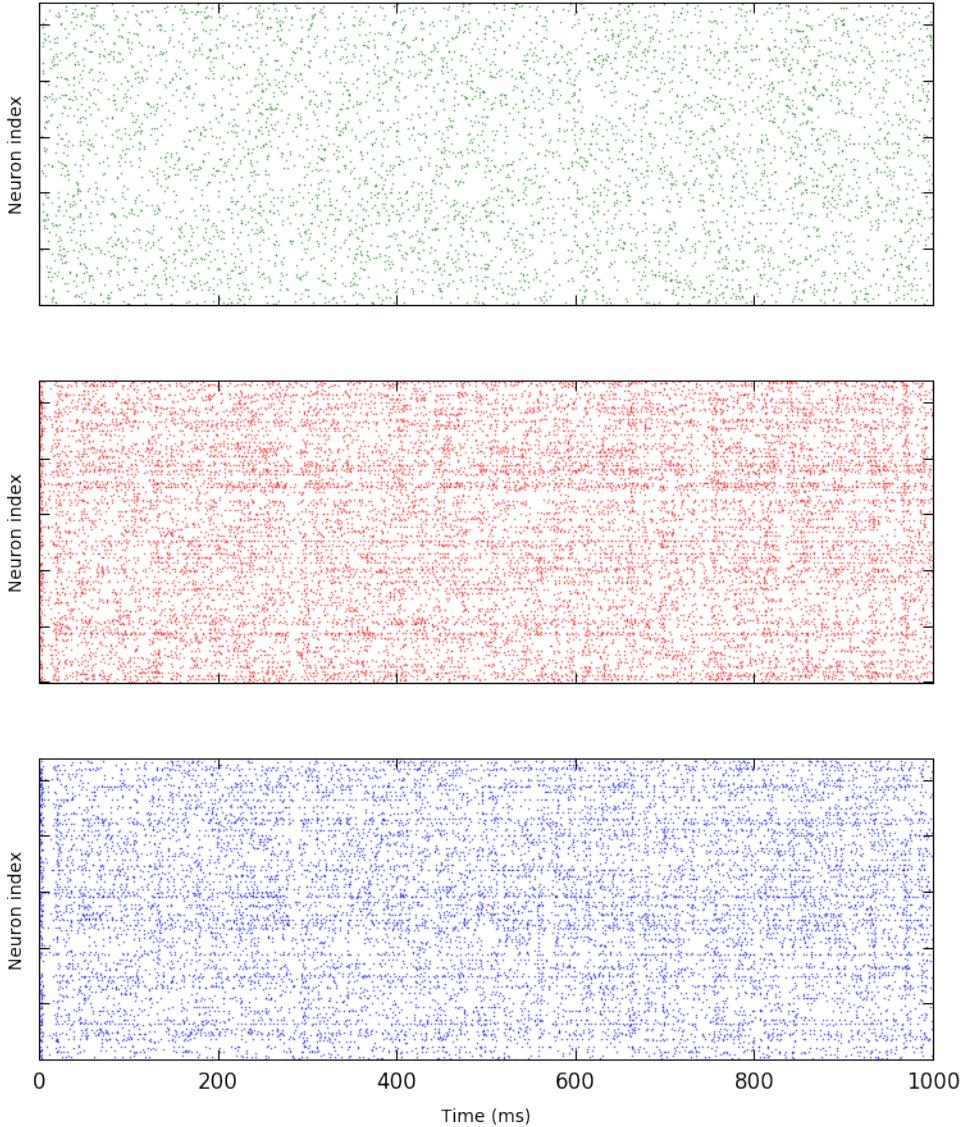
----- w_input_exc = 0.17379819904439014 -----



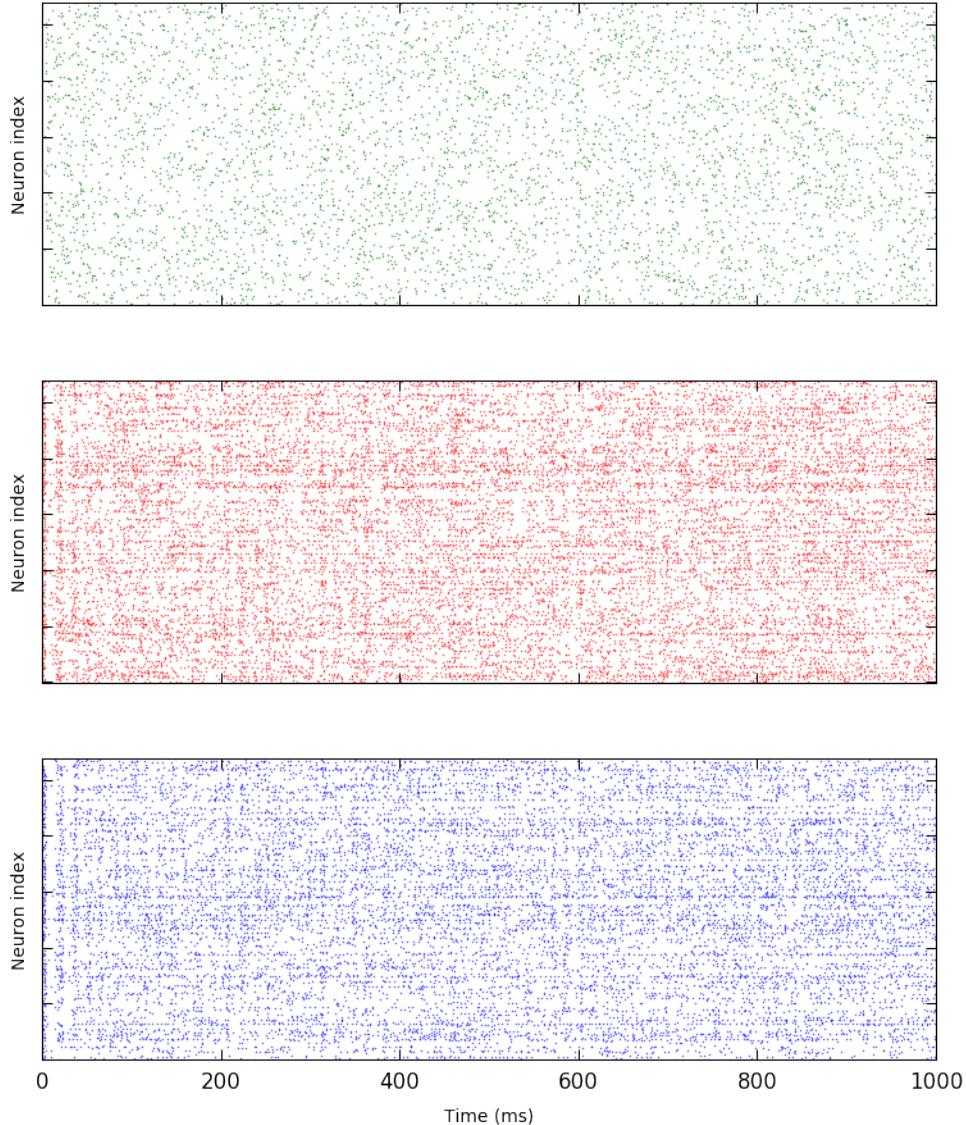
----- w_input_exc = 0.22146669760252063 -----



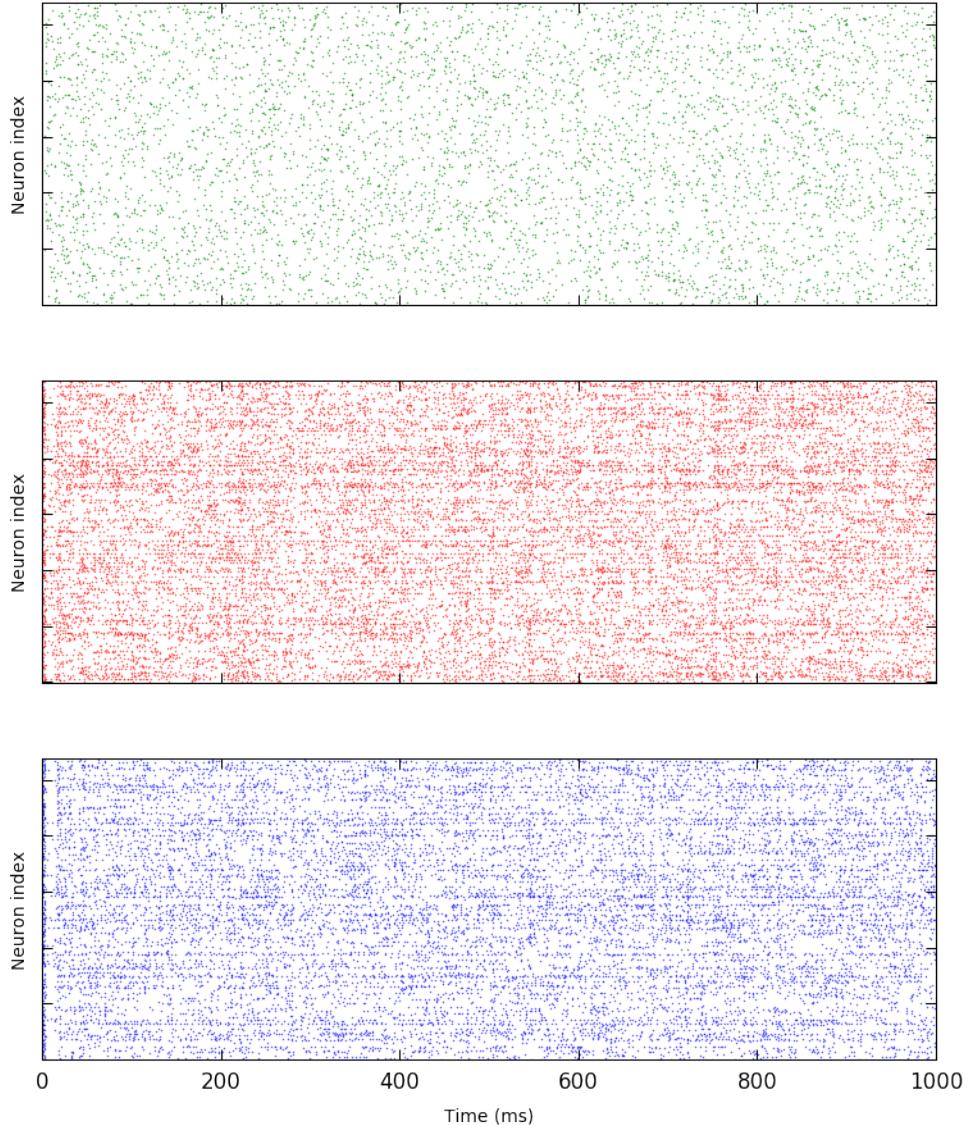
----- w_input_exc = 0.2822094729211722 -----



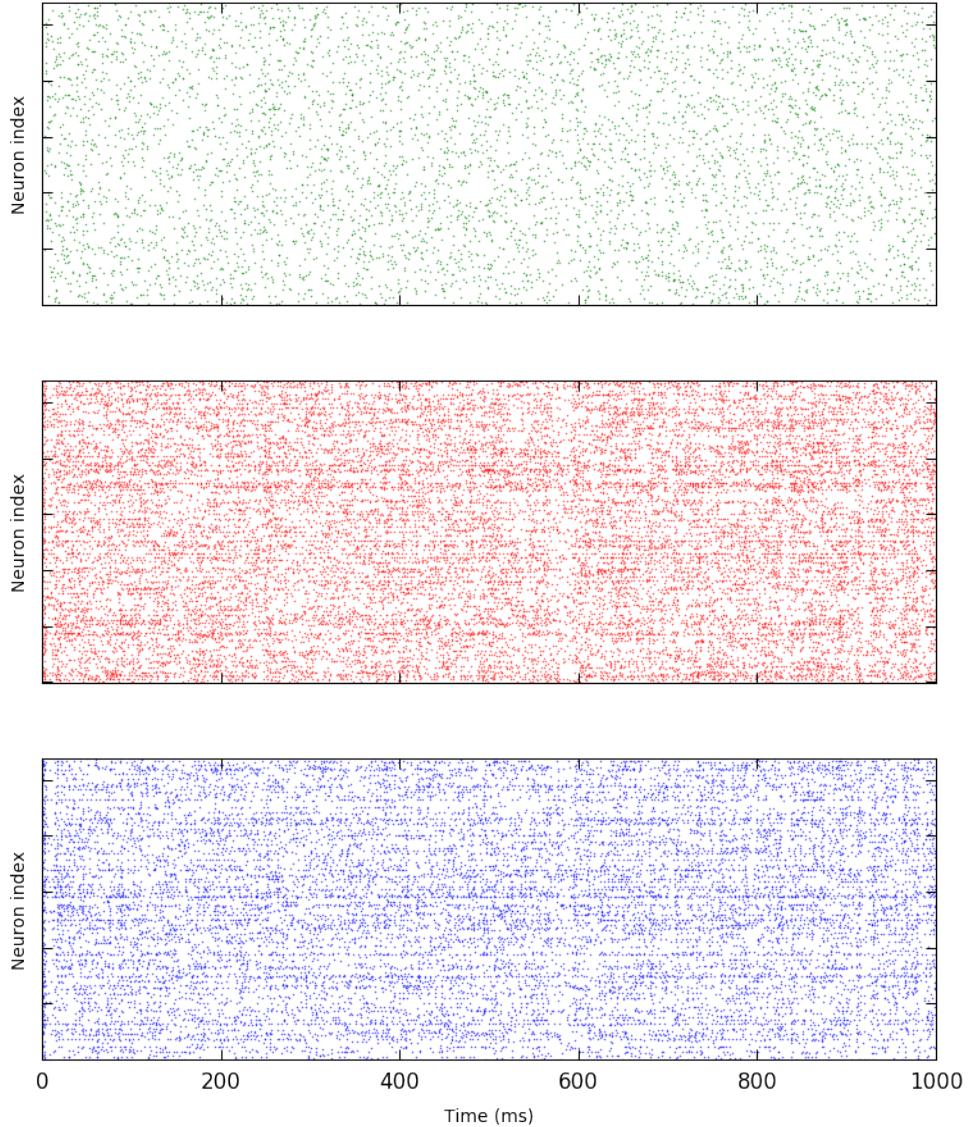
----- w_input_exc = 0.35961247207191577 -----



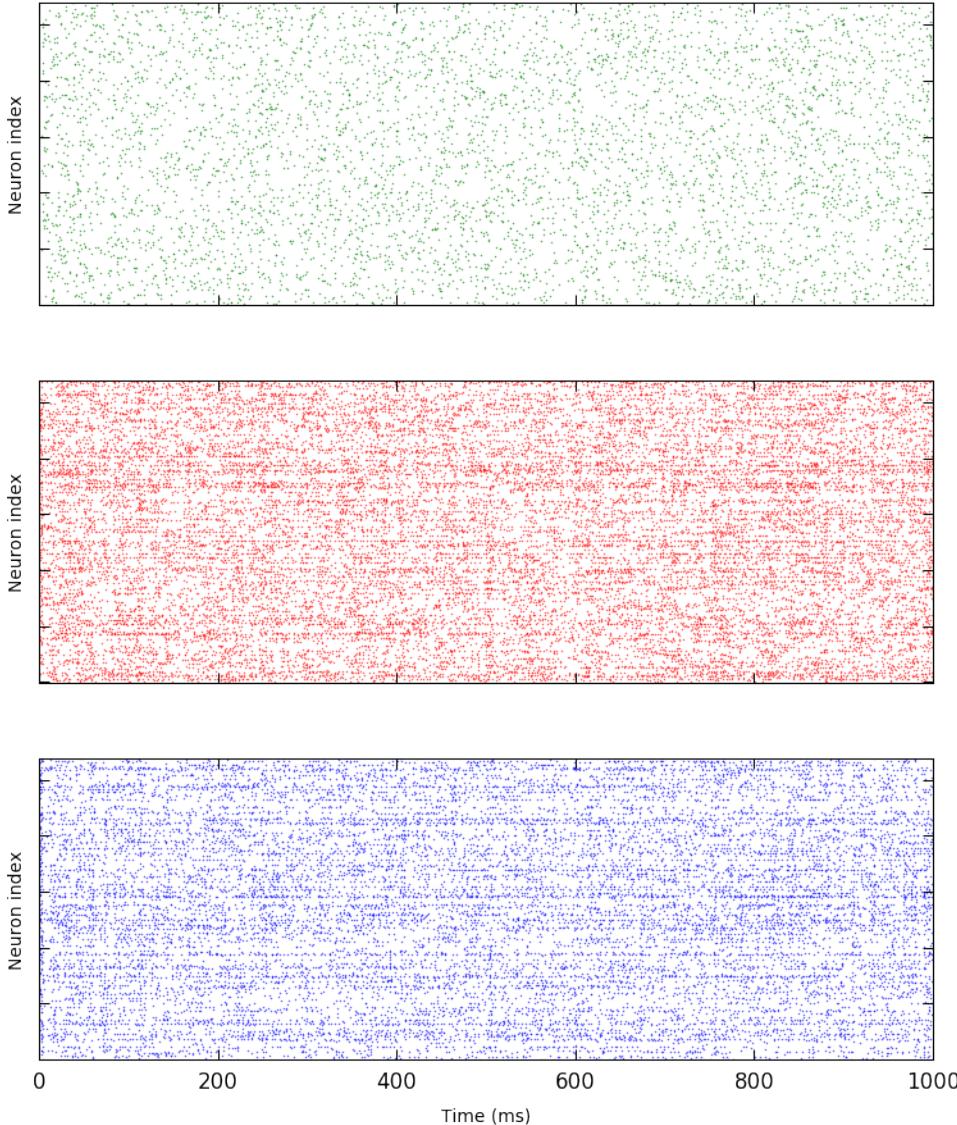
----- w_input_exc = 0.4582451777081089 -----



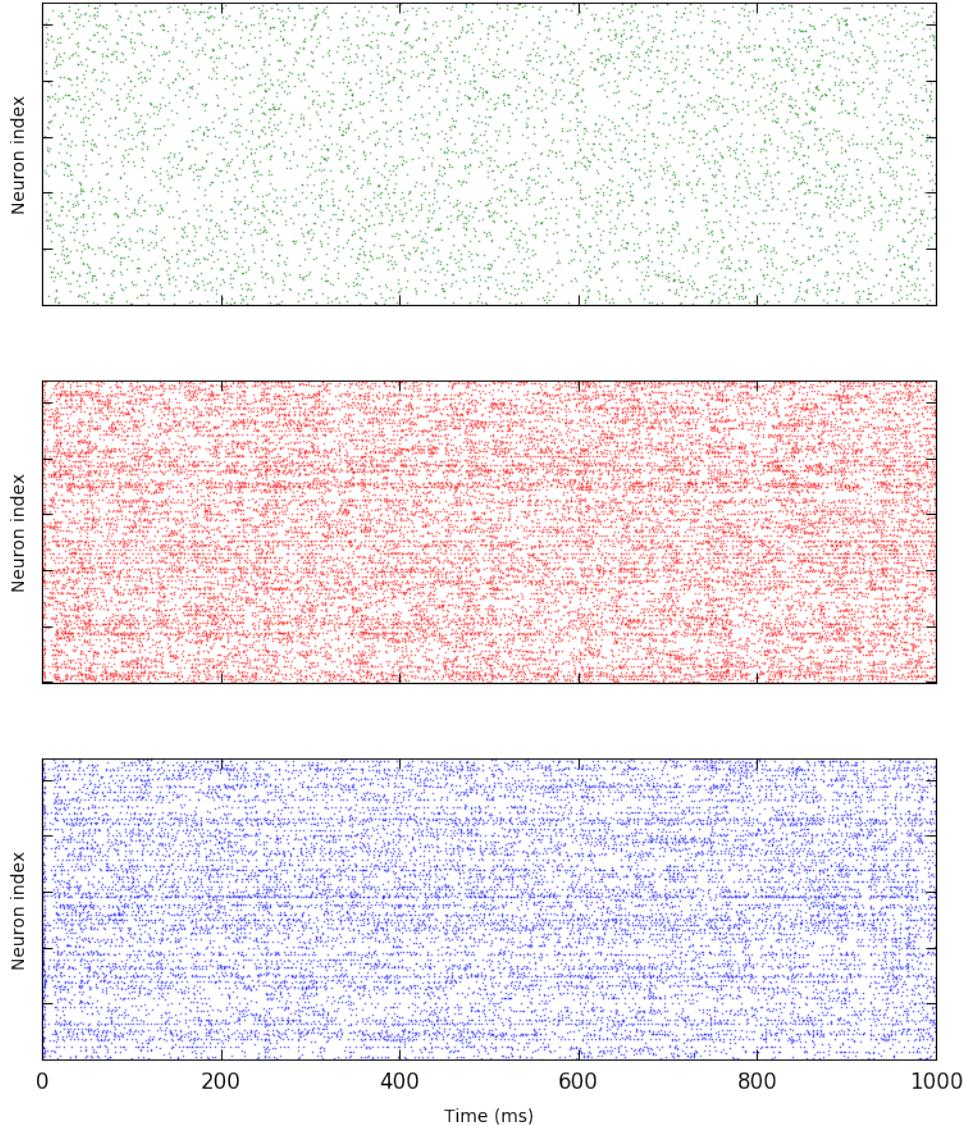
----- w_input_exc = 0.5839303672725303 -----



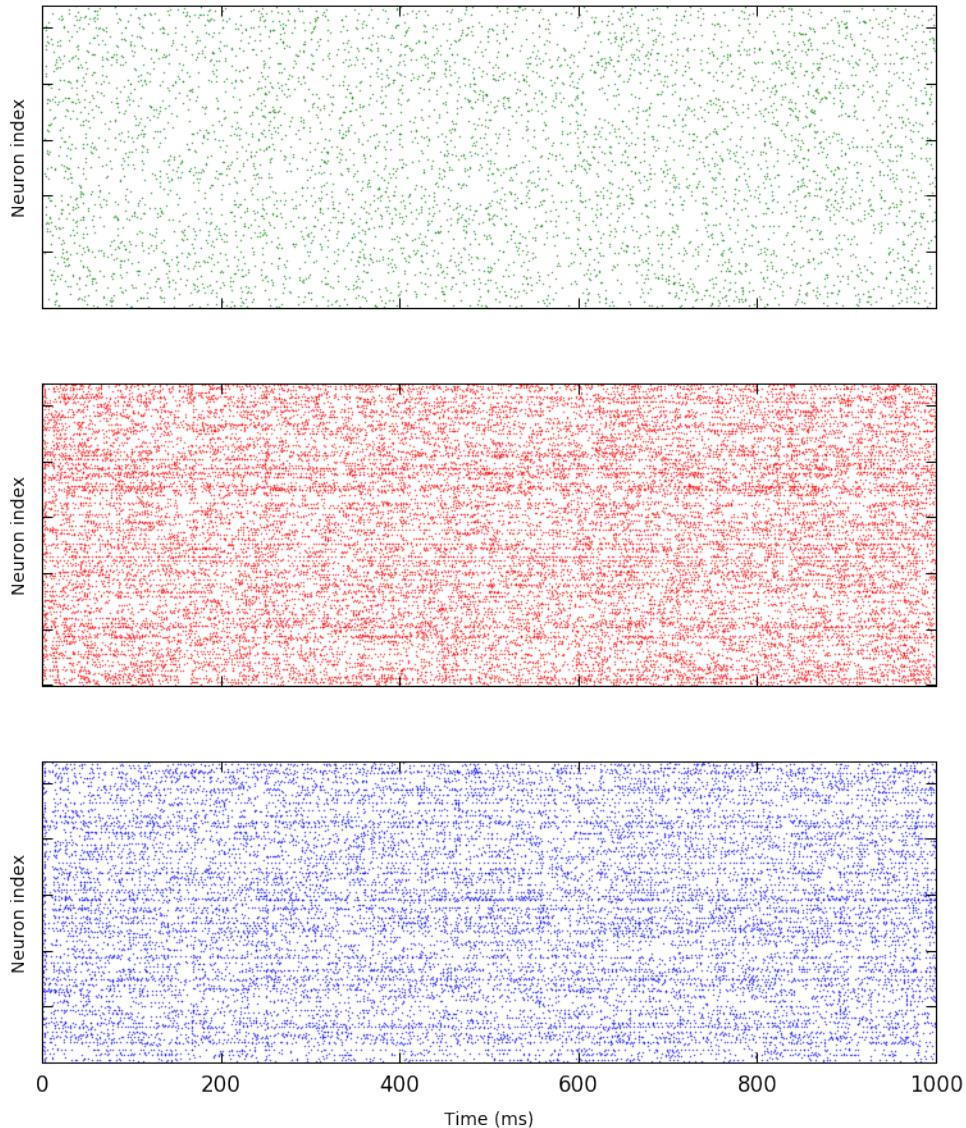
----- w_input_exc = 0.7440878604078294 -----



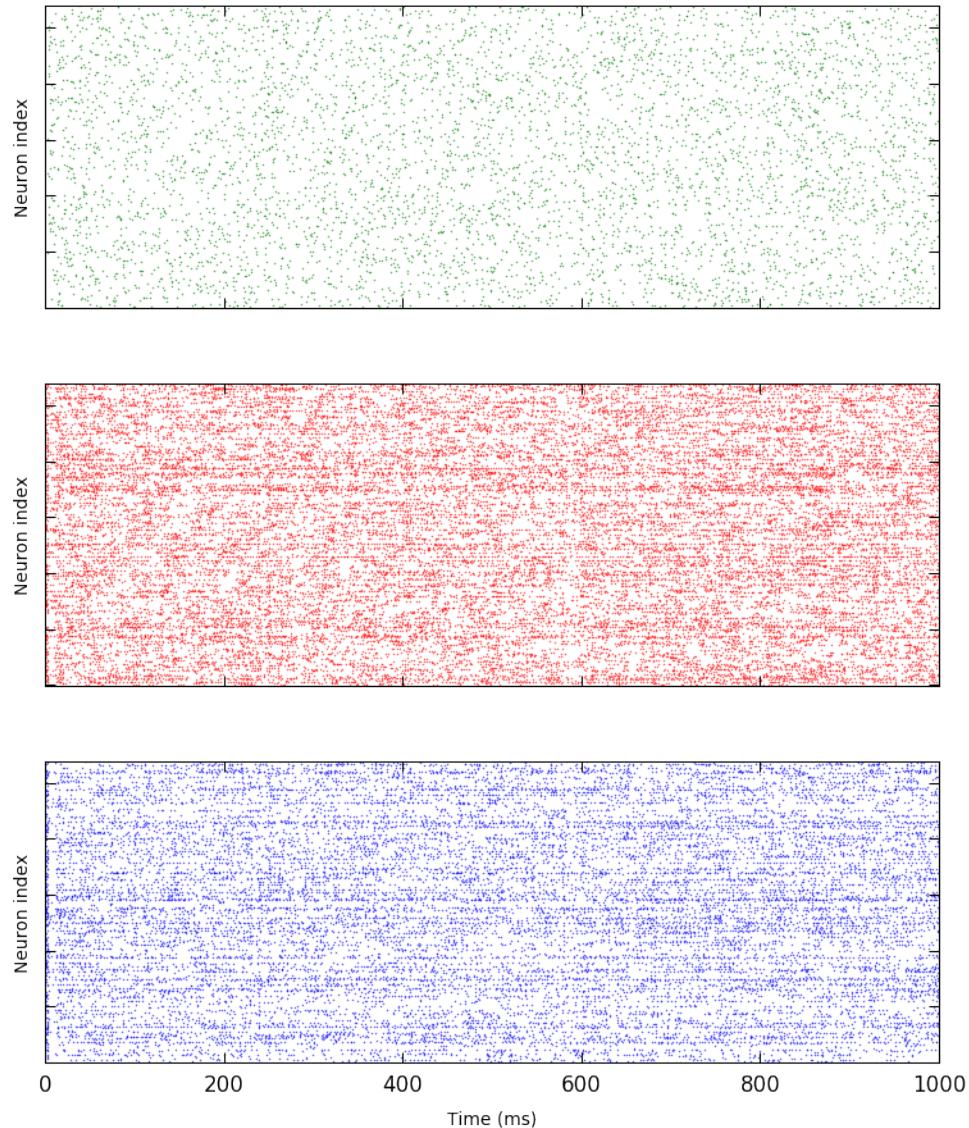
----- w_input_exc = 0.9481725476830625 -----



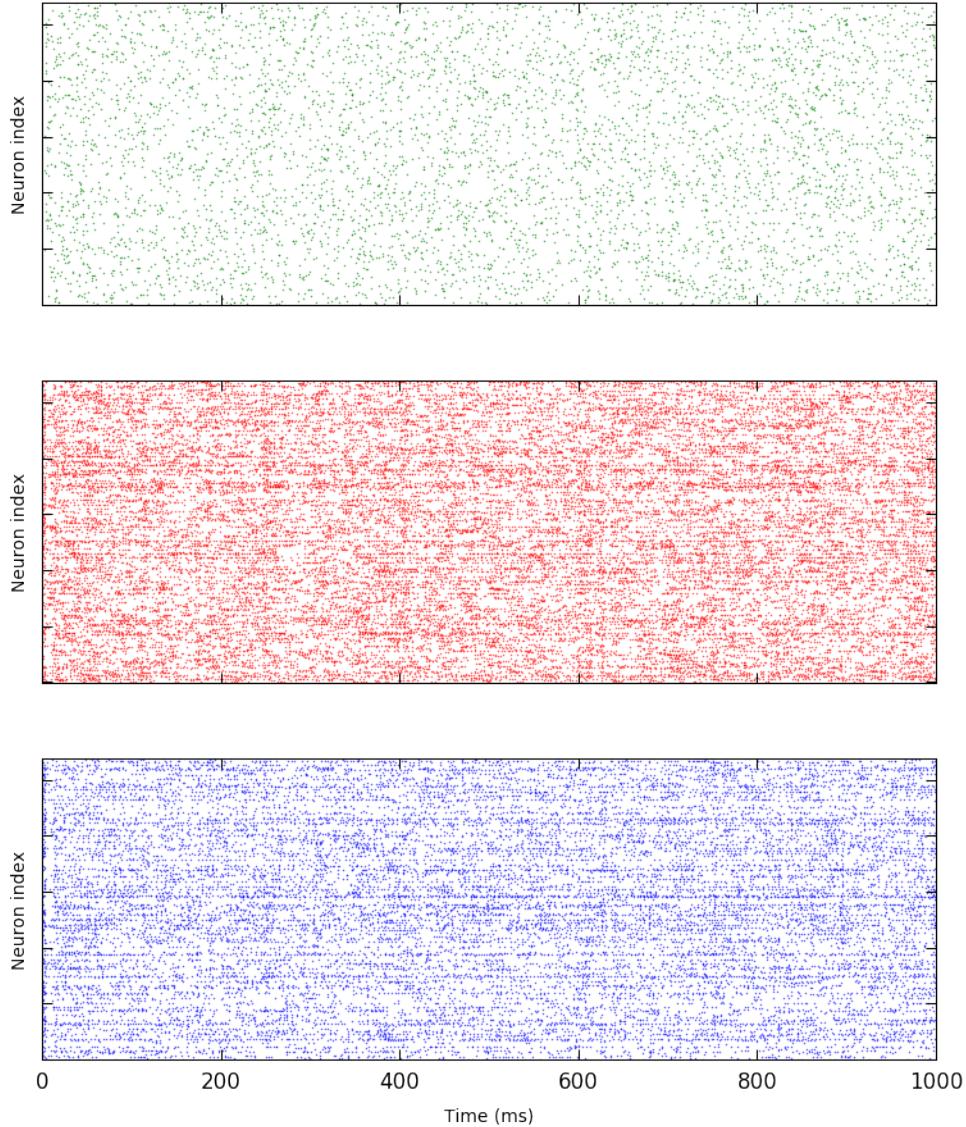
----- w_input_exc = 1.208232559642938 -----



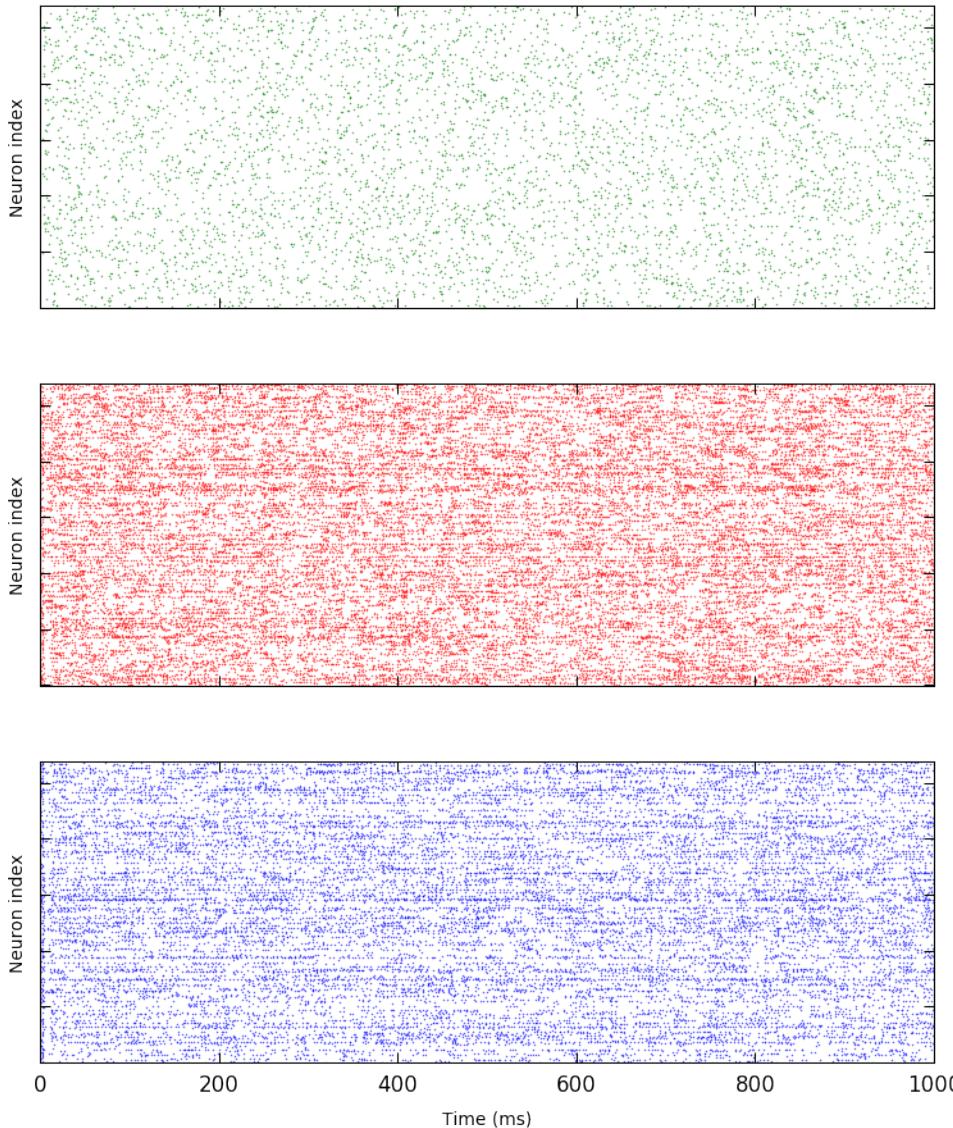
----- w_input_exc = 1.5396205276650652 -----



----- w_input_exc = 1.9618999258786527 -----



----- w_input_exc = 2.5 -----



```
>>> n_sim_each, time = 25, 100
...
... net = RRNN(time = time)
... sim_list = [
...     ('input_rate', net.sim_params['input_rate']*np.logspace(-1,
...     ('w_input_exc', net.sim_params['w_input_exc']*np.logspace(-1,
... ]
...
... net.paramRole(sim_list, f_rate_max=None, datapath='/tmp/OB-V1_data/RRNN-
```

```

-----
NameError                                                 Traceback (most recent call last)

<ipython-input-4fea7d4e59fd7> in <module>()
    7 ]
    8
--> 9 net.paramRole(sim_list, f_rate_max=None, datapath='/tmp/OB-V1_da

NameError: name 'tag' is not defined

```

2.2.6 Effet d'une covariation de l'activité d'entrée et de son poids :

Il a été observé précédemment que, l'activité de la population source et les poids des connexions, entre la population source et la population E, sont positivement corrélés à l'activité du réseau. Aussi, ces deux paramètres paraissent dépendants l'un de l'autre. Il est donc nécessaire de vérifier cette interdépendance.

Pour cela, des rasterplots sont générés à partir d'une covariation de ces deux paramètres. Cette covariation est effectuée en pondérant chacun des deux paramètres avec un coefficient distinct. Ces deux coefficients vont parcourir le même intervalle de valeurs, mais l'un dans un sens contraire à l'autre. De telle sorte, en fait, que le produit des deux paramètres soit toujours le même.

Nous observons que seule l'activité de la population source varie. L'activité observée dans les populations E et I n'évolue pas, car le flux d'entrée, défini par l'activité de la population source et son poids, est en fait toujours le même. Nous étudions maintenant les propriétés de la connectique interne au réseau.

```

>>> from RRNN import RRNN
... import numpy as np
...
... net = RRNN()
... rateI = net.sim_params['input_rate']
... wIe = net.sim_params['w_input_exc']
...
... _ = net.doubleVariationRaster_P2P('input_rate', rateI*np.logspace(-1, 1,
... 'w_input_exc', wIe*np.logspace(1, -1,

```

2.2.7 Rôle du poids global

Maintenant que le flux d'entrée est défini, il s'agit ici d'étudier les effets de la manipulation du poids global W . Modifier ce paramètre revient à changer tous les poids, hormis celui de la projection entre la source et la population E.

Une simulation d'une seconde est exécutée pour chaque valeur de W appartenant à l'intervalle de sa variation. Des rasterplots représentant l'activité des trois populations du modèle sont alors générés.

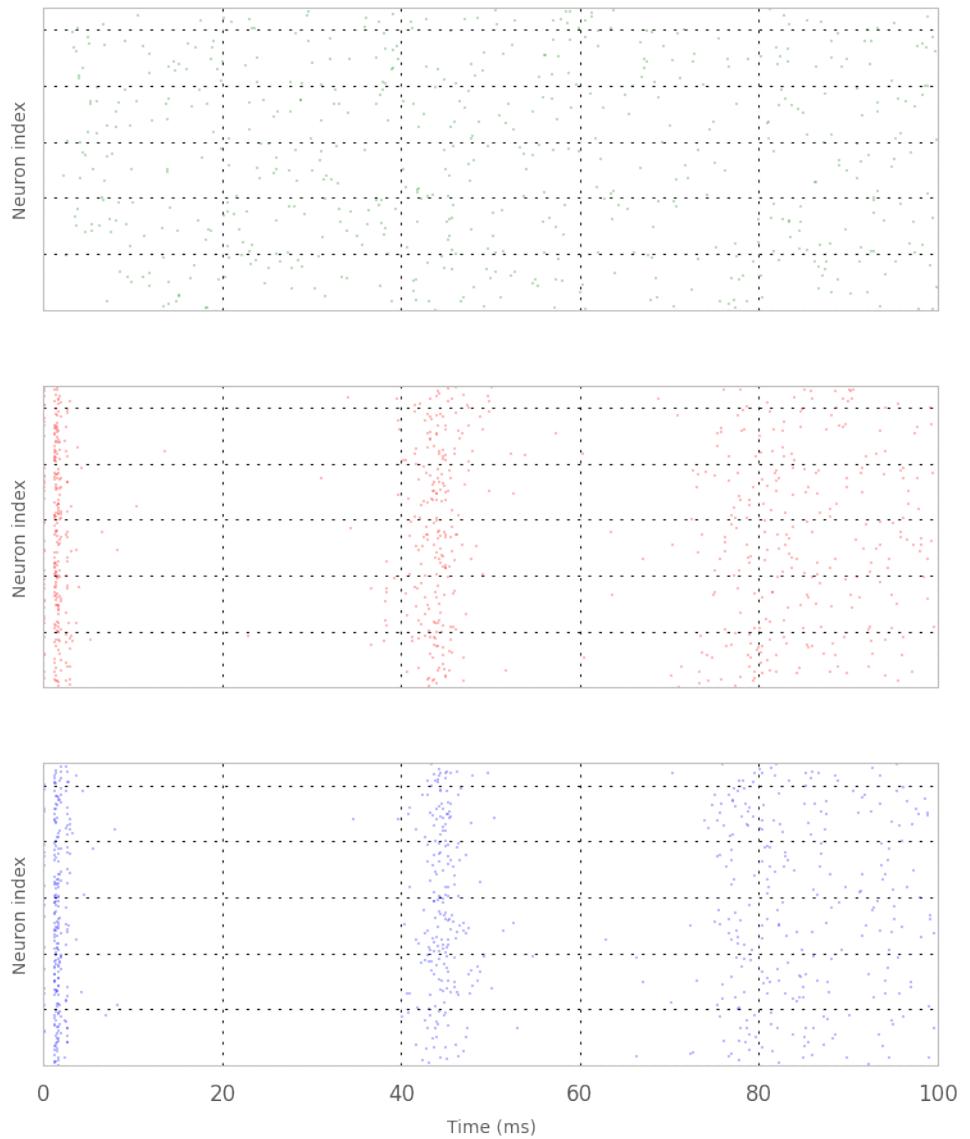
Une augmentation de W modifie bien le comportement des populations excitatrice et inhibitrice. Nous pouvons observer pour certaines valeurs de W , dans les populations E et I, un comportement de type Asynchronous Regular qui est un des quatres états d'activité décrits par Brunel, dont nous parlerons plus tard.

```
>>> from RRNN import RRNN
... import numpy as np
... import matplotlib.pyplot as plt
...
... n_sim_each, time = 25, 1000
... n_sim_each, time = 5, 100
...
... net = RRNN(time=time)
... df, spikesE, spikesI = net.model()
... net.Raster(df, spikesE, spikesI, input=True, title='weight = {}'.format
... plt.show()
...
... for w in net.w * np.logspace(-1, 1., n_sim_each):
...     net = RRNN(w=w)
...     df, spikesE, spikesI = net.model()
...     net.Raster(df, spikesE, spikesI, input=True, title='weight = {}'.format
...     plt.show()
...
... for w in net.w * np.logspace(-.3, .3, n_sim_each):
...     net = RRNN(w=w)
...     df, spikesE, spikesI = net.model()
...     net.Raster(df, spikesE, spikesI, input=True, title='weight = {}'.format
...     plt.show()

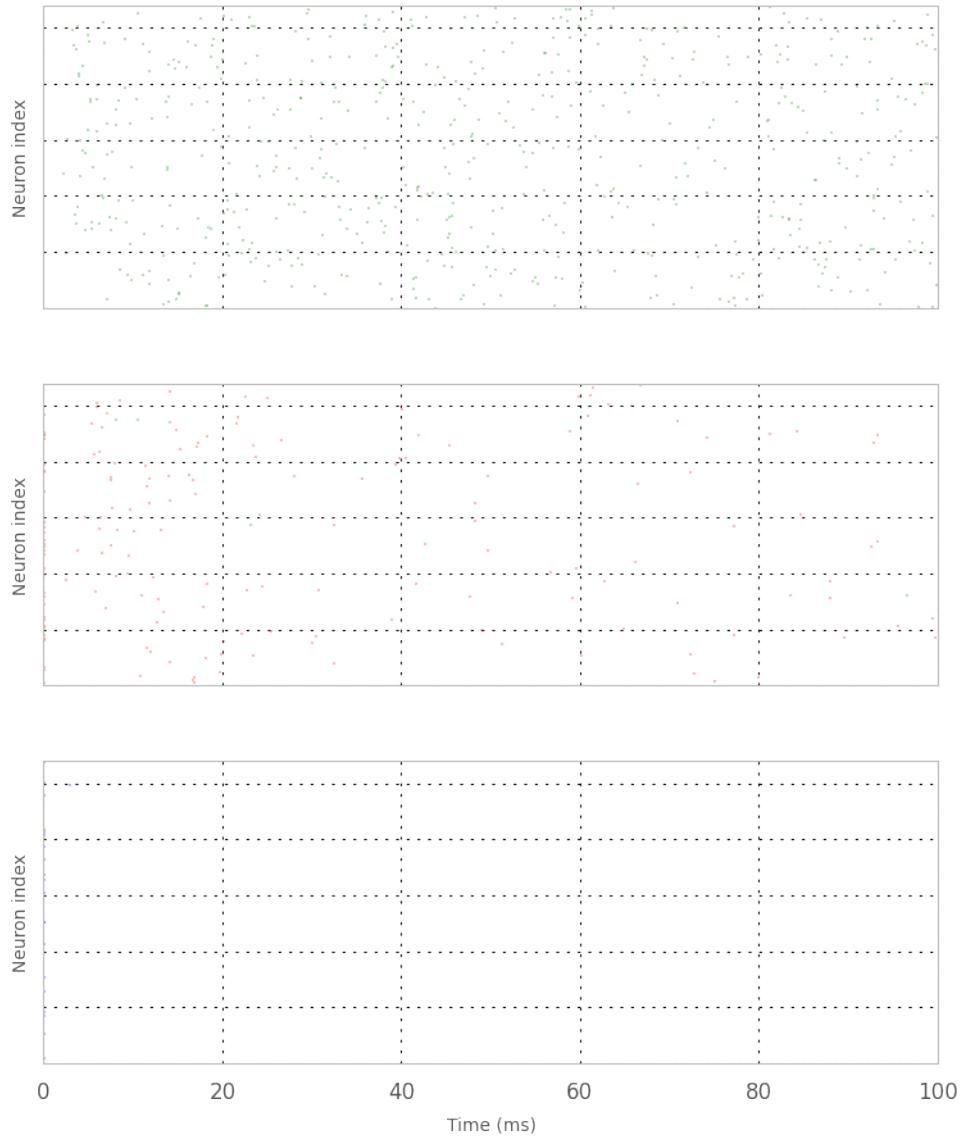
CSAConnector: libneurosim support not available in NEST.
Falling back on PyNN's default CSAConnector.
Please re-compile NEST using --with-libneurosim=PATH
/usr/local/lib/python2.7/site-packages/matplotlib/__init__.py:1318: UserWarning:
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

/usr/local/lib/python2.7/site-packages/NeuroTools/__init__.py:130: DependencyWarning:
To have functions using interval please install the package.
website : http://pypi.python.org/pypi/interval/1.0.0
```

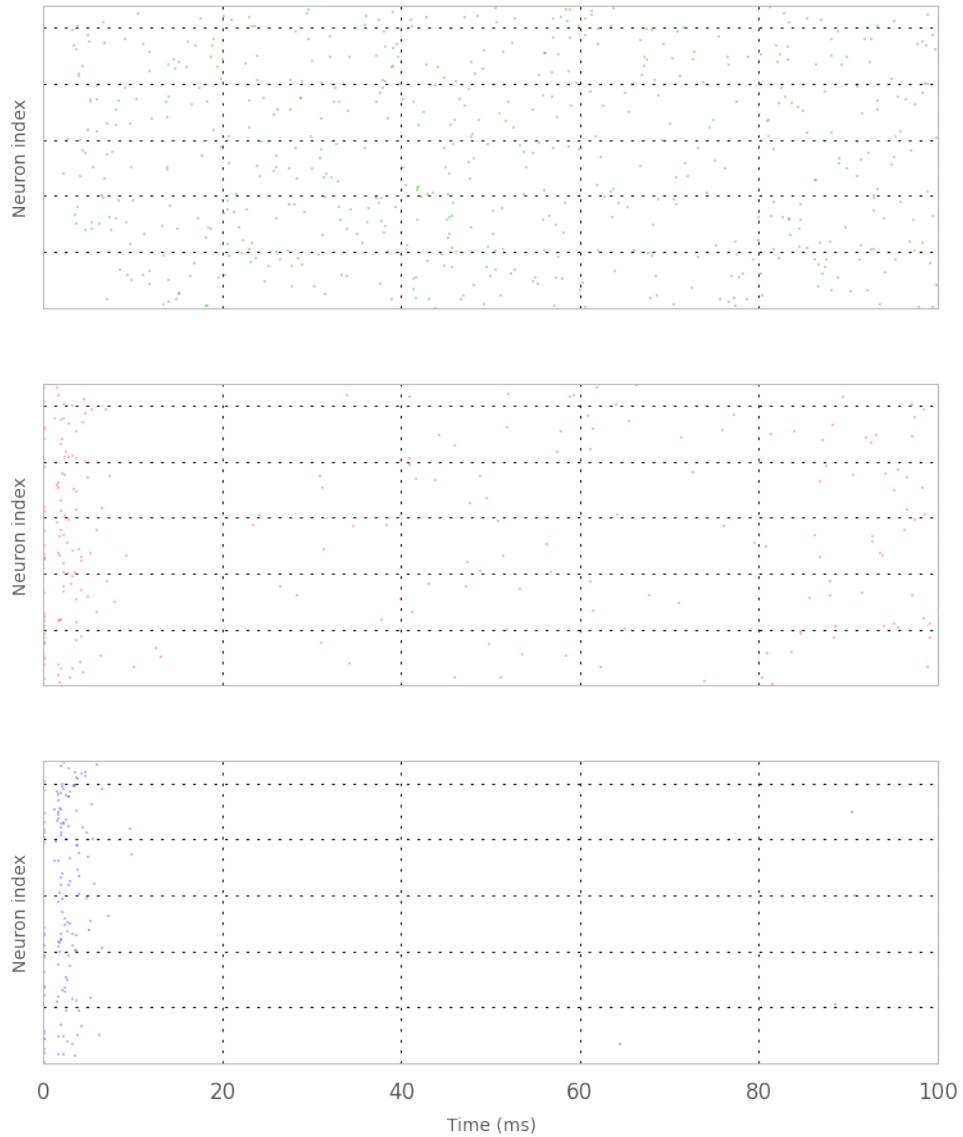
----- weight = 0.06 -----



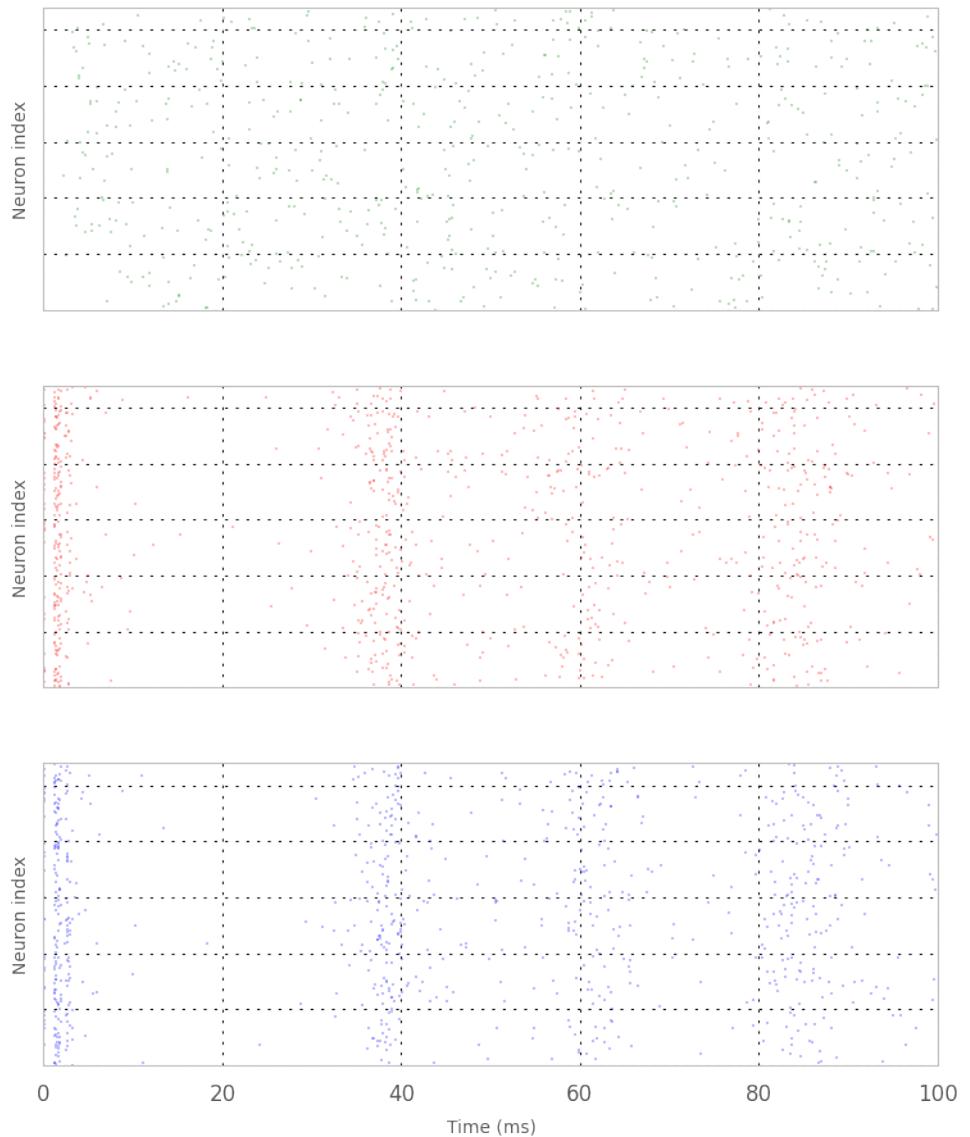
----- weight = 0.006 -----



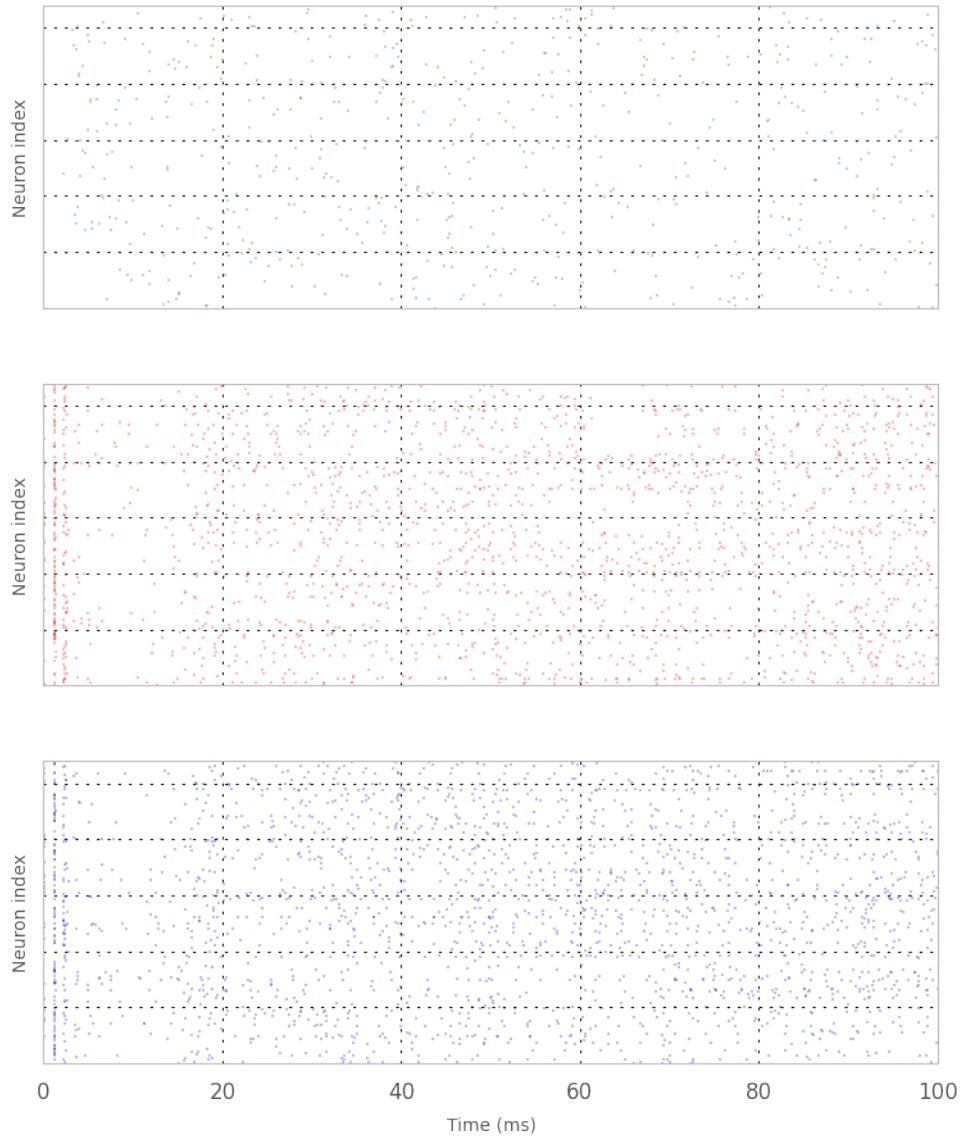
----- weight = 0.018973665961 -----



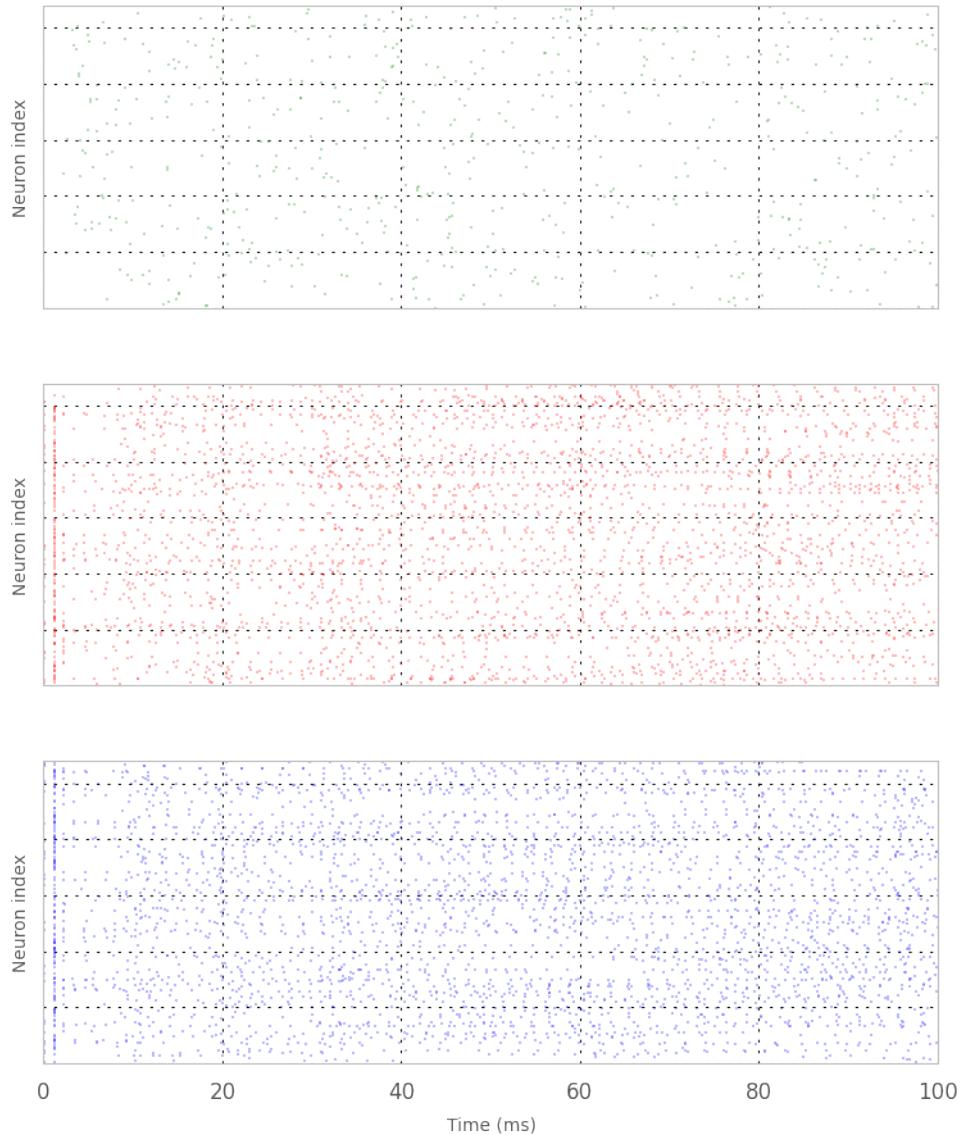
----- weight = 0.06 -----



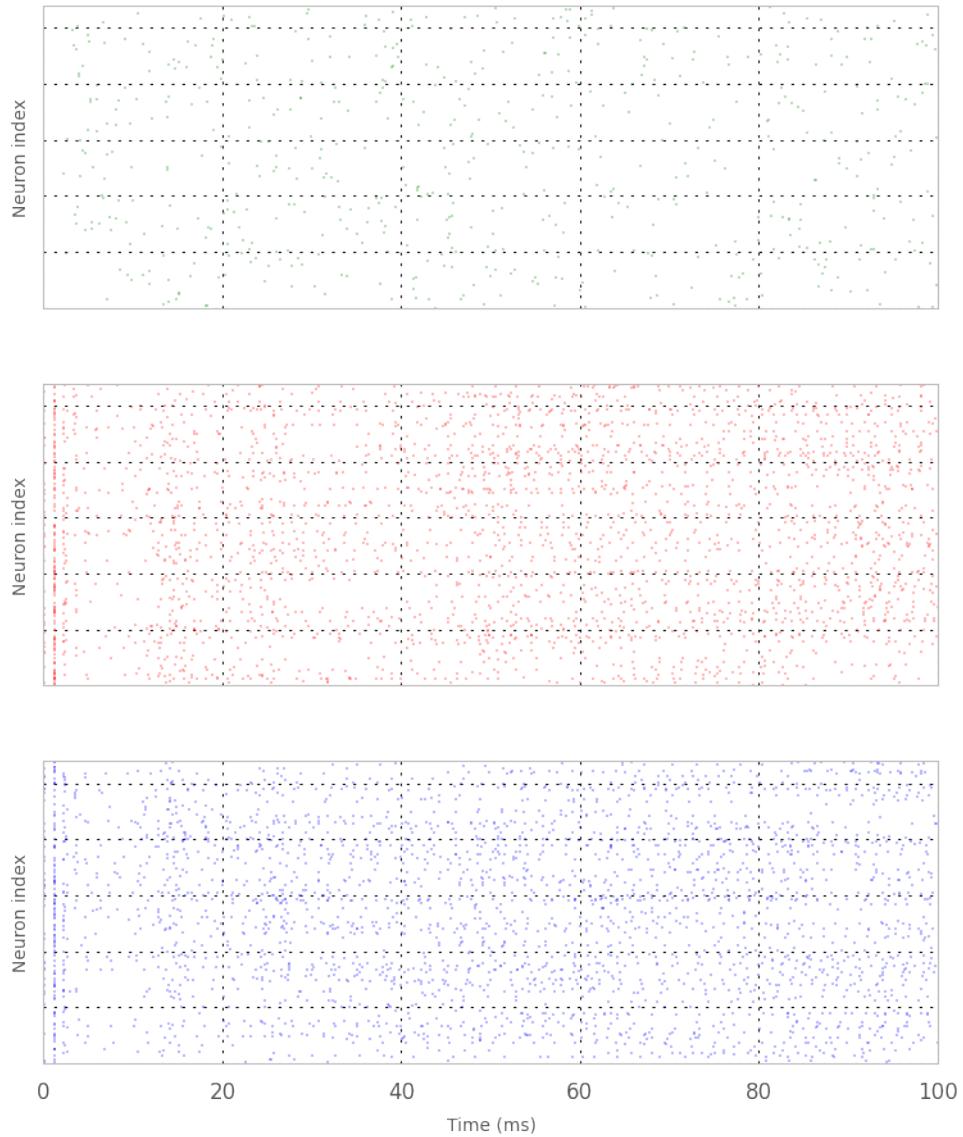
----- weight = 0.18973665961 -----



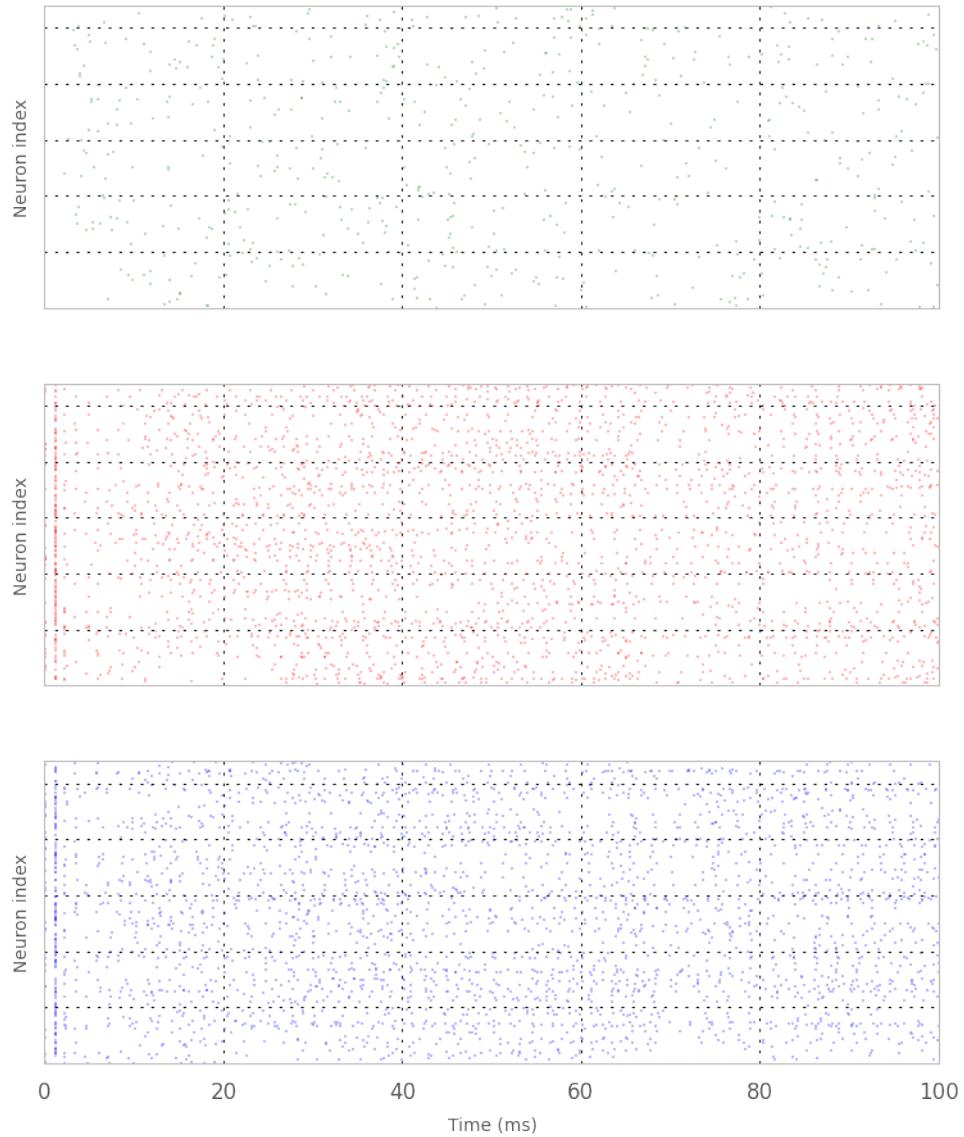
----- weight = 0.6 -----



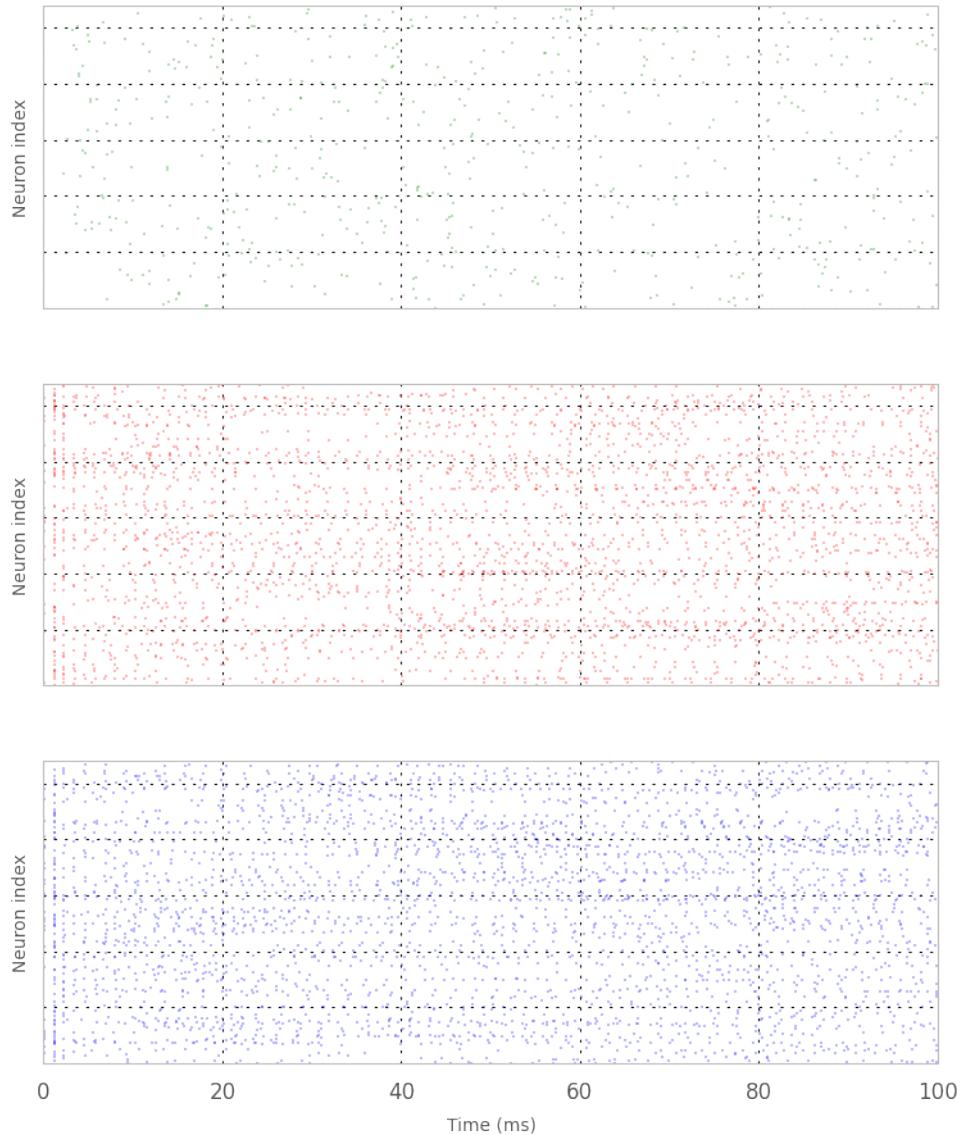
----- weight = 0.300712340176 -----



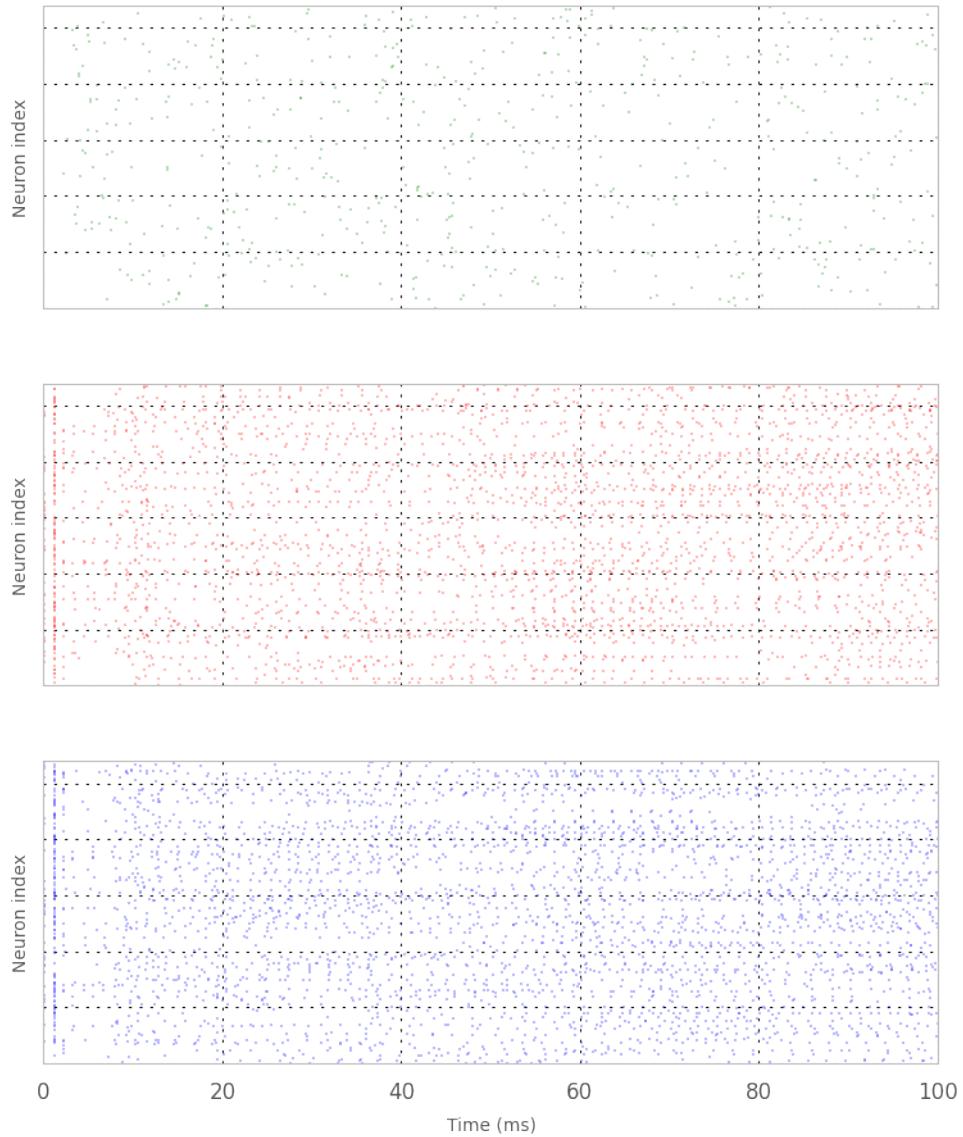
----- weight = 0.42476747063 -----

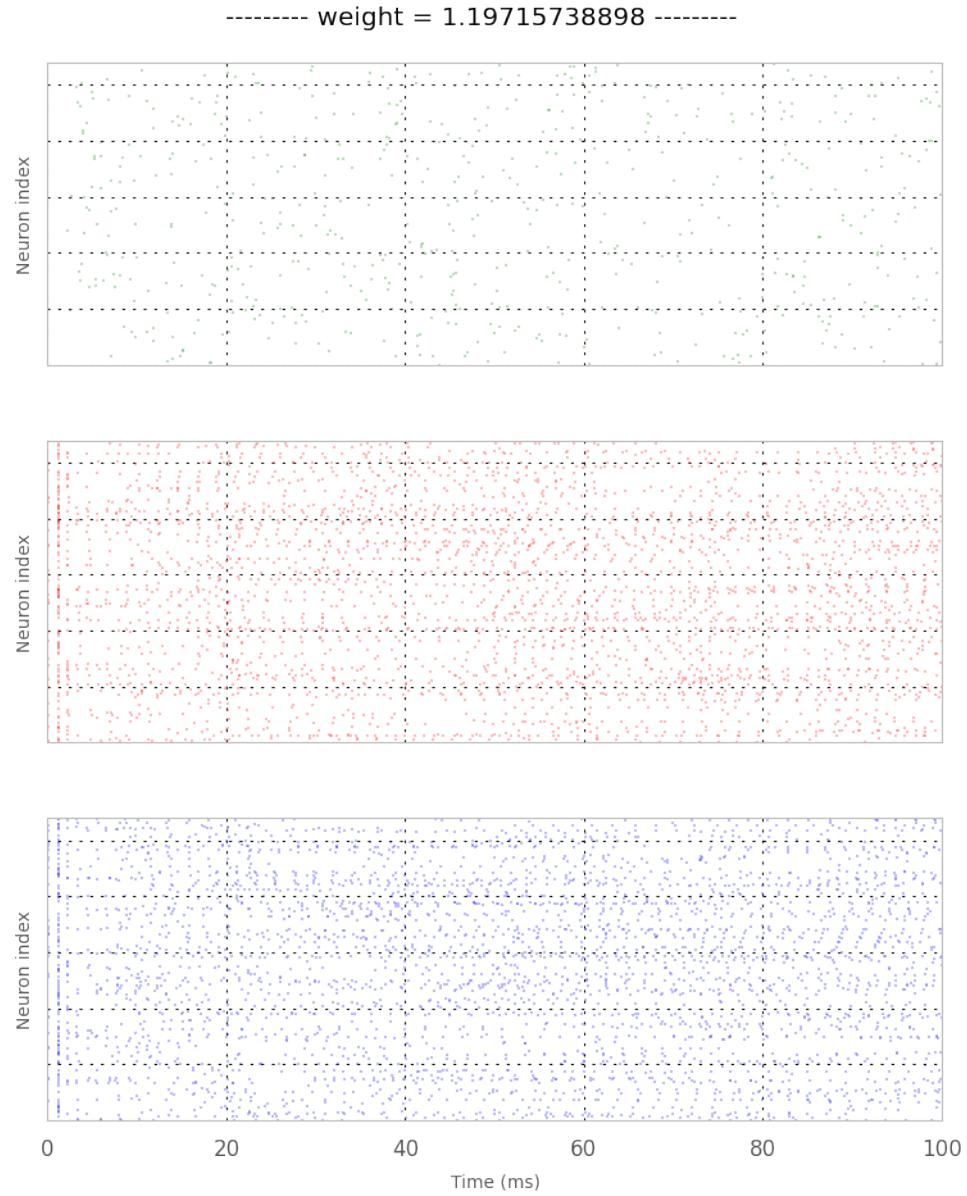


----- weight = 0.6 -----



----- weight = 0.847522526774 -----





2.2.8 Courbe de taux de décharge en fonction des poids synaptiques

Une fois l'effet d'une variation du poids global W observé, des tests plus spécifiques doivent être effectués. Hormis les connexions de la source au réseau, nous rappelons que le modèle comporte quatre projections. Soient E et I , la population excitatrice et inhibitrice l'ensemble des poids entre ces deux populations est : $\{W_{EI}, W_{EE}, W_{IE}, W_{II}\}$. L'effet des poids de chacune des quatre projections sur le taux de décharge des neurones du réseau est étudié.

Pour chaque type de connexion, plusieurs simulations du modèle sont lancés avec différentes valeurs du poids synaptique des connexions de ce type. Et pour chaque simulation, le taux de décharge neuronal moyen des populations E et I est récupéré. Les résultats sont alors affichés dans une courbe de taux de décharge en fonction d'une variation de poids synaptique.

Nous remarquons que l'augmentation de W_{EI} comme W_{IE} , le poids des connexions latérales, induit une diminution du taux de décharge en sortie du réseau. Aussi, l'augmentation de W_{EE} et W_{II} , le poids des connexions récurrentes, provoque l'effet inverse.

```
>>> import numpy as np
... from RRNN import RRNN
...
... n_sim_each, time = 25, 1000
... net = RRNN(time=time)

CSAConnector: libneurosim support not available in NEST.
Falling back on PyNN's default CSAConnector.
Please re-compile NEST using --with-libneurosim=PATH
/usr/local/lib/python2.7/site-packages/matplotlib/__init__.py:1318: UserWarning:
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

/usr/local/lib/python2.7/site-packages/NeuroTools/__init__.py:130: DependencyWarning:
To have functions using interval please install the package.
website : http://pypi.python.org/pypi/interval/1.0.0
```

2.2.9 Courbes de taux de décharge en fonction des paramètres de sparseness

Les effets observés de la manipulation des poids des différentes projections génèrent des variations de flux d'activité entre les différentes populations du réseau. Mais ces flux ne dépendent pas seulement des poids synaptiques. Ils dépendent également du nombre de connexions existantes entre les populations.

Le simulateur utilisé permet de caractériser, de différentes manières, la façon de connecter les neurones de deux populations. Parmi les options proposées, se trouve un connecteur à probabilité fixe qui va connecter deux populations selon un paramètre de "sparseness". La "sparseness" est un paramètre définissant une probabilité de connexion synaptique entre les neurones de deux populations. Si la sparseness d'une projection est égale à 1, la projection est de type "all to all". Chaque projection possède donc un paramètre de "sparseness".

Ici, l'évolution du taux de décharge en fonction d'une variation de "sparseness" des différentes projections est étudiée. Ces analyses permettent d'apprécier la contribution de ces différents paramètres à l'activité.

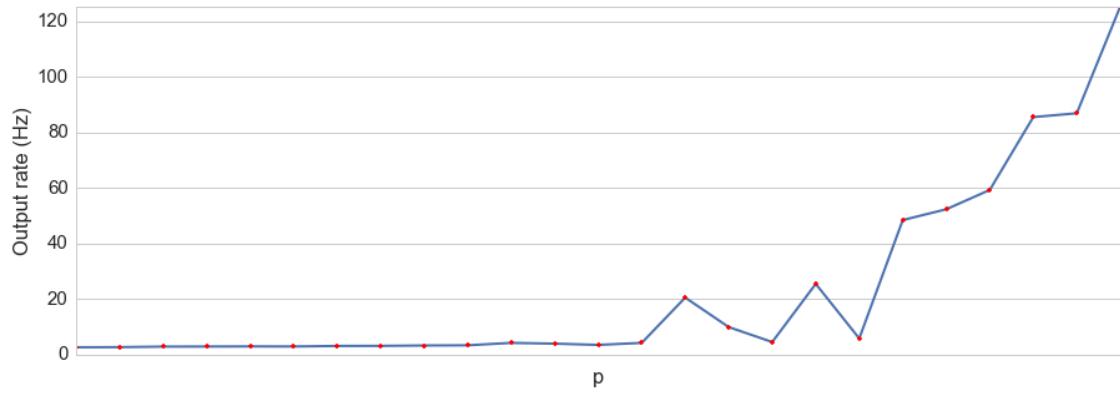
Comme cela a pu être fait pour d'autres paramètres, une courbe de taux de décharge en fonction d'une variation de sparseness est générée pour chaque projection.

L'augmentation de la probabilité de connexion pour les projections EI et IE (latérales) provoque une diminution du taux de décharge. Alors que la même manipulation pour les projections EE et II (recurrentes) provoque une augmentation du taux de décharge.

Les résultats sont similaires à ceux obtenus avec la variation des poids synaptiques, ce qui amène à considérer la modification de la "sparseness" ou des poids d'une même projection, comme une manière ou une autre de manipuler le flux d'activité entre les populations que cette projection relie. C'est pourquoi dans la suite de ce projet, une manipulation du flux d'activité d'une population A à une autre population B sera implémentée simplement par une modification du poids de la projection AB.

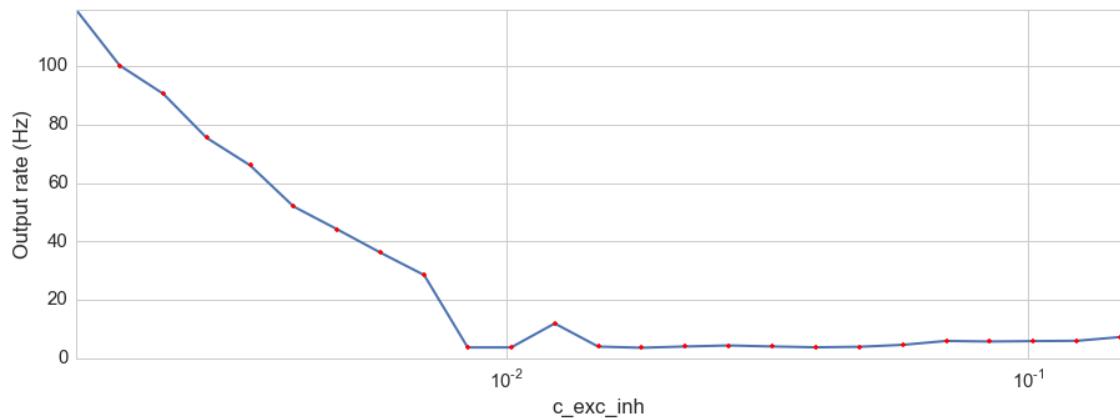
```
>>> import numpy as np
... n_sim_each, time = 25, 100
... from RRNN import RRNN
... net = RRNN(time = time)
...
... sim_params = net.sim_params
... cell_params = net.cell_params
... sim_list = [
...     ('p', sim_params['p'] * np.logspace(-.2, .2, n_sim_each)),
...     ('c_exc_inh', sim_params['c_exc_inh'] * np.logspace(-1, 1, n_sim_each)),
...     ('c_inh_exc', sim_params['c_inh_exc'] * np.logspace(-1, 1, n_sim_each)),
...     ('c_exc_exc', sim_params['c_exc_exc'] * np.logspace(-1, 1, n_sim_each)),
...     ('c_inh_inh', sim_params['c_inh_inh'] * np.logspace(-1, 1, n_sim_each))
... ]
...
... net.paramRole(sim_list, f_rate_max=None, datapath='/tmp/RRNN_topo1' + ta
----- p -----
0    0.315479
1    0.327821
2    0.340646
3    0.353973
4    0.367821
5    0.382211
6    0.397164
7    0.412702
8    0.428848
9    0.445625
10   0.463059
11   0.481175
12   0.500000
13   0.519561
14   0.539888
```

```
15    0.561009
16    0.582957
17    0.605764
18    0.629463
19    0.654089
20    0.679678
21    0.706269
22    0.733900
23    0.762611
24    0.792447
Name: p, dtype: float64
```



```
----- c_exc_inh -----
0    0.001500
1    0.001817
2    0.002202
3    0.002667
4    0.003232
5    0.003915
6    0.004743
7    0.005747
8    0.006962
9    0.008435
10   0.010219
11   0.012381
12   0.015000
13   0.018173
14   0.022017
15   0.026674
16   0.032317
```

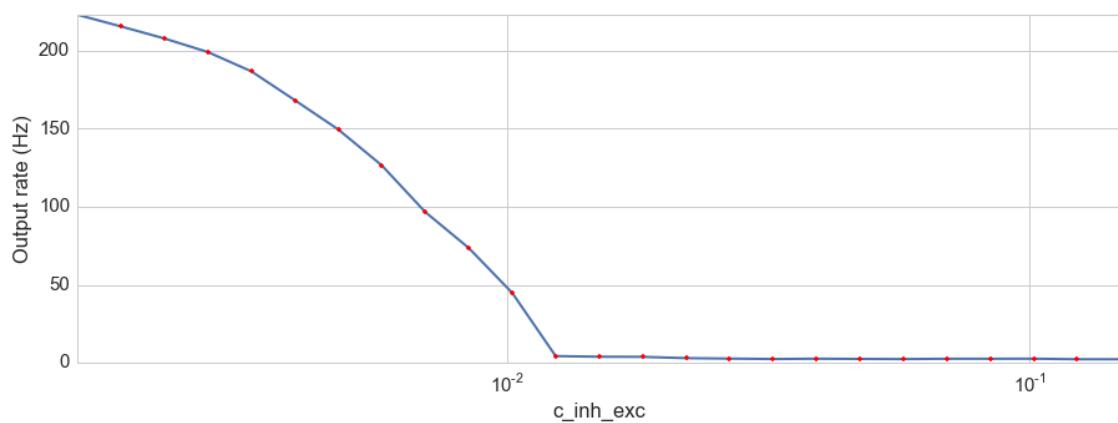
```
17    0.039152
18    0.047434
19    0.057468
20    0.069624
21    0.084351
22    0.102194
23    0.123811
24    0.150000
Name: c_exc_inh, dtype: float64
```



----- c_inh_exc -----

```
0    0.001500
1    0.001817
2    0.002202
3    0.002667
4    0.003232
5    0.003915
6    0.004743
7    0.005747
8    0.006962
9    0.008435
10   0.010219
11   0.012381
12   0.015000
13   0.018173
14   0.022017
15   0.026674
16   0.032317
17   0.039152
```

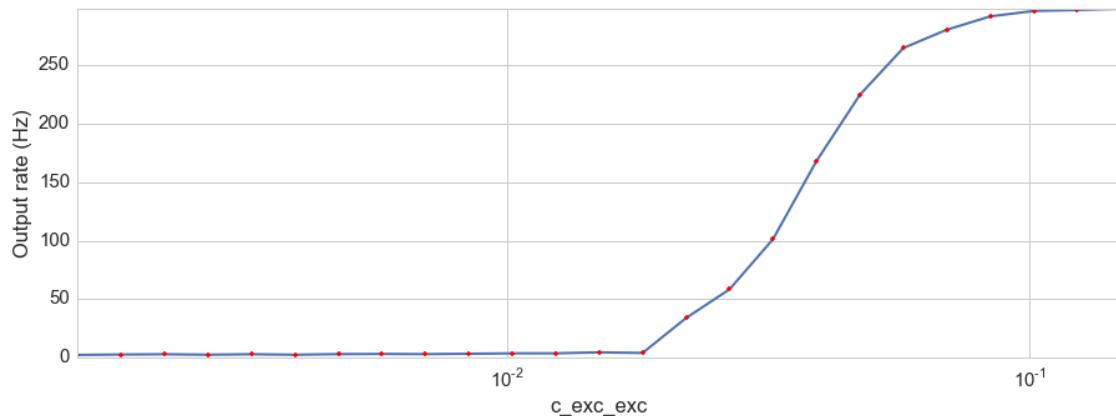
```
18    0.047434
19    0.057468
20    0.069624
21    0.084351
22    0.102194
23    0.123811
24    0.150000
Name: c_inh_exc, dtype: float64
```



----- c_exc_exc -----

```
0      0.001500
1      0.001817
2      0.002202
3      0.002667
4      0.003232
5      0.003915
6      0.004743
7      0.005747
8      0.006962
9      0.008435
10     0.010219
11     0.012381
12     0.015000
13     0.018173
14     0.022017
15     0.026674
16     0.032317
17     0.039152
18     0.047434
```

```
19      0.057468
20      0.069624
21      0.084351
22      0.102194
23      0.123811
24      0.150000
Name: c_exc_exc, dtype: float64
```



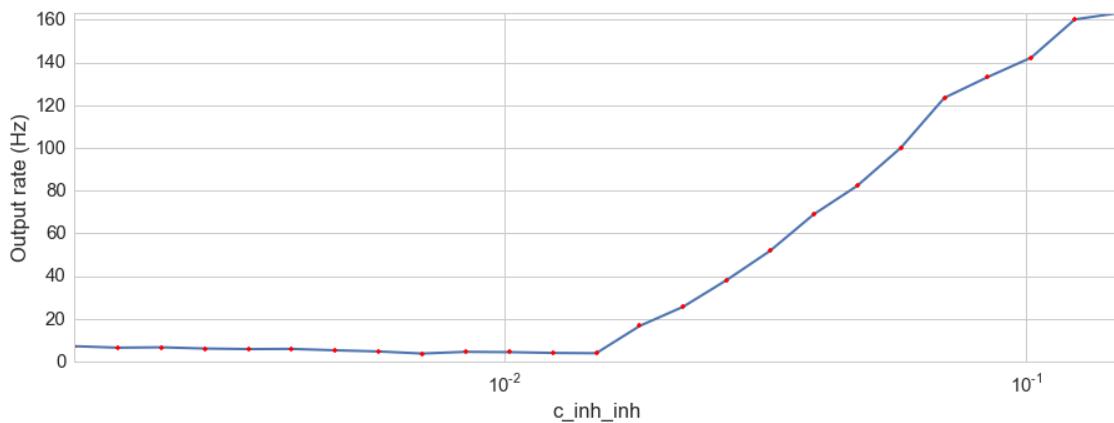
----- c_inh_inh -----

0	0.001500
1	0.001817
2	0.002202
3	0.002667
4	0.003232
5	0.003915
6	0.004743
7	0.005747
8	0.006962
9	0.008435
10	0.010219
11	0.012381
12	0.015000
13	0.018173
14	0.022017
15	0.026674
16	0.032317
17	0.039152
18	0.047434
19	0.057468

```

20    0.069624
21    0.084351
22    0.102194
23    0.123811
24    0.150000
Name: c_inh_inh, dtype: float64

```



2.3 Les états du réseau

Brunel décrit quatre états du RRNN :

- Synchronous Regular : Les activations neuronales sont synchrones et les intervalles inter-décharges pour chaque neurone sont identiques
- Synchronous Irregular : Les activations neuronales sont synchrones et les intervalles inter-décharges sont différents
- Asynchronous Regular : Les neurones ne s'activent pas en même temps et les intervalles inter-décharges sont identiques
- Asynchronous Irregular : Les neurones ne s'activent pas en même temps et les intervalles inter-décharges sont différents

Il fait également référence à un paramètre de coupling g , qui régulerait le poids de l'inhibition par rapport à celui de l'excitation. La modification d'un tel paramètre permettrait une transition entre les différents états du réseau [?].

Nous faisons alors des suppositions sur la définition de g , et nous observons l'effet de g sur le comportement du RRNN, pour chacune de ces suppositions.

2.3.1 Définir le coupling

La réelle définition du paramètre de coupling n'est pas évidente à appréhender. La seule certitude que nous avons est que le paramètre de coupling doit être un ratio de

certains poids par rapport au poids global W . C'est pourquoi différents ratio de poids synaptiques pouvant correspondre à g sont ici testés, à savoir :

- $g = W_{IE}, W_{II}/W$
- $g = W_{IE}, W_{EI}/W$
- $g = W_{EI}, W_{II}/W$

Chacune des trois définitions possibles de g correspond à des variations de poids de certaines projections par rapport à d'autres. Aussi, il a précédemment été fait mention d'une méthode permettant d'obtenir une courbe de taux de décharge en fonction de la variation de paramètre. Une méthode similaire est ici utilisée, mais cette fois, les taux de décharge sont calculés à partir d'une covariation de deux paramètres de poids synaptique et ce, pour chacune des définitions de g .

Il semble que si g est défini par le ratio du poids des connexions latérales par rapport à W , l'activité du réseau diminue quand g croît. Ce qui n'est pas sans rappeler les observations effectuées du taux de décharge lors de la manipulation des poids des différentes projections internes au réseau.

2.3.2 Rôle du coupling

Ainsi, faire varier g devrait provoquer une transition de phase des états d'activité du réseau. Pour observer un tel comportement, la mesure du taux de décharge neuronal, bien qu'elle aie l'avantage d'amener à des observations quantitatives, n'est pas suffisamment informative. C'est pourquoi une méthode qualitative, mais néammoins plus pertinente dans ce contexte, comme la génération de rasterplot, est utilisée.

Une simulation du RRNN est exécutée pour chaque valeur prise par g . Les spikes des neurones des populations source, excitatrice et inhibitrice sont récupérés et affichés dans le rasterplot. Ainsi, il est possible de savoir quels neurones ont déchargés et quand.

Un changement brusque du comportement du réseau est observé lorsque g dépasse une certaine valeur. Par la suite, l'évolution de l'état du réseau se fait de manière continue.

```
>>> import numpy as np
... import matplotlib.pyplot as plt
... import pandas as ps
... from RRNN import RRNN
...
... n_sim_each, time = 15, 1000
... n_sim_each, time = 5, 100
... net = RRNN(time=time)
...
... # j'ai commenté ça car ça ne teste pas autour du point central de nos pa
... net.setParams(['w_exc_inh', 'w_exc_exc', 'w_inh_exc', 'w_inh_inh'],
...               [net.w, net.w, net.w, net.w])
... #
... #Cette ligne est là pour m'assurer que l'on teste bien ce que l'on veut
... #(le coefficient g est aussi dans le modèle). Cela me permet de garder
... #le modèle tel qu'il est actuellement
```

```

CSAConnector: libneurosim support not available in NEST.
Falling back on PyNN's default CSAConnector.
Please re-compile NEST using --with-libneurosim=PATH
/usr/local/lib/python3.5/site-packages/matplotlib/__init__.py:1350: UserWarning:
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

```

2.3.3 L'état balancé

Nous avons précédemment évoqué les différents états de l'activité du RRNN. L'état balancé que nous recherchons correspond à l'état Asynchronous Irregular. Il correspond à un équilibre entre l'excitation et l'inhibition. Un tel équilibre a fait l'objet d'études théoriques qui montrent qu'un RRNN balancé produit une activité bruitée, proche de celle pouvant être générée par un processus de Poisson. Aussi, la réponse d'un RRNN balancé à une entrée est linéaire et plus rapide que l'intégration d'information pouvant être effectuée par un seul neurone [?]. Ces propriétés sont intéressantes pour la modélisation de réseaux de neurones corticaux, tels que le cortex visuel primaire. En effet, nous rappelons que la sélectivité à l'orientation au sein des pinwheels et chez le rat est possible si le réseau est dans cet état.

Maintenant que des outils ont été développés pour observer des variations de comportement du RRNN, la recherche d'états balancés du réseau peut débuter. Nous introduisons à présent des indices utiles pour caractériser des aspects de l'activité :

- $\frac{dF}{dT}$ est la dérivée du taux de décharge en sortie par rapport au taux de décharge de la population source. Cet indice correspond au gain du réseau produit par les propriétés cellulaires et les propriétés des connexions entre les populations.
- CV est le coefficient de variation de l'intervalle inter-décharge. Il est défini par le ratio de l'écart-type de la distribution des intervalles sur leur moyenne. C'est une mesure de la variabilité des intervalles inter-décharges.

$$CV = \frac{\sqrt{\langle T^2 \rangle - \langle T \rangle^2}}{\langle T \rangle}$$

Nous implémentons également une méthode d'optimisation permettant d'obtenir des paramètres de poids et de coupling satisfaisant certaines contraintes. Les indices évoqués plus haut sont autant de contraintes devant être satisfaites.

2.3.4 Optimisation du coupling g

Ici, nous tentons de trouver la valeur de coupling qui satisfait des contraintes de CV et de $\frac{dF}{dT}$ traduisant un état AI (i.e. l'état balancé) du réseau.

L'algorithme d'optimisation est exécuté sur une variation de coupling. Une figure qui nous permet d'estimer la valeur de g maximisant le CV et $\frac{dF}{dT}$ est alors générée. La valeur de g obtenue est ensuite fournie au modèle afin de pouvoir évaluer l'excitabilité d'un RRNN balancé.

```

>>> from RRNN import RRNN
... import numpy as np
... import matplotlib.pyplot as plt
... import pandas as ps
... import os

CSAConnector: libneurosim support not available in NEST.
Falling back on PyNN's default CSAConnector.
Please re-compile NEST using --with-libneurosim=PATH
/usr/local/lib/python2.7/site-packages/matplotlib/__init__.py:1318: UserWarning:
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

/usr/local/lib/python2.7/site-packages/NeuroTools/__init__.py:130: DependencyWarning:
To have functions using interval please install the package.
website : http://pypi.python.org/pypi/interval/1.0.0

```

```

>>> net = RRNN(time = 1000)

>>> net.multiOptimisation(np.linspace(0, 10, 30)*net.w, 'g')

0      9.549926
1      10.000000
2      10.471285
Name: input_rate, dtype: float64
0      9.549926
1      10.000000
2      10.471285
Name: input_rate, dtype: float64
0      9.549926
1      10.000000
2      10.471285
Name: input_rate, dtype: float64
0      9.549926
1      10.000000
2      10.471285
Name: input_rate, dtype: float64
0      9.549926
1      10.000000
2      10.471285
Name: input_rate, dtype: float64
0      9.549926

```

```
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
```

```
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
Name: input_rate, dtype: float64
0    9.549926
1    10.000000
2    10.471285
```

```
Name: input_rate, dtype: float64
0    9.549926
1   10.000000
2   10.471285
Name: input_rate, dtype: float64
0    9.549926
1   10.000000
2   10.471285
Name: input_rate, dtype: float64
```

KeyError

Traceback (most recent call last)

```
<ipython-input-5-9a1719e6915b> in <module>()
----> 1 net.multiOptimisation(np.linspace(0, 10, 30)*net.w, 'g')

/Users/davidarbib/BalaV1/DetectingOrientations/Stage M2 David Arbib,
585         df = self.g_dFoverdI(values)
586         df.to_pickle(filename)
--> 587         print(self.value_minCost(df, len(values), 'w_inh_ex')
588 #----- Sparseness -----
589         elif c_or_w == 'c' :

/Users/davidarbib/BalaV1/DetectingOrientations/Stage M2 David Arbib,
619         dI0, dI1, dI2 = np.array(df['input_rate'])[0], np.array(
620             fr = np.array(df['m_f_rate'])
--> 621             cv = np.array(df['cv'])
622             cv = cv.reshape((n, 3))
623             g = np.array(df[var])

/usr/local/lib/python2.7/site-packages/pandas/core/frame.pyc in __ge__(self, other, axis=0, **kwargs)
1795         return self._getitem_multilevel(key)
1796     else:
-> 1797         return self._getitem_column(key)
1798
1799     def _getitem_column(self, key):
```

/usr/local/lib/python2.7/site-packages/pandas/core/frame.pyc in __getitem__(self, key, **kwargs)

```
1802         # get column
1803         if self.columns.is_unique:
-> 1804             return self._get_item_cache(key)
1805
1806         # duplicate columns & possible reduce dimensionaility

/usr/local/lib/python2.7/site-packages/pandas/core/generic.pyc in __
1082             res = cache.get(item)
1083             if res is None:
-> 1084                 values = self._data.get(item)
1085                 res = self._box_item_values(item, values)
1086                 cache[item] = res

/usr/local/lib/python2.7/site-packages/pandas/core/internals.pyc in __
2849
2850             if not isnull(item):
-> 2851                 loc = self.items.get_loc(item)
2852             else:
2853                 indexer = np.arange(len(self.items))[isnull(self.

/usr/local/lib/python2.7/site-packages/pandas/core/index.pyc in get_
1570         """
1571         if method is None:
-> 1572             return self._engine.get_loc(_values_from_object(key))
1573
1574         indexer = self.get_indexer([key], method=method)

pandas/index.pyx in pandas.index.IndexEngine.get_loc (pandas/index.
pandas/index.pyx in pandas.index.IndexEngine.get_loc (pandas/index.

pandas/hashtable.pyx in pandas.hashtable.PyObjectHashTable.get_item

pandas/hashtable.pyx in pandas.hashtable.PyObjectHashTable.get_item

KeyError: 'cv'
```

```
>>> net.multiOptimisation(np.linspace(4, 8, 30)*net.w, 'g')  
>>> net.multiOptimisation(np.linspace(4., 6., 30)*net.w, 'g')
```

Chapitre 3

Le Ring

Une fois le RRNN créé, paramétré et optimisé, nous lui ajoutons certaines propriétés dans le but de le transformer en “ring”. Ce dernier va nous permettre d’implémenter le modèle de la sélectivité à l’orientation reproduisant ce qui peut être observé au sein des colonnes corticales du cortex visuel primaire.

Le ring est un réseau récurrent disposant d’une certaine topologie. En effet, selon sa position dans le réseau, un neurone possède une certaine sélectivité à l’orientation et les connexions sont locales dans l’espace des orientations. Quelques propriétés de la réponse à l’orientation vont conditionner cette sélectivité et induire un certain comportement du réseau, en réponse à une orientation présentée sur son entrée :

- m est l’angle d’orientation préférée d’un neurone. Cela signifie que ce dernier aura une réponse maximale si une orientation d’un angle θ , tel que $\theta = m$, est présentée. Notons que le ring est construit de telle sorte que toutes les orientations sont codées avec une précision de vingt minutes d’arc et qu’il est non orienté, ainsi $0 \leq m \leq \pi$.
- la bandwidth σ est la largeur à mi-hauteur de la courbe d’accord d’un neurone. Elle sert à représenter la sélectivité de la réponse neuronale à d’autres orientations que celle préférée. Dans ce modèle, les bandwidth ne sont pas paramétrés par neurone mais plutôt par type de connexion entre les populations E et I. Nous implementons également une bandwidth dans les connexions entre la source et la population E. Ainsi, nous cherchons à ce que les neurones d’une colonne corticale aient une certaine bandwidth de sélectivité à l’orientation du fait de leurs connexions avec d’autres colonnes.

Une fonction d’accord est aussi implementée. Cette fonction permet de calculer le poids synaptique de chacune des connexions d’une projection à partir des propriétés décrites plus haut. De part sa généralité, nous utiliserons une loi de Von Mises (loi normale circulaire) définie par :

$$f(\theta) = \frac{1}{Z(\kappa)} \cdot e^{\kappa \cos(2(\theta - m))}$$

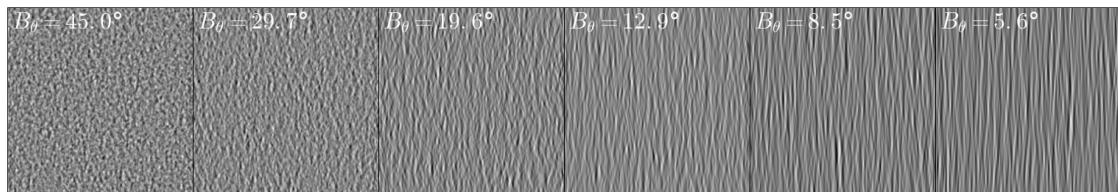
où Z est la fonction de normalisation. Par analogie avec la déviation standard d’une loi Gaussienne, on définit $\kappa = \frac{1}{\sigma^2}$. Notons que $f(\theta + \pi) = f(\theta)$.

```
>>> import numpy as np
```

```

... import MotionClouds as mc
... import matplotlib.pyplot as plt
... downscale = 1
... fx, fy, ft = mc.get_grids(mc.N_X/downscale, mc.N_Y/downscale, 16)
...
...
...
... N_theta = 6
... bw_values = np.pi*np.logspace(-2, -5, N_theta, base=2)
... fig_width = 21
...
...
... fig, axs = plt.subplots(1, N_theta, figsize=(fig_width, fig_width/N_theta))
... for i_ax, B_theta in enumerate(bw_values):
...     mc_i = mc.envelope_gabor(fx, fy, ft, V_X=0., V_Y=0.,
...                             theta=np.pi/2, B_theta=B_theta)
...     im = mc.random_cloud(mc_i)
...
...     axs[i_ax].imshow(im[:, :, 0], cmap=plt.gray())
...     axs[i_ax].text(5, 29, r'$B_\theta=%1f^\circ$' % (B_theta*180/np.pi), color='white')
...     axs[i_ax].set_xticks([])
...     axs[i_ax].set_yticks([])
... plt.tight_layout()
... fig.subplots_adjust(hspace = .0, wspace = .0, left=0.0, bottom=0., right=1.0)
...
... import os
... fig.savefig(os.path.join('../figs', 'orientation_tuning.png'))

```



3.1 Le ring non accordé

3.1.1 Effet de la bandwidth d'entrée dans un réseau non récurrent

Nous testons l'effet du changement de la bandwidth du signal d'entrée sur le comportement du réseau en désactivant toutes les projections sauf la projection d'entrée.

Pour cela, nous paramétrons l'activité des neurones de la population source de telle sorte que celle-ci représente une orientation de contraste de 90°. Une simulation du mo-

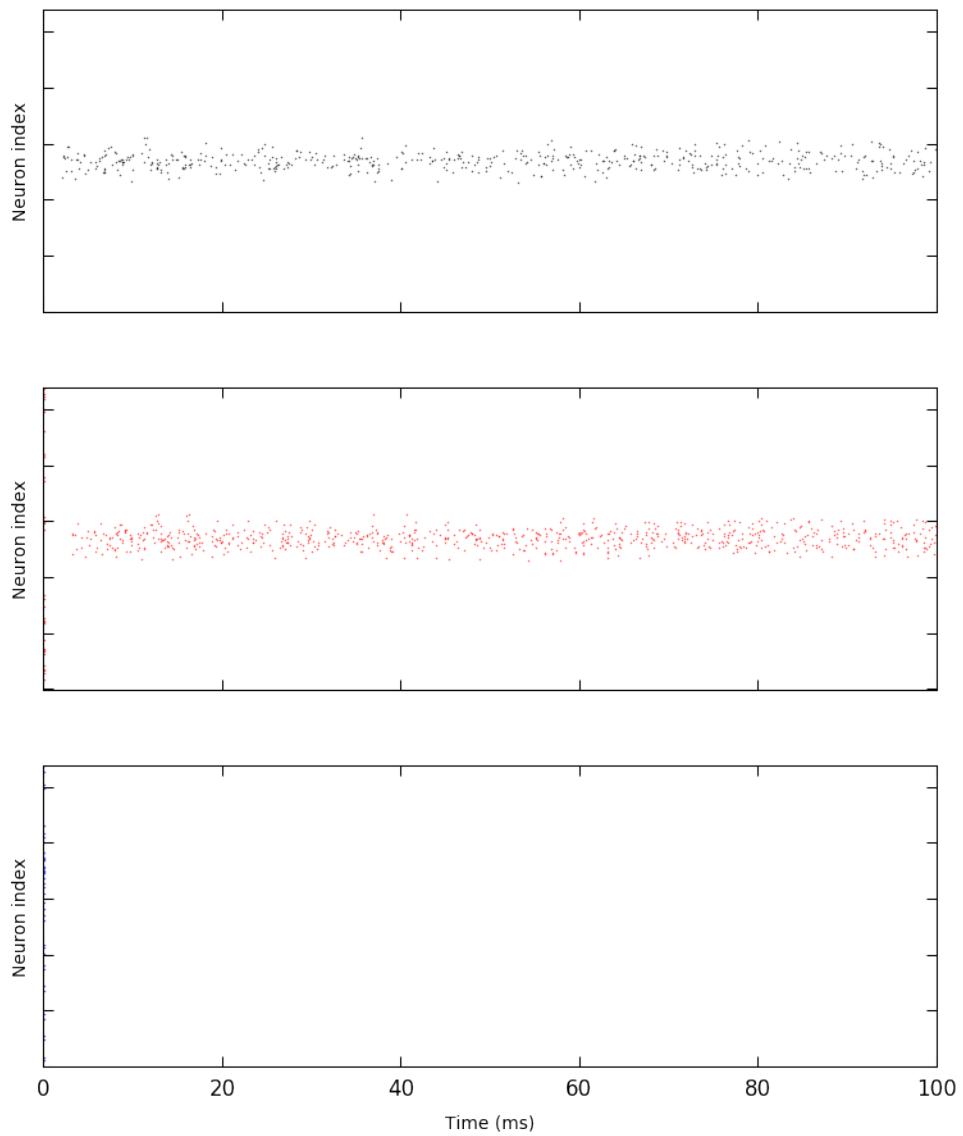
dèle est exécutée pour chaque valeur de bandwidth. Seule la projection de la source à la population E est active, W est nul. Des rasterplots des trois populations sont alors affichés.

Nous observons bien que certains neurones de la population E, ceux qui ont une réponse sélective à une orientation de ou proche de 90° sont actifs. Aussi, les neurones de la population I sont inactifs.

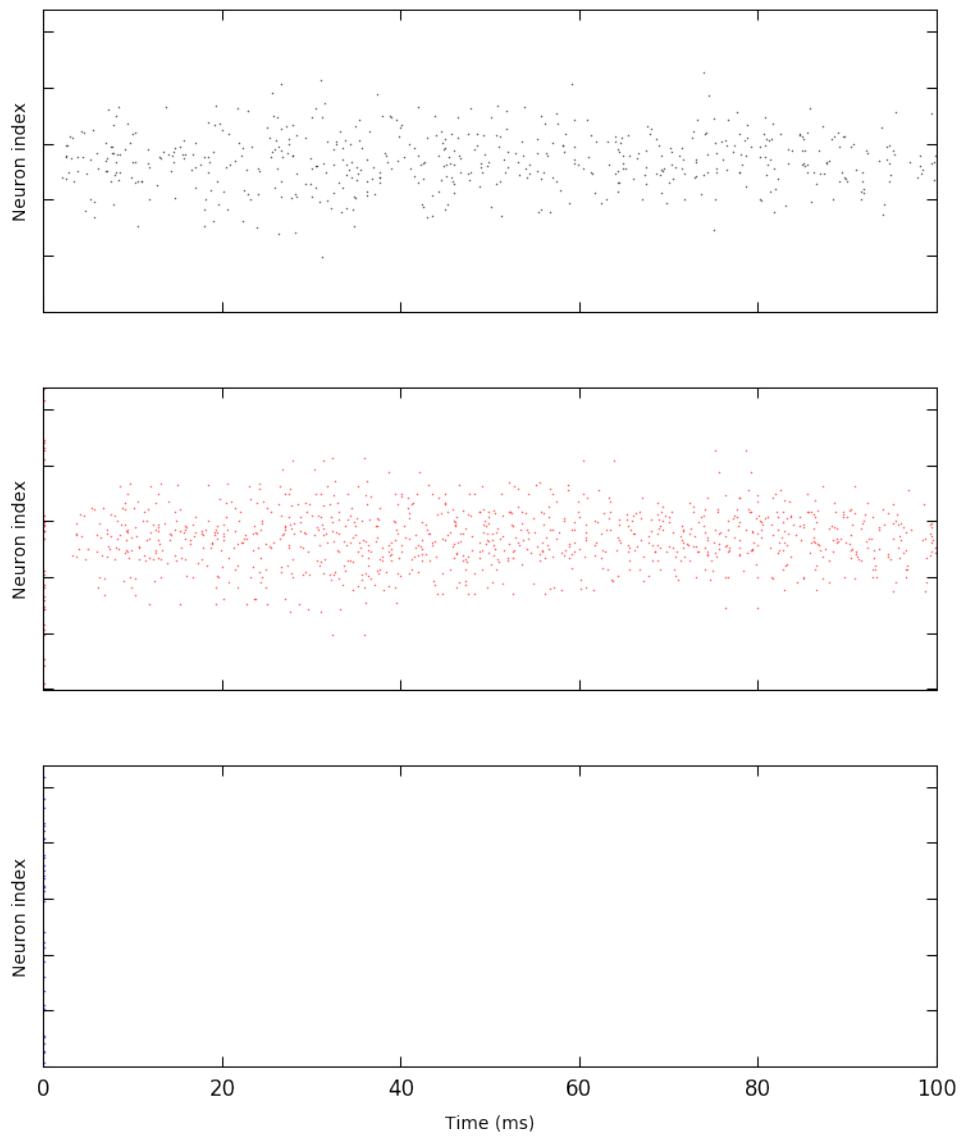
```
>>> from RRNN import RRNN
... import numpy as np
...
... net = RRNN(w=0.)
... _ = net.variationRaster('b_input', np.linspace(10, 180, 10))

CSAConnector: libneurosim support not available in NEST.
Falling back on PyNN's default CSAConnector.
Please re-compile NEST using --with-libneurosim=PATH
/usr/local/lib/python3.5/site-packages/matplotlib/__init__.py:1350: UserWarning:
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.
```

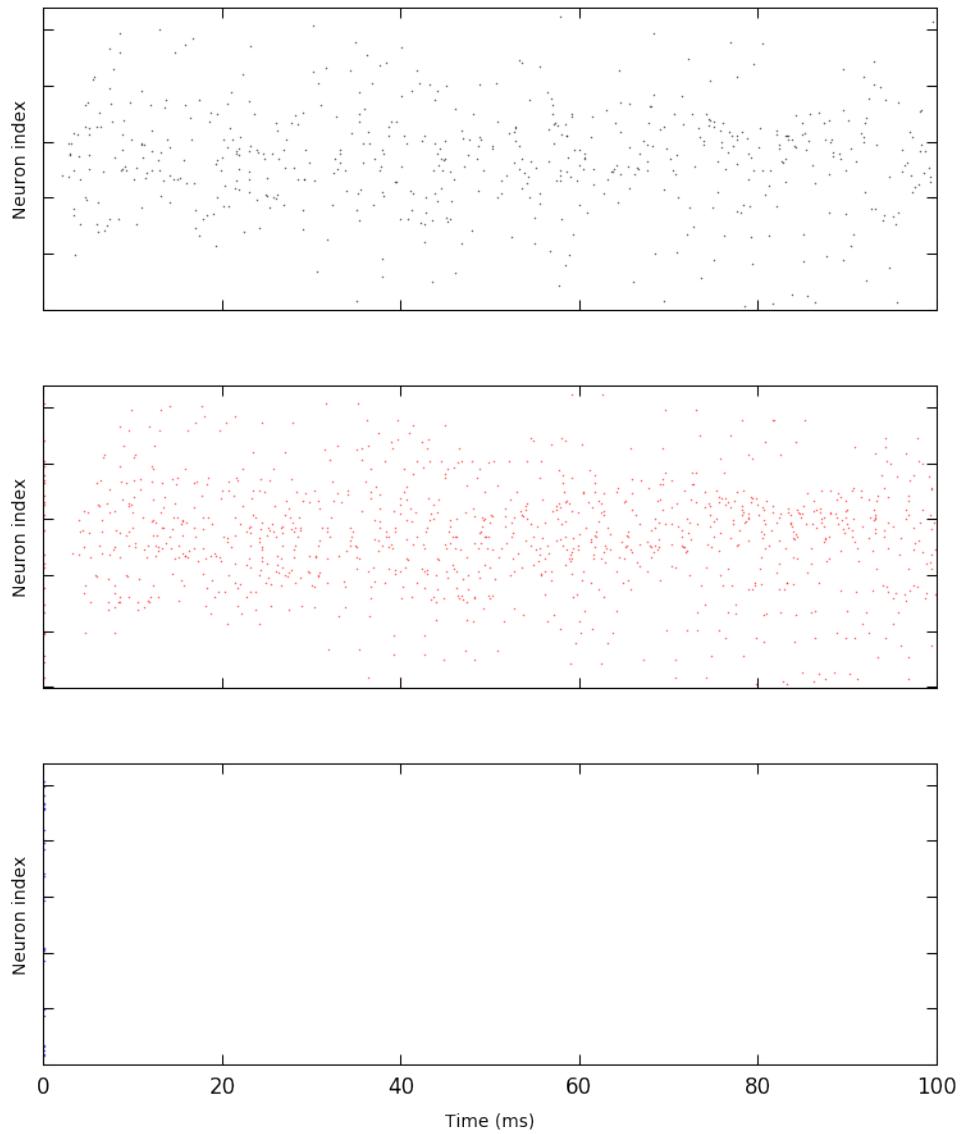
----- b_input = 10.0 -----



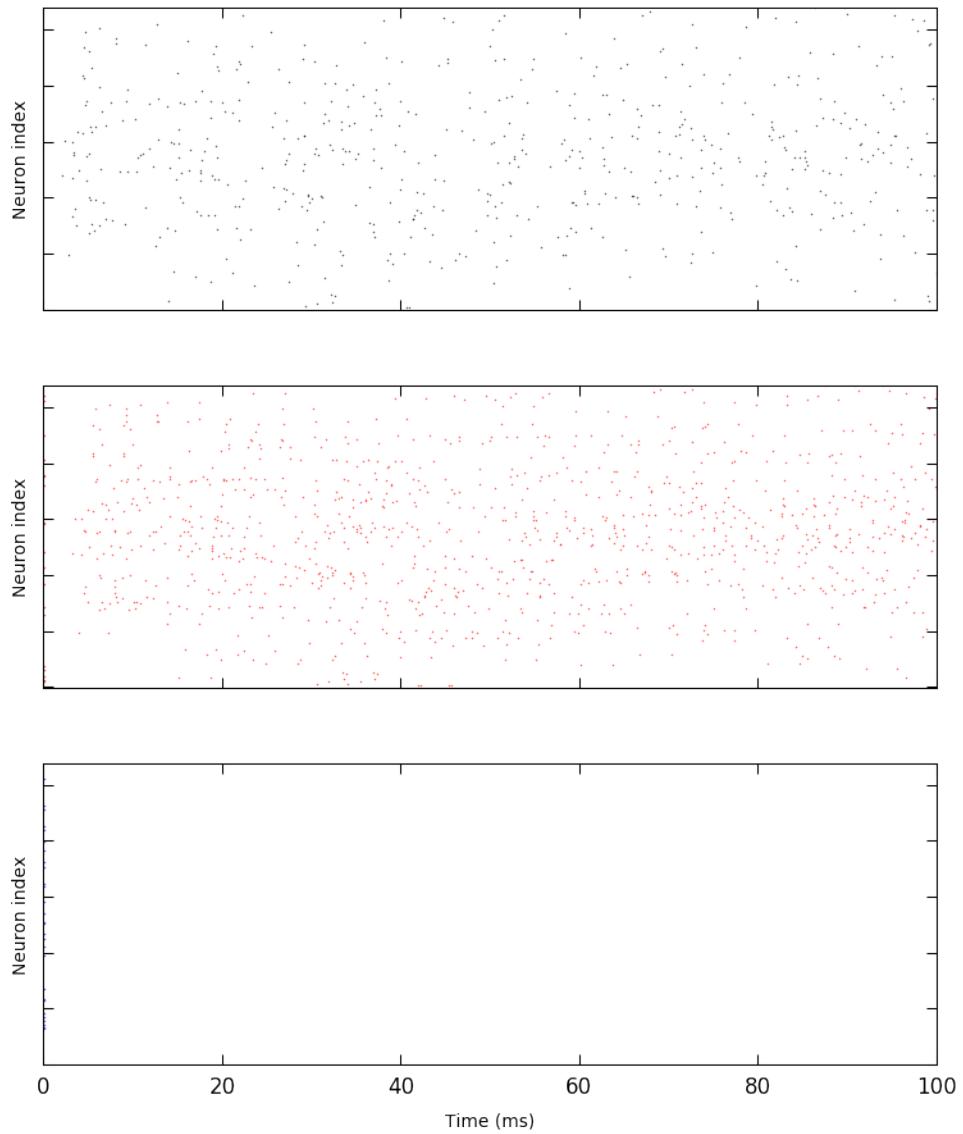
----- b_input = 28.88888888888889 -----



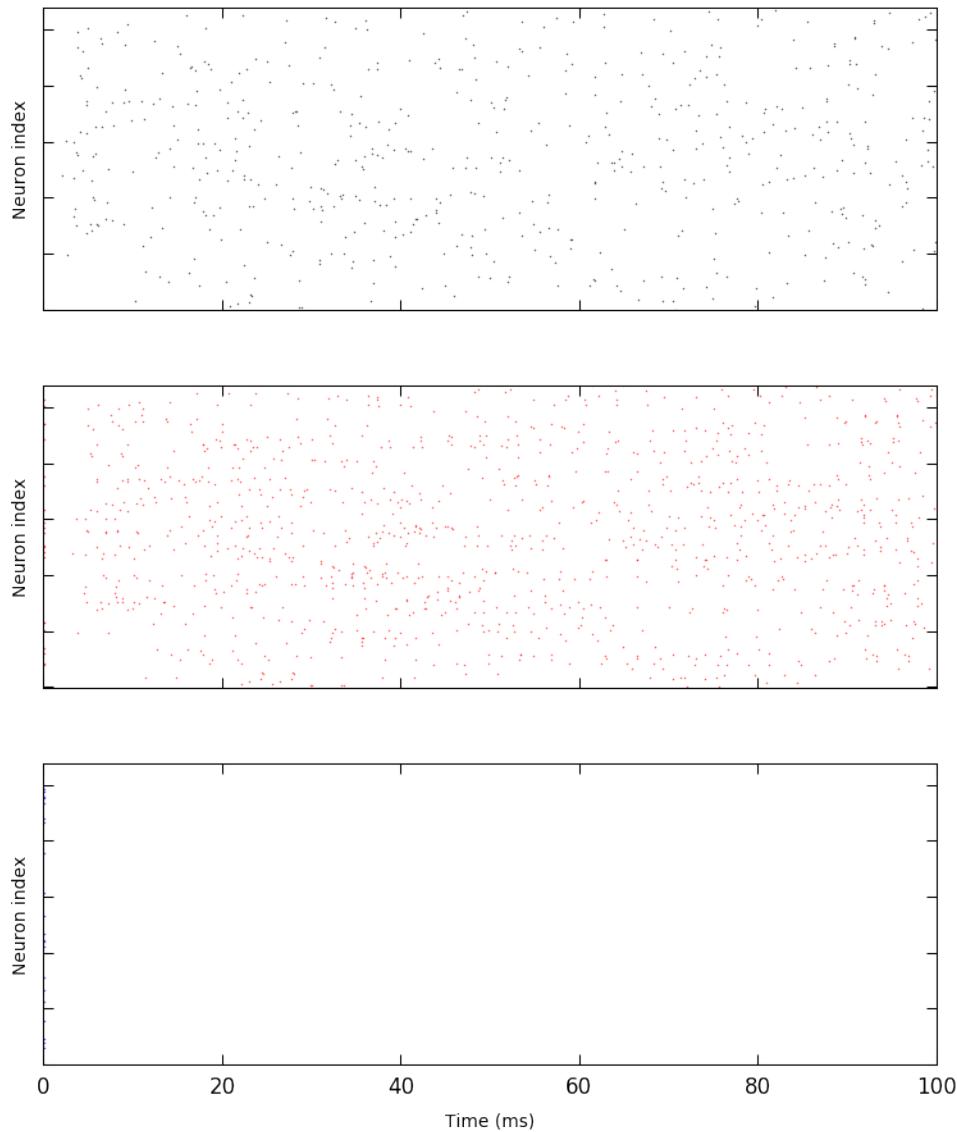
----- b_input = 47.77777777777778 -----



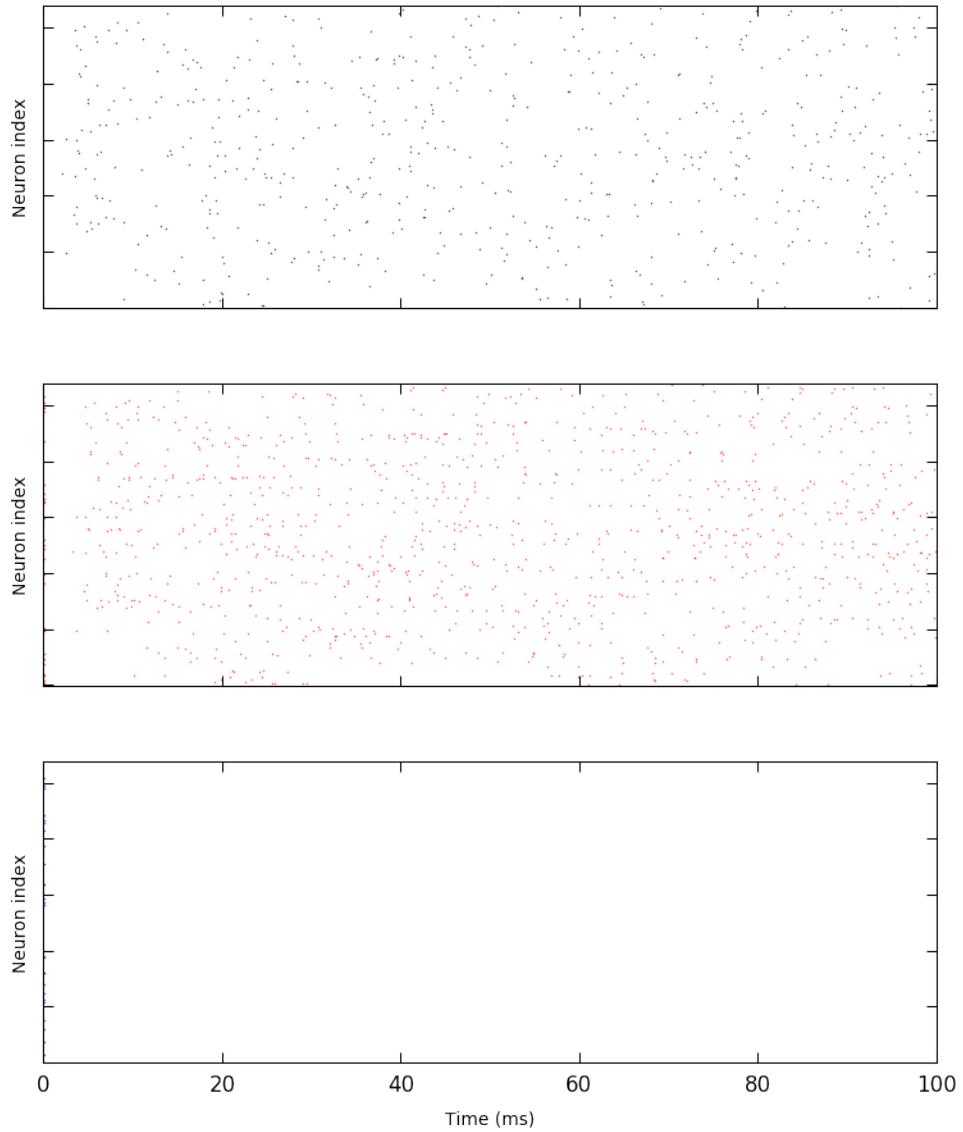
----- b_input = 66.66666666666667 -----



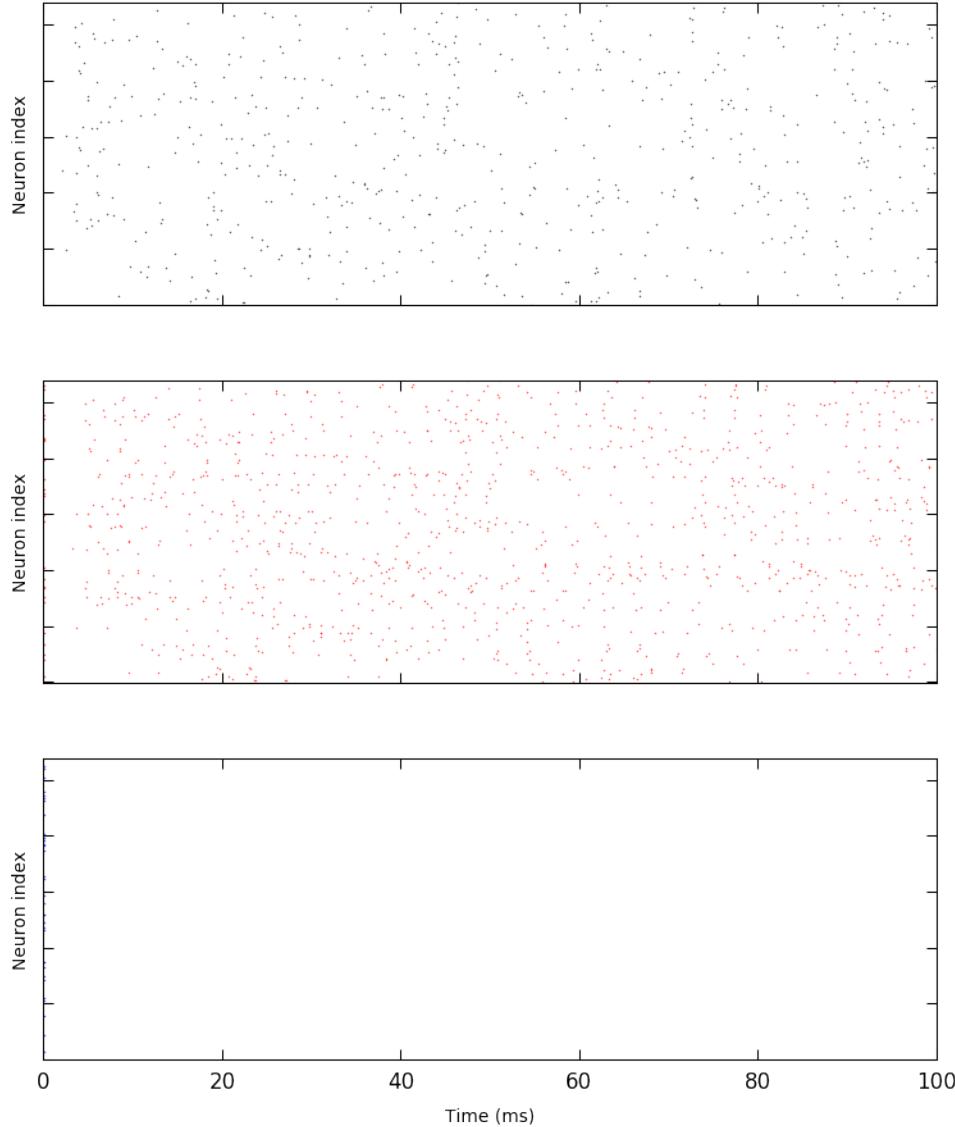
----- b_input = 85.55555555555556 -----



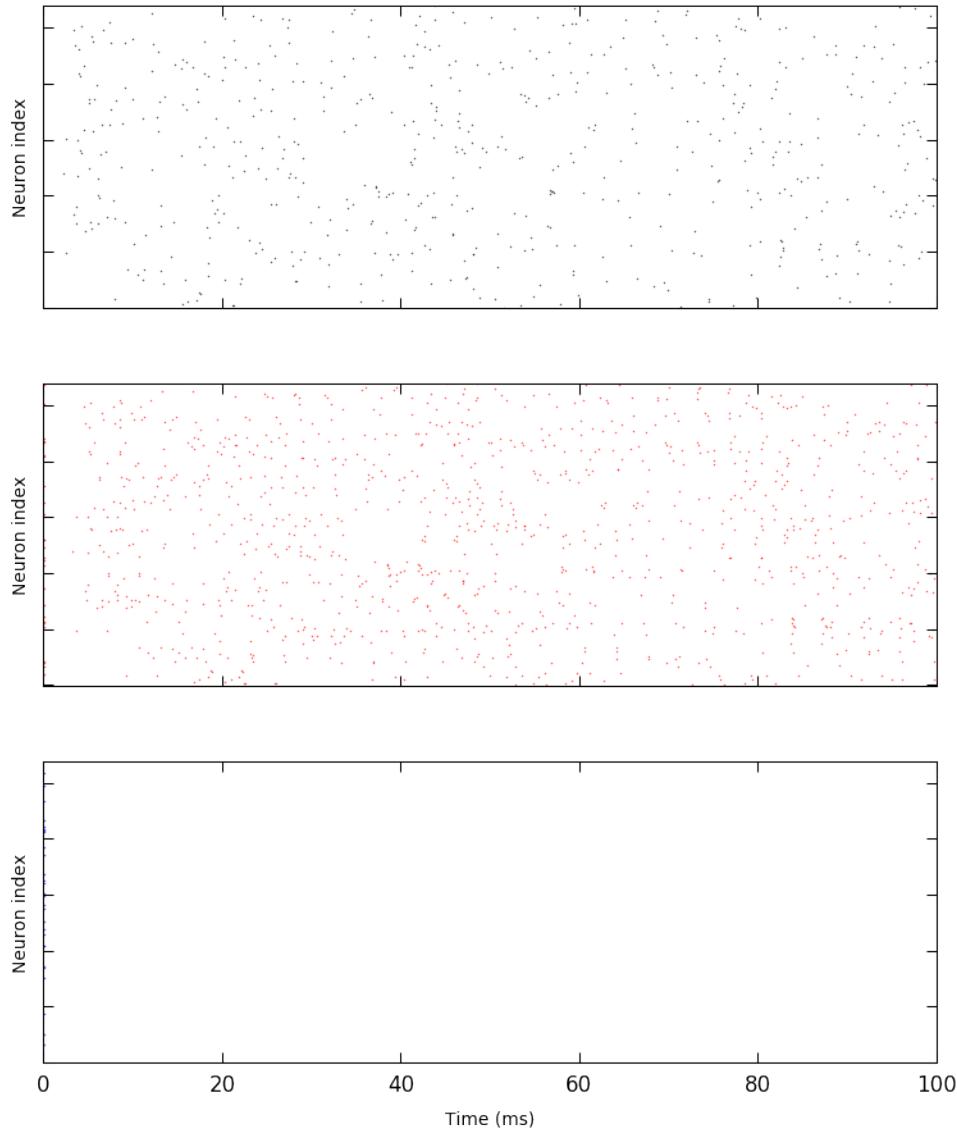
----- b_input = 104.44444444444444 -----



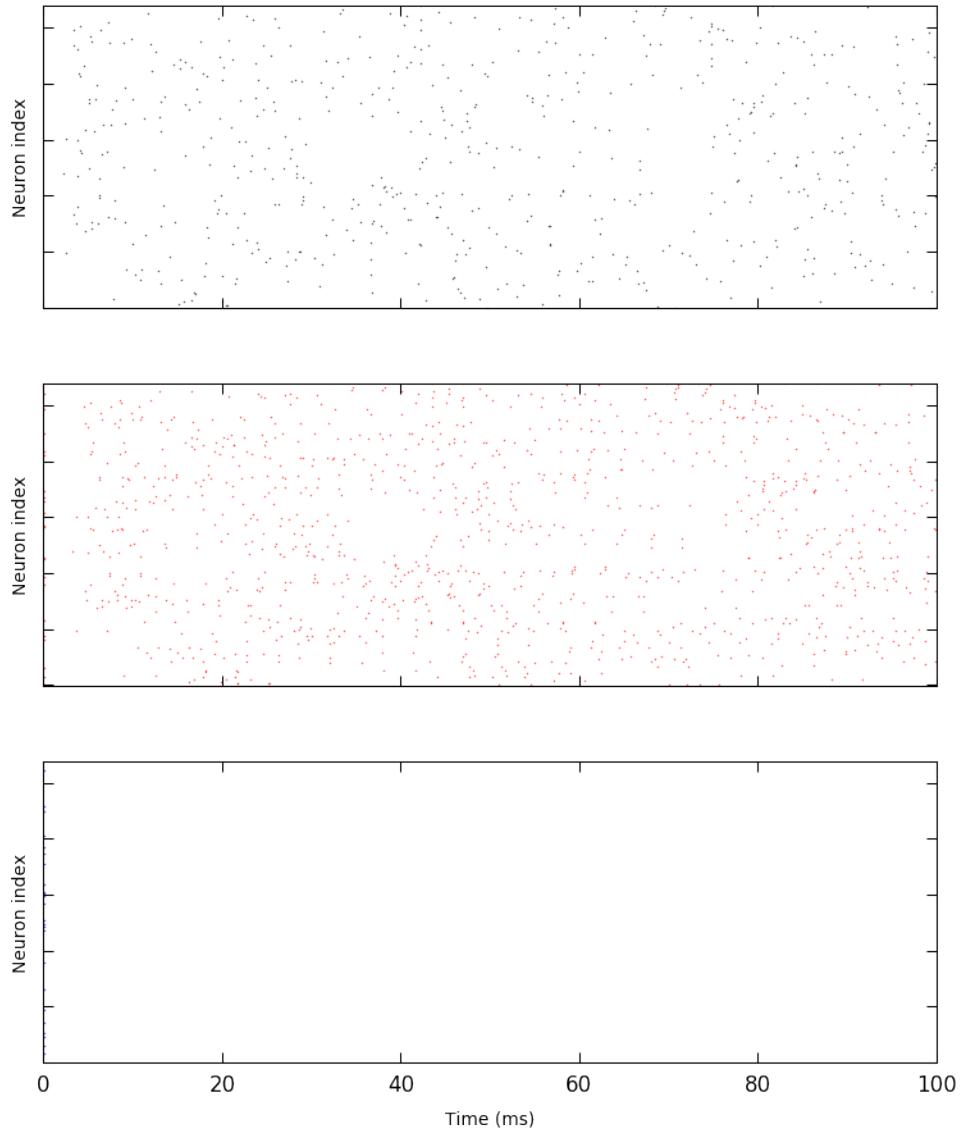
----- b_input = 123.3333333333334 -----

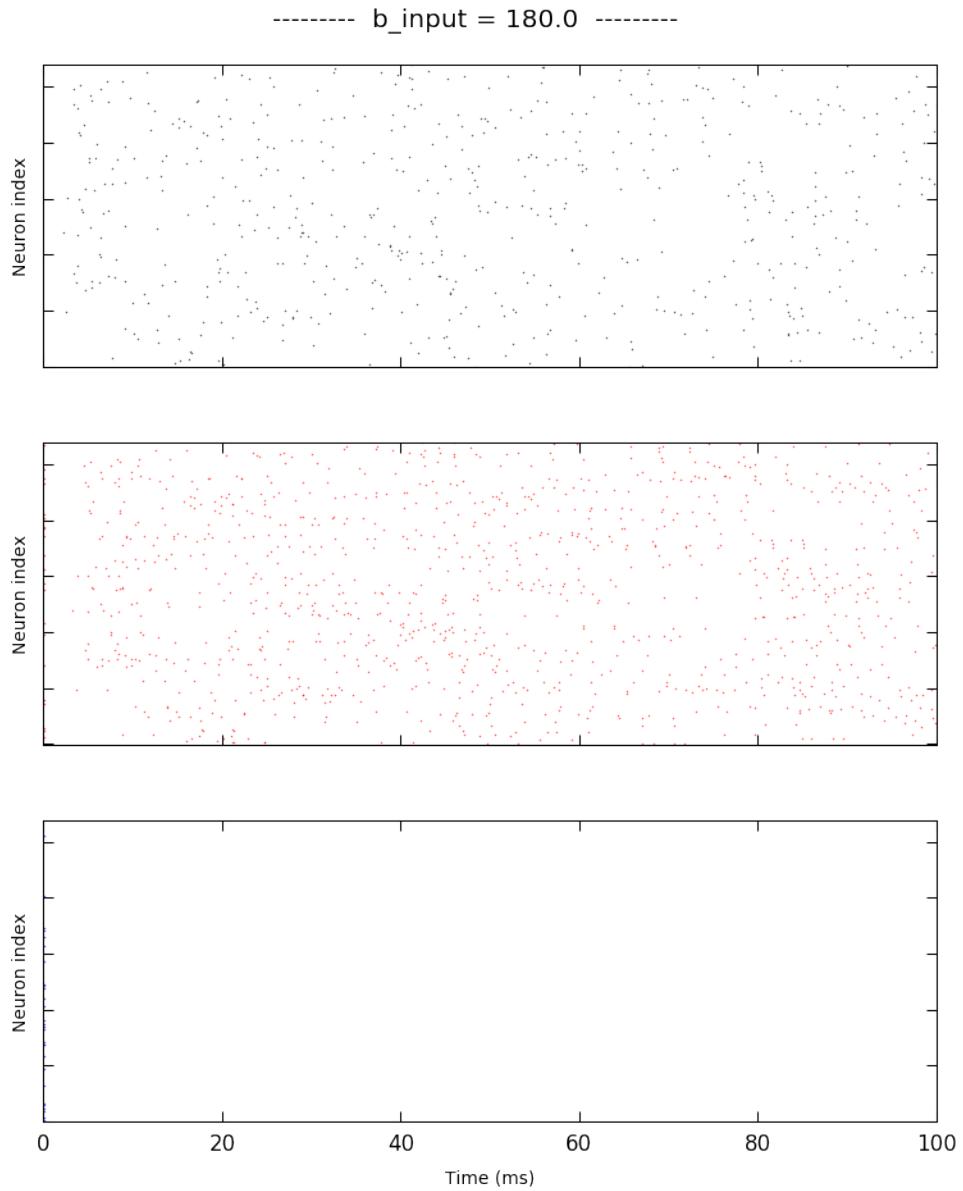


----- b_input = 142.2222222222223 -----



----- b_input = 161.1111111111111 -----





3.1.2 Effet de la bandwidth d'entrée

Nous avons étudié l'effet de la bandwidth du signal présenté en entrée dans le cas où les connexions du réseau, hormis les connexions entre la population source et la population E, sont désactivées. A présent, nous étudions son effet lorsque les poids synaptiques des connexions du ring sont homogènes.

Pour cela, nous paramétrons les connexions des quatres différentes projections de sorte qu'elles aient le même poids synaptique. Puis, une simulation du modèle est exécutée pour chaque valeur de bandwidth, et les rasterplots des trois populations sont alors

générés.

Nous observons que le poids W a un effet sur l'activité ou non de la population inhibitrice, mais n'a aucun effet sur le nombre de neurones actifs dans la population excitatrice.

3.1.3 Effet des bandwidths des projections internes

Après avoir étudié l'effet de la bandwidth de l'entrée sur un ring où les poids de toutes les connexions, sauf celles qui vont de la population source à la population excitatrice, sont identiques, nous cherchons maintenant à observer les effets d'une variation de bandwidth de chacune des quatre projections sur l'activité neuronale.

Pour cela, nous fixons la bandwidth de l'entrée à 15° et nous générerons des rasterplots pour chaque valeur de bandwidth d'une projection. Cette opération est répétée pour chacune des quatre projections.

Nous constatons que, concernant l'activité des neurones de la population inhibitrice, seule une modification de la bandwidth pour la projection allant de la population excitatrice à la population inhibitrice (EI) a un effet sur le nombre de neurones actifs.

3.1.4 Courbes d'accord d'un ring non accordé

(1 : quoi) Pour démontrer notre démarche, nous allons maintenant appliquer des entrées sélectives à un réseau de connectivité aléatoire, trouver la courbe de selectivité (de type von Mises) qui correspond à la meilleure courbe d'accord sur les fréquences de décharge.

(2 : comment) Pour cela, nous procédons à un ajustement de la réponse du réseau à une entrée sélective avec une loi de Von Mises.

(3 : résultat) On observe une transmission parfaite de l'information lorsque les poids internes sont nuls (couplage nul) puis graduellement une diffsuion de cette information. XXX Maintenant, on va étudier le comportement d'un réseau avec une connectivité en ring.

<http://docs.scipy.org/doc/scipy/reference/special.html#module-scipy.special>

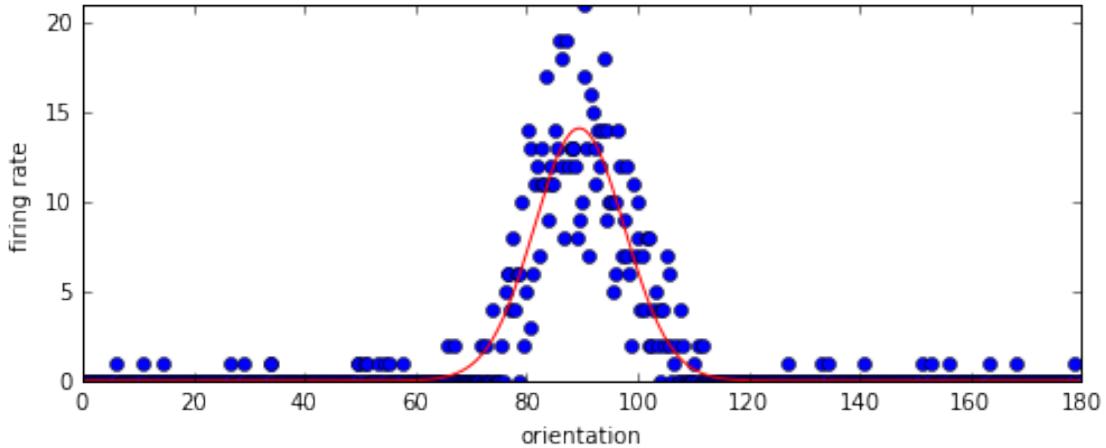
Maintenant, on TODO : "fitting network response with Von Mises, then fit en fonction de la bandwidth"

```
>>> net = RRNN(w=0.0)
... net.sim_params['b_input'] = 15.
...
... df, spikesE, spikesI = net.model()
...
... theta, fr, result = net.fit_vonMises(spikesE)
...
... fig, ax = plt.subplots(figsize=(8,3))
... ax.plot(theta*180/np.pi, fr, 'bo')
... #plt.plot(x, result.init_fit, 'k--')
... ax.plot(theta*180/np.pi, result.best_fit, 'r-')
... ax.axis('tight')
```

```

... ax.set_xlabel('orientation')
... ax.set_ylabel('firing rate')
... ax.set_ylim(0)
... plt.show()

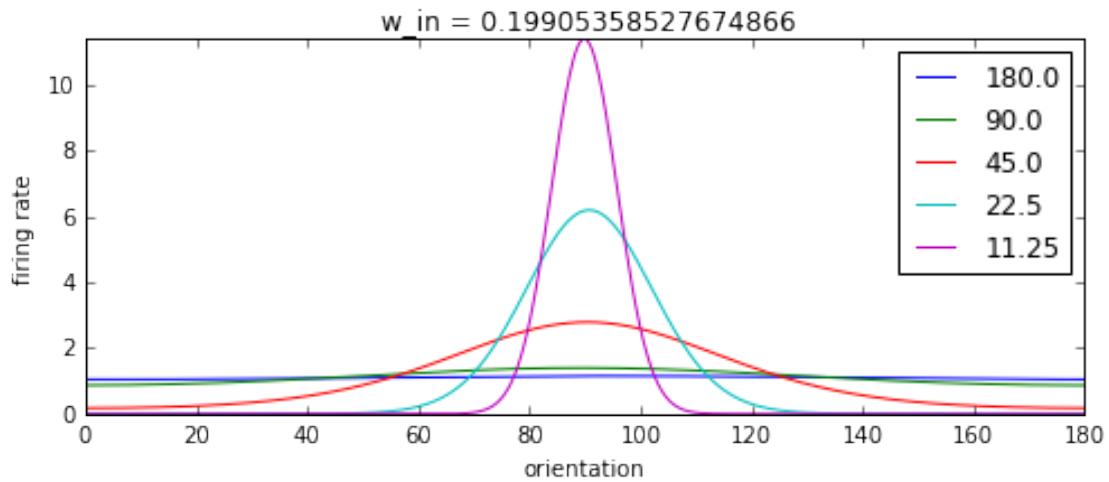
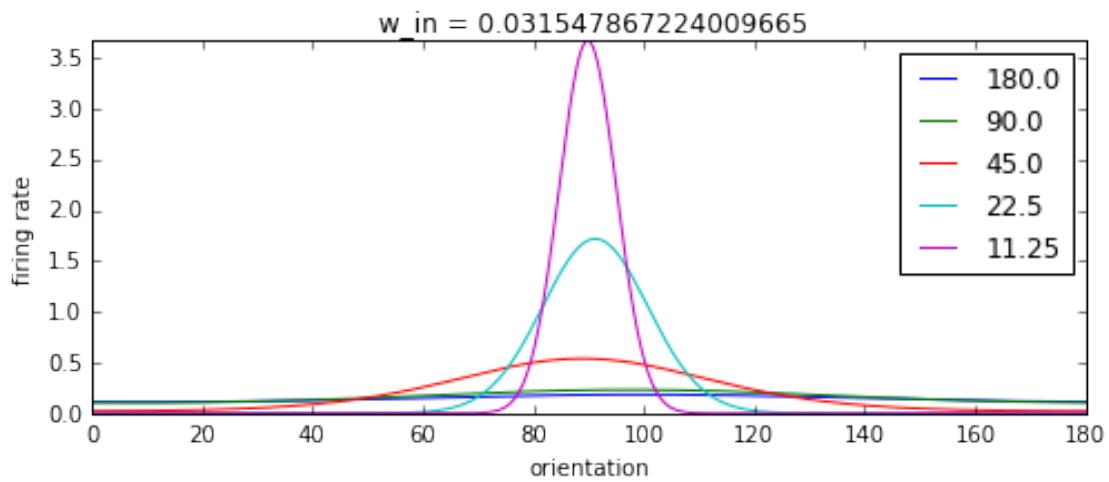
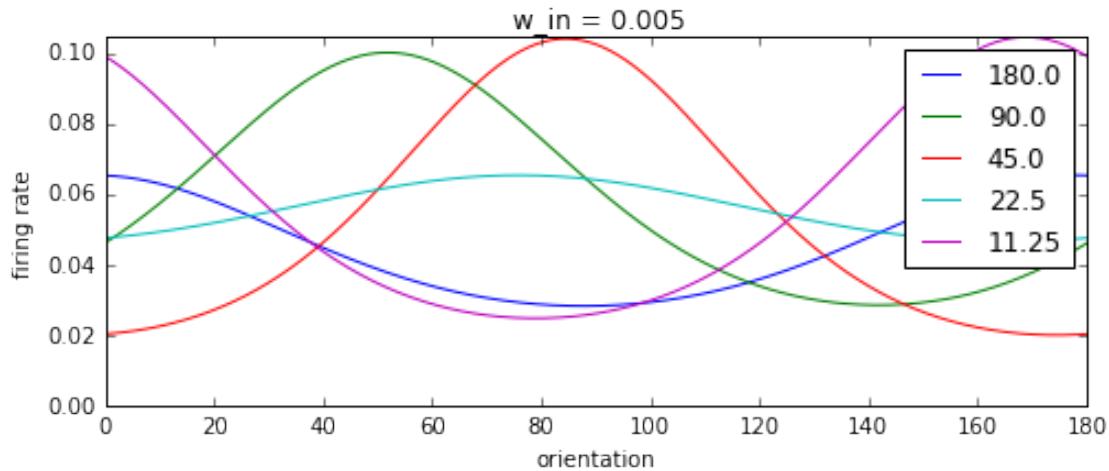
```

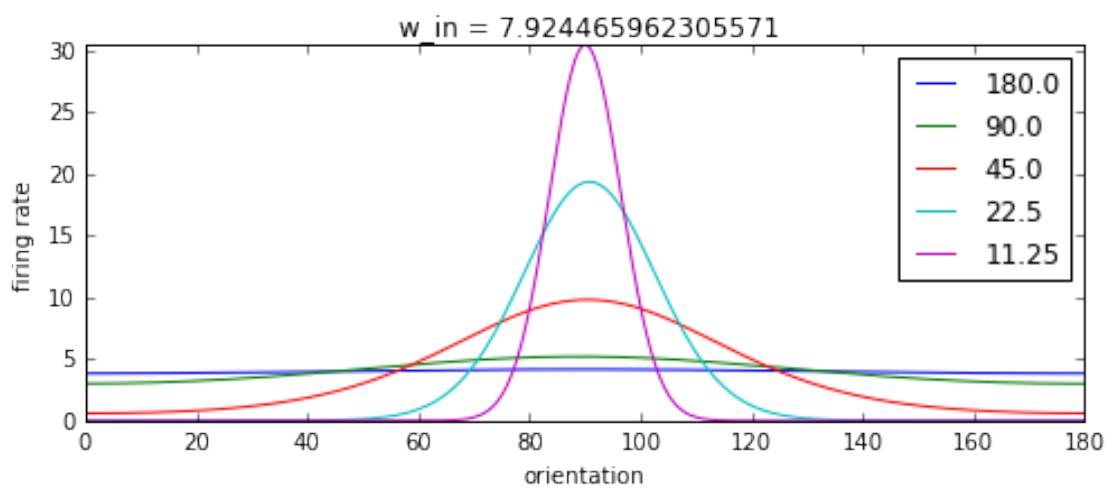
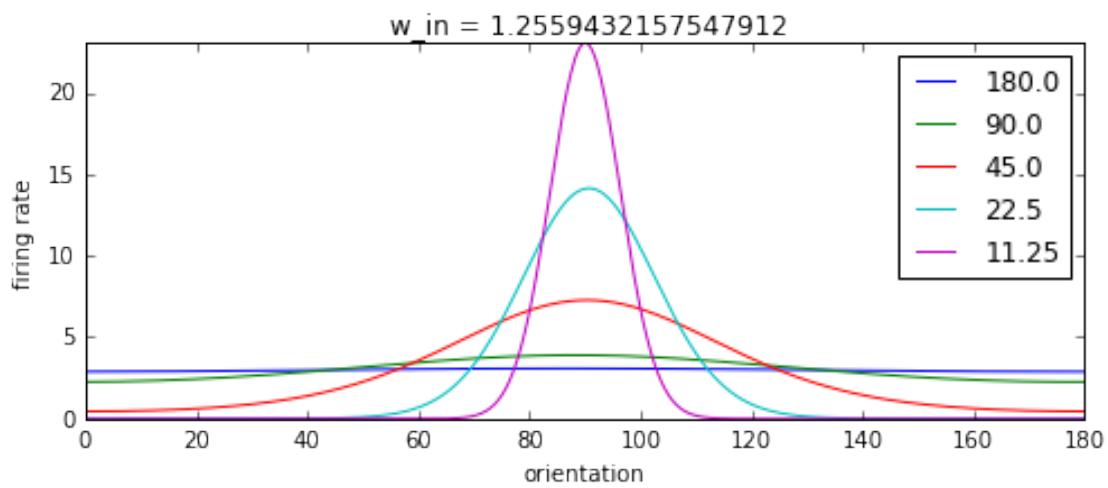


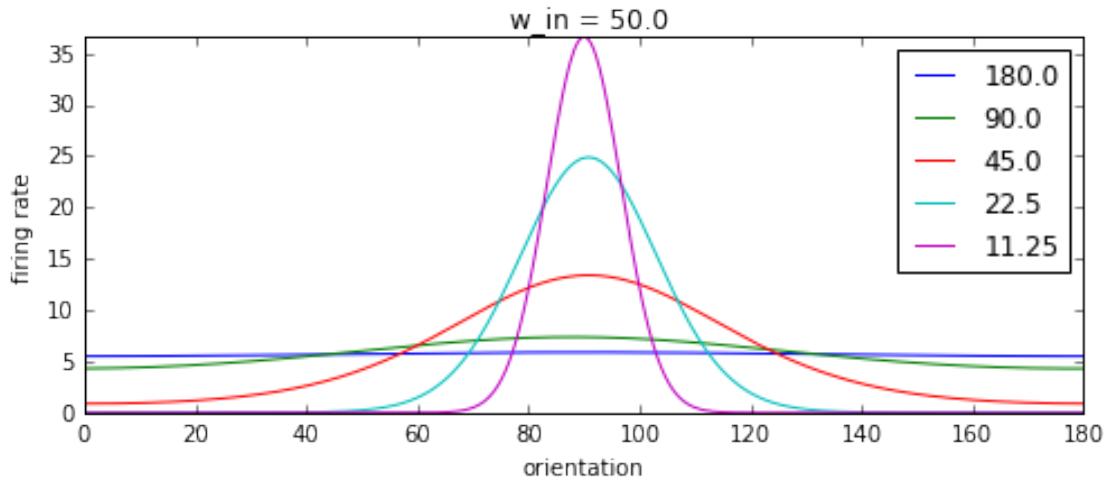
```

>>> net = RRNN()
... ws = net.w_in * np.logspace(-2, 2, 6)
... bw_values = 180*np.logspace(0, -4, 5, base=2)
... for w_in in ws:
...     fig, ax = plt.subplots(figsize=(8, 3))
...     for bw_value in bw_values:
...         net = RRNN(w_in=w_in, w=0)
...         net.sim_params['b_input'] = bw_value
...
...         df, spikesE, spikesI = net.model()
...         theta, fr, result = net.fit_vonMises(spikesE)
...         ax.plot(theta*180/np.pi, result.best_fit, label=str(bw_value))
...
...         ax.set_title(' w_in = {}'.format(w_in))
...         ax.set_xlabel('orientation')
...         ax.set_ylabel('firing rate')
...         ax.axis('tight')
...         ax.set_ylim(0)
...         plt.legend()
...         plt.show()

```



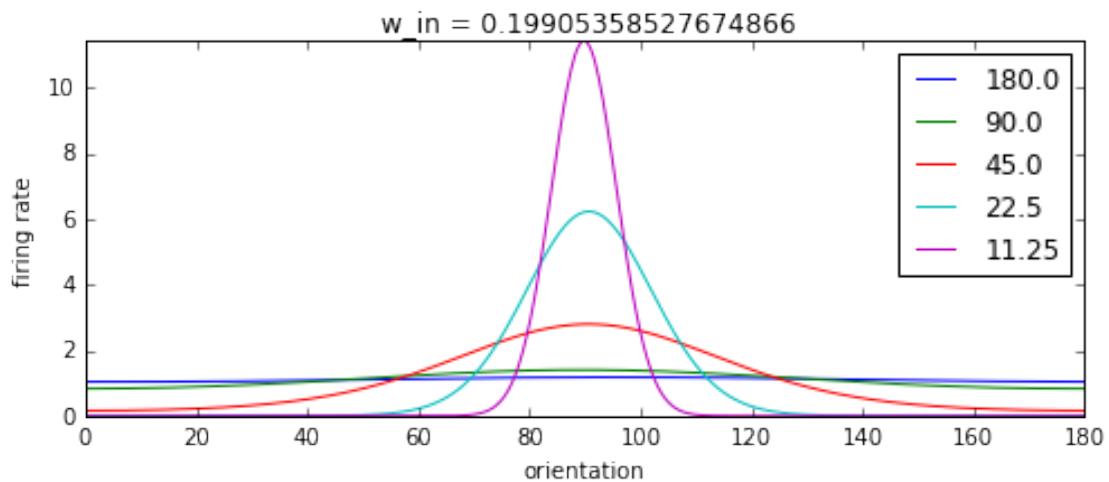
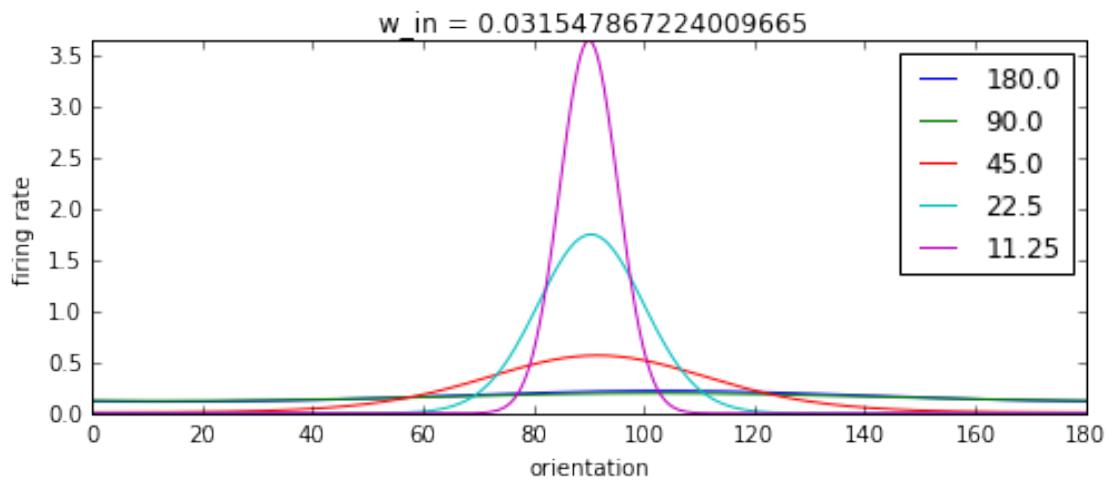
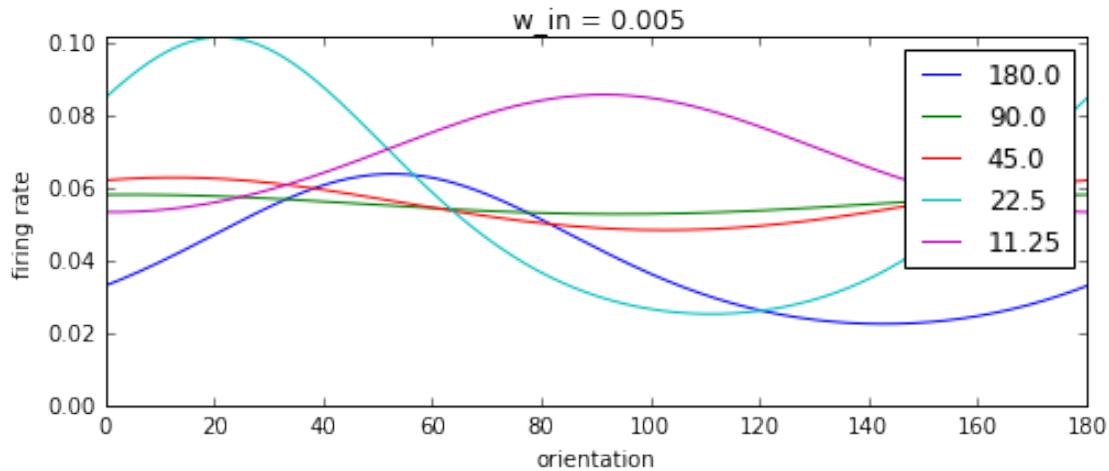


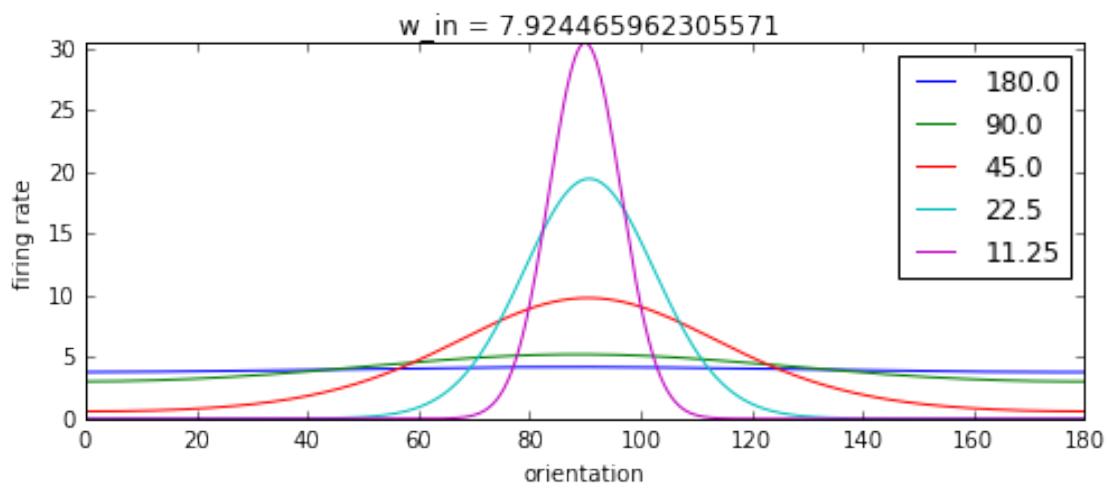
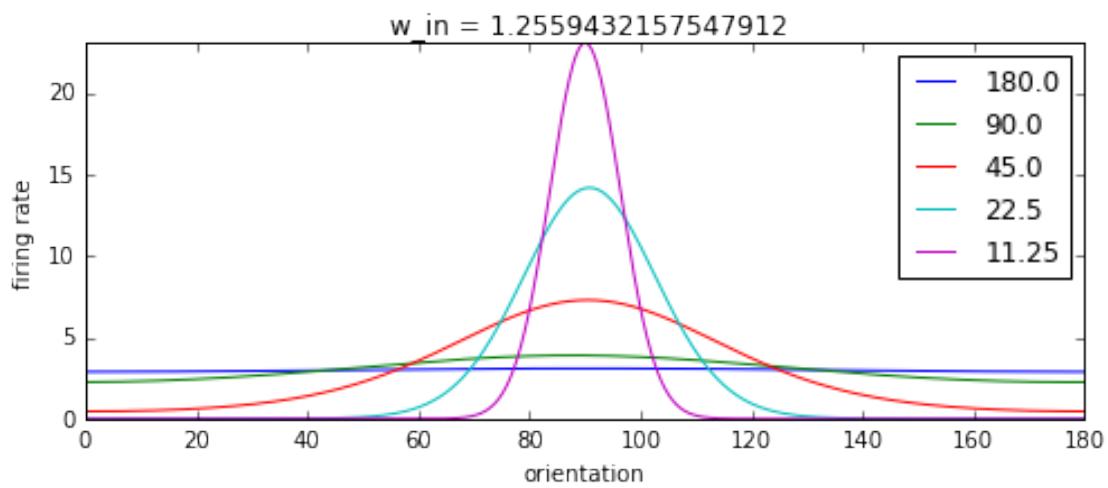


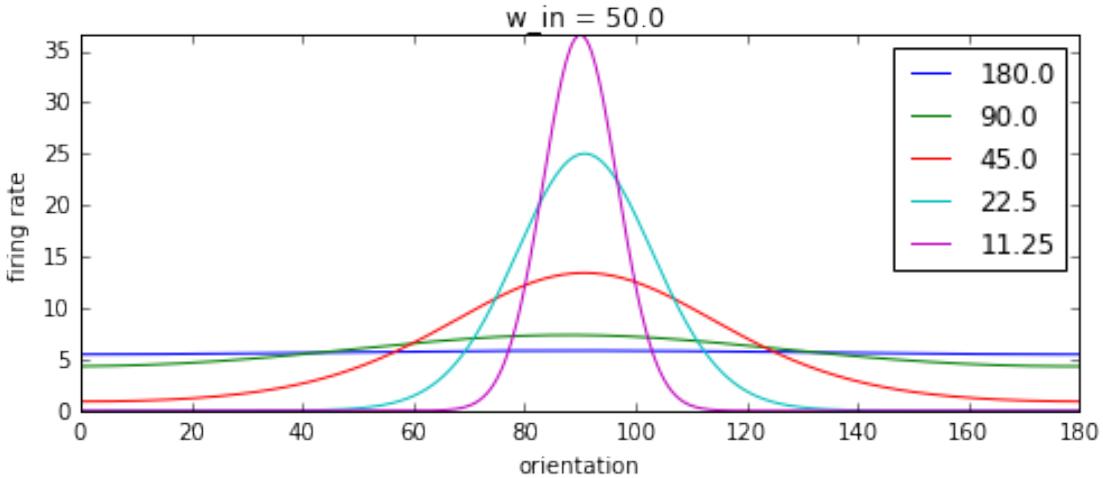
```

>>> for w_in in ws:
...     fig, ax = plt.subplots(figsize=(8, 3))
...     for bw_value in bw_values:
...         net = RRNN(w_in=w_in)
...         net.sim_params['b_input'] = bw_value
...
...         df, spikesE, spikesI = net.model()
...         theta, fr, result = net.fit_vonMises(spikesE)
...         ax.plot(theta*180/np.pi, result.best_fit, label=str(bw_value))
...
...         ax.set_title(' w_in = {}'.format(w_in))
...         ax.set_xlabel('orientation')
...         ax.set_ylabel('firing rate')
...         ax.axis('tight')
...         ax.set_ylim(0)
...         plt.legend()
...         plt.show()

```







3.2 Le ring accordé

3.2.1 connexion des couches suivant une topologit locale

Nous testons l'effet du changement de la bandwidth du signal d'entrée sur le comportement du réseau en désactivant toutes les projections sauf la projection d'entrée.

Pour cela, nous paramétrons l'activité des neurones de la population source de telle sorte que celle-ci représente une orientation de contraste de 90° . Une simulation du modèle est exécutée pour chaque valeur de bandwidth. Seule la projection de la source à la population E est active, W est nul. Des rasterplots des trois populations sont alors affichés.

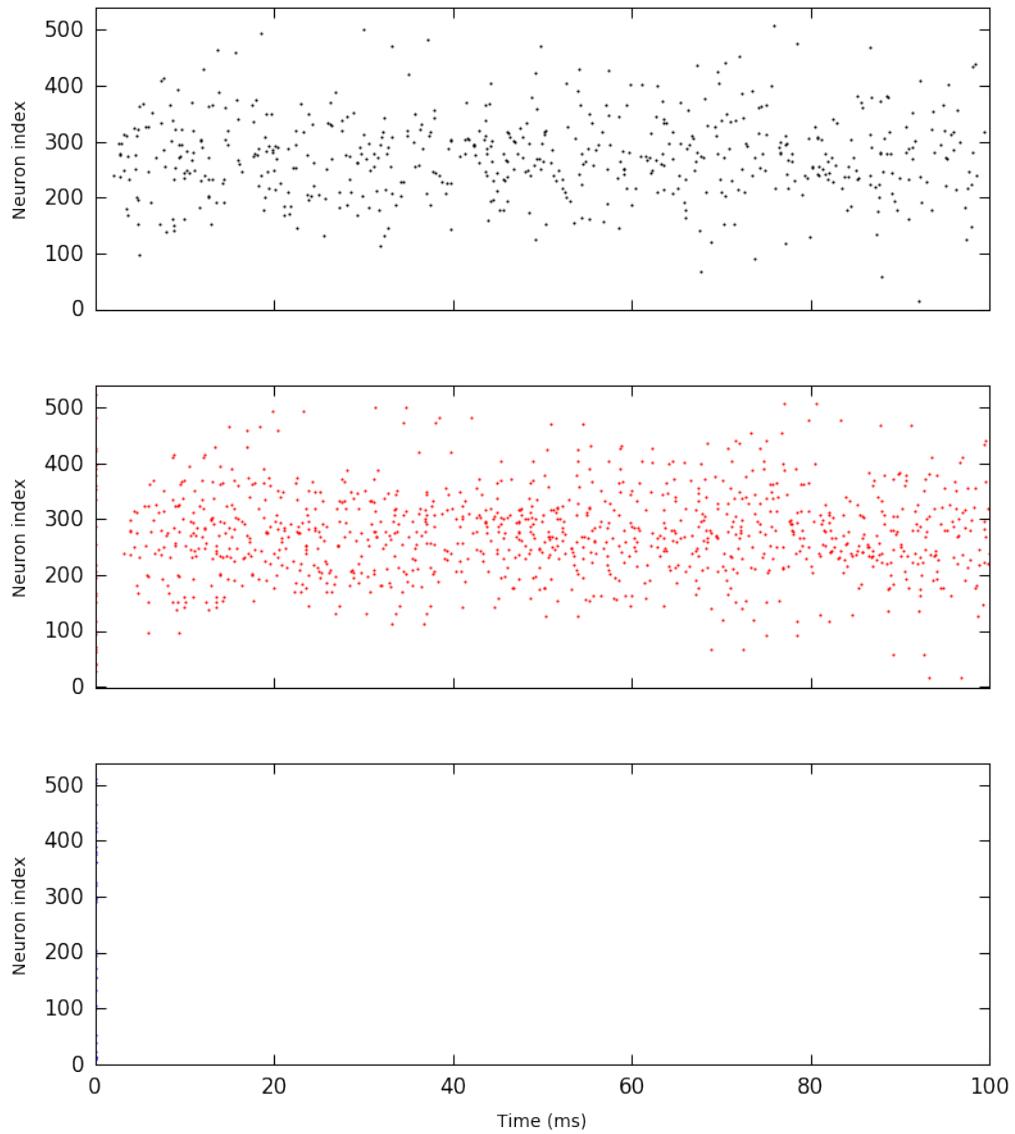
Nous observons bien que certains neurones de la population E, ceux qui ont une réponse sélective à une orientation de ou proche de 90° sont actifs. Aussi, les neurones de la population I sont inactifs.

```
>>> from RRNN import RRNN
... import numpy as np
... import matplotlib.pyplot as plt
... from pyNN.utility.plotting import Figure, Panel
...
... markersize = 1.
... time=100
... net = RRNN(time=time, w=0.)
... net.sim_params['b_input'] = 40
... net.model()
... title = 'Feed-Forward'
...
... fig = Figure(Panel(net.spikesP.spiketrains, xticks=False, yticks=True,
```

```
...     Panel(net.spikesE.spiketrains, xticks=False, yticks=True)
...
...     Panel(net.spikesI.spiketrains, xlabel="Time (ms)", xticks=True,
...           title='----- {} -----'.format(title))
...
... fig.fig.savefig("../figs/ringFF.png", dpi = 600)

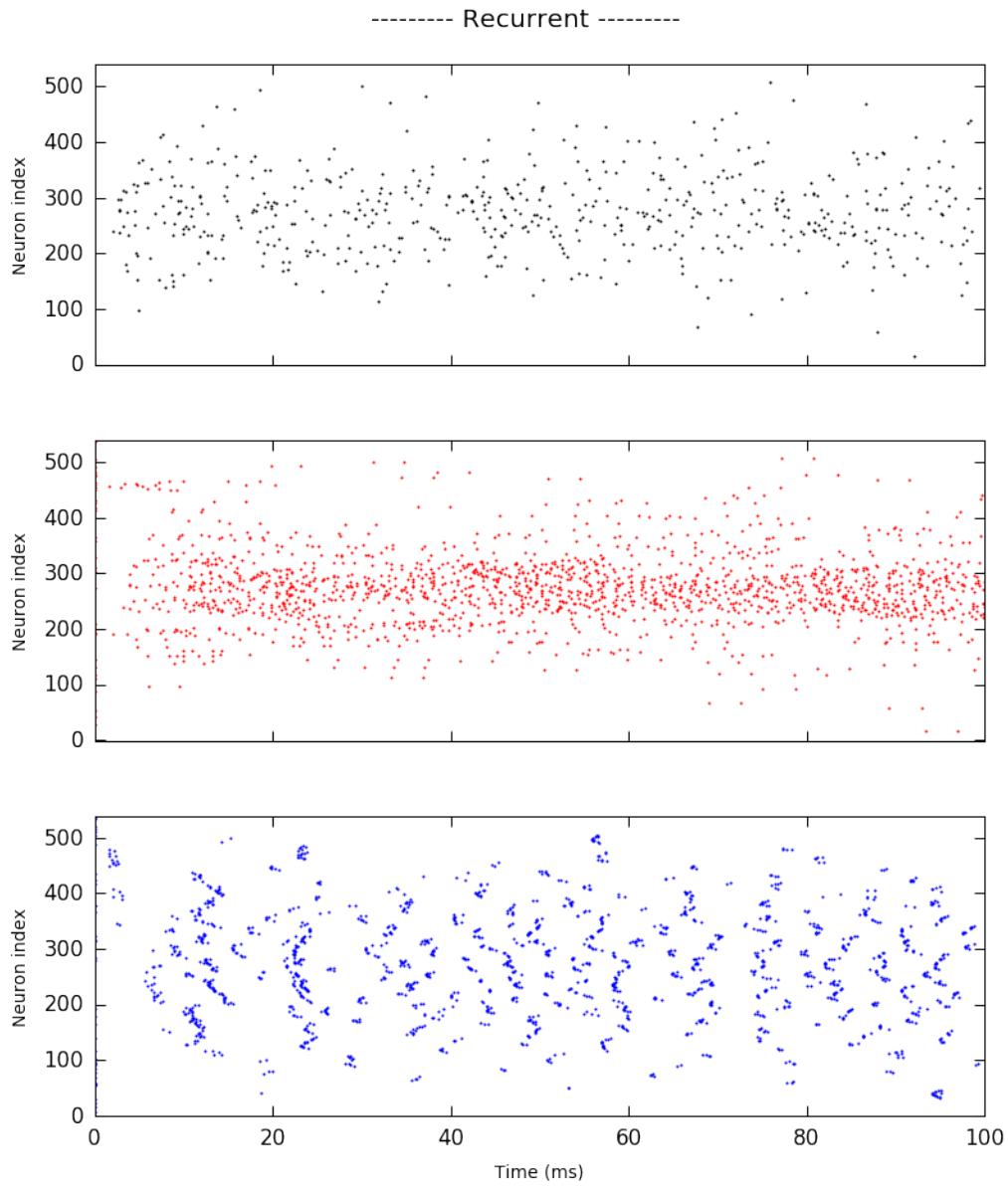
CSAConnector: libneurosim support not available in NEST.
Falling back on PyNN's default CSAConnector.
Please re-compile NEST using --with-libneurosim=PATH
/usr/local/lib/python3.5/site-packages/matplotlib/__init__.py:1350: UserWarning:
because the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.
```

----- Feed-Forward -----



```
>>> markersize = 1.  
... time = 100  
...  
... net = RRNN(time=time)  
... w, g, w_in =net.w, net.g, net.w_in  
... def rec_net(time=time, w=w, g=g,  
...             b_input=40, b_exc_inh=50, b_exc_exc=4, b_inh_exc=50, b_inh_<br/>...             c=1)  
...     net = RRNN(time=time, w=w, g=2.5, c=1)  
...     net.sim_params['b_input'] = b_input  
...     net.sim_params['b_exc_inh'] = b_exc_inh
```

```
...     net.sim_params['b_exc_exc'] = b_exc_exc
...     net.sim_params['b_inh_exc'] = b_inh_exc
...     net.sim_params['b_inh_inh'] = b_inh_inh
...     return net
...
...
... net = rec_net(time=time)
... #df, spikesE, spikesI = net.model()
... #net = RRNN(time=1000, c=1, g=2)
... #net.sim_params['b_input'] = 45
... #net.sim_params['b_exc_inh'] = 15
... #net.sim_params['b_exc_exc'] = 15
... #net.sim_params['b_inh_exc'] = 15
... #net.sim_params['b_inh_inh'] = 15
... net.model()
... title = 'Recurrent'
...
...
... fig = Figure(Panel(net.spikesP.spike trains, xticks=False, yticks=True,
...                     Panel(net.spikesE.spike trains, xticks=False, yticks=True, y),
...                     Panel(net.spikesI.spike trains, xlabel="Time (ms)", xticks=True, y),
...                     title='----- {} -----'.format(title)))
...
...
... fig.fig.savefig("../figs/ringRecurrent.png", dpi = 600)
```



3.2.2 Effet de la bandwidth d'entrée

Ici, nous étudions l'effet d'une variation de la bandwidth d'entrée sur l'activité neuronale au sein d'un ring, que nous voulons accordé, afin de vérifier qu'il l'est bien.

Nous paramétrons le ring de telle sorte qu'il soit accordé. Puis, nous exécutons une simulation pour chaque valeur de bandwidth d'entrée désirée. Nous générerons alors, pour chaque simulation, les rasterplots des populations source, excitatrice et inhibitrice.

Nous constatons alors que l'activité des neurones des populations excitatrices et inhibitrices reproduisent bien l'activité de la population source. En effet, seuls les neurones

que nous voulions sélectifs aux orientations contenues dans l'entrée sont actifs.

```
>>> from RRNN import RRNN
... from pyNN.utility.plotting import Figure, Panel
... markersize = .5
...
... time = 100
... net = RRNN(time=time)
... net.sim_params['b_input'] = 15
... df_sim, spikesE, spikesI = net.model()
... f = net.Raster(df_sim, spikesE, spikesI)
... #f.fig.savefig('../figs/ring_untuned_rasterplot.png', dpi = 600)
... #f

...
... def rec_net(time=time, w=net.w, g=net.g, b_input=15, b_exc_inh=50, b_ex
...     net = RRNN(time=time, w=w, g=g, c=1)
...     net.sim_params['b_input'] = b_input
...     net.sim_params['b_exc_inh'] = b_exc_inh
...     net.sim_params['b_exc_exc'] = b_exc_exc
...     net.sim_params['b_inh_exc'] = b_inh_exc
...     net.sim_params['b_inh_inh'] = b_inh_inh
...     return net
...
... net = rec_net(time=time)
... df_sim, spikesE, spikesI = net.model()
... f = net.Raster(df_sim, spikesE, spikesI)
... #f.fig.savefig('../figs/ring_recurrent_rasterplot.png', dpi = 600)
... #f.fig.show()
```

CSAConnector: libneurosim support not available in NEST.

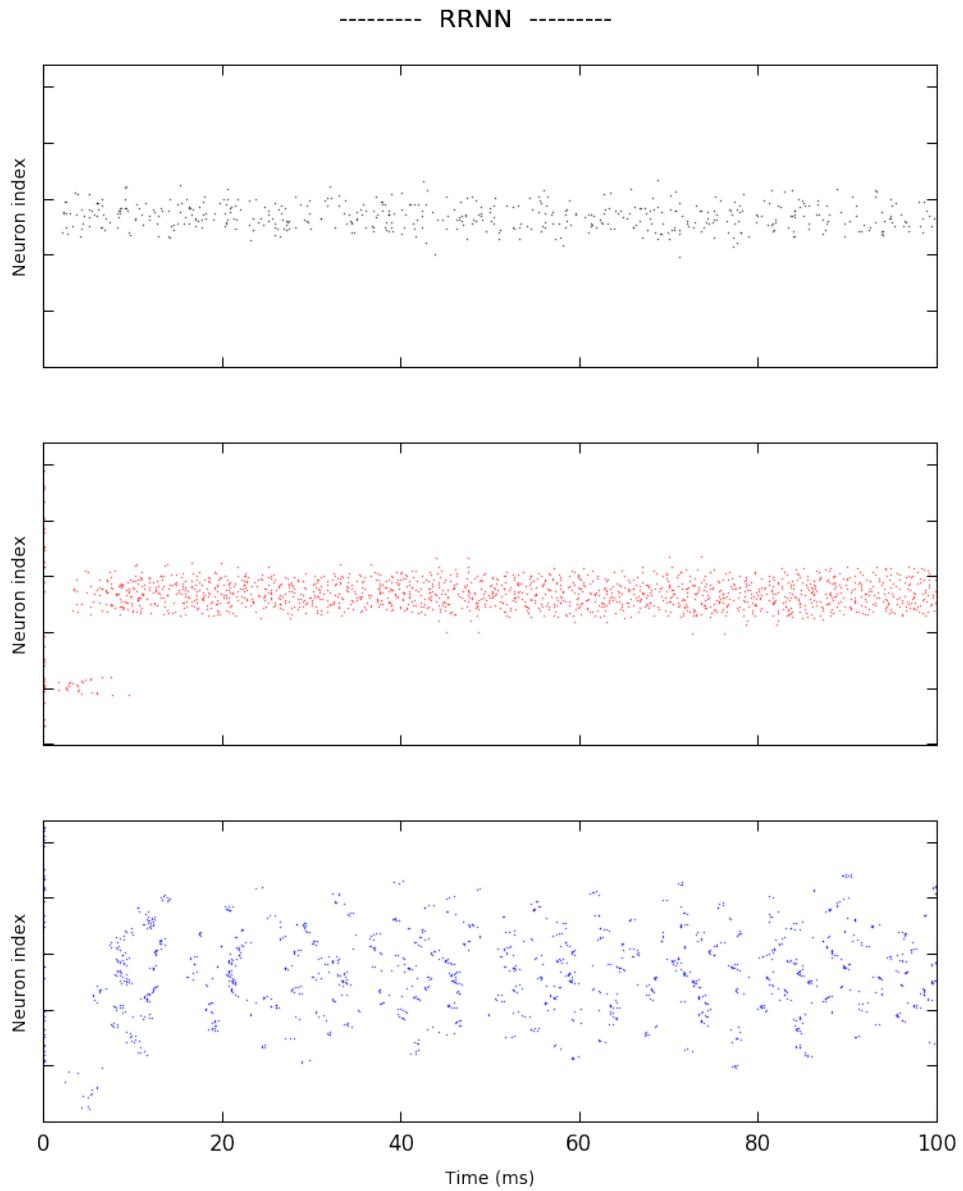
Falling back on PyNN's default CSAConnector.

Please re-compile NEST using --with-libneurosim=PATH

/usr/local/lib/python3.5/site-packages/matplotlib/__init__.py:1350: UserWarning:

because the backend has already been chosen;

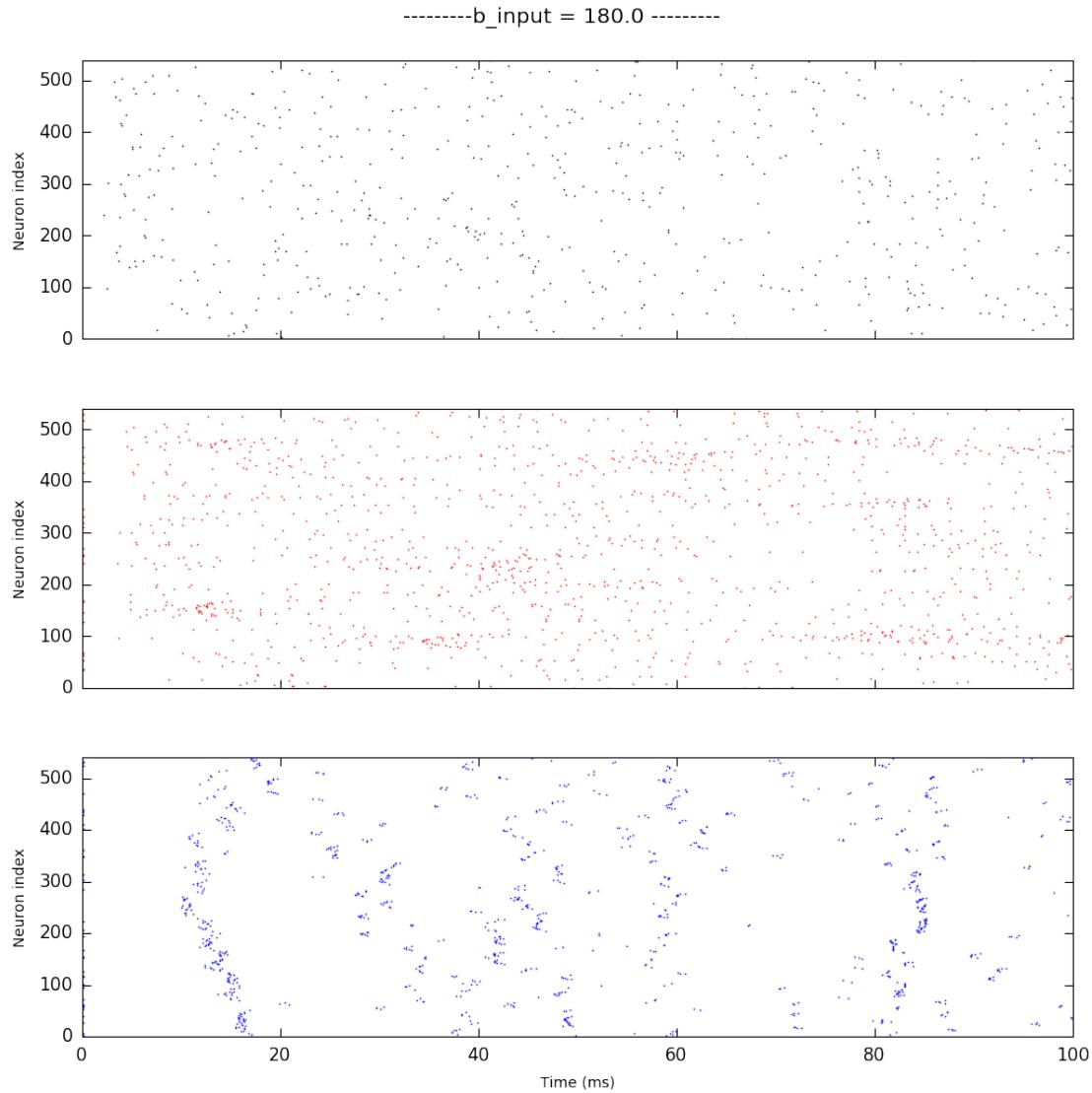
matplotlib.use() must be called ***before*** pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.



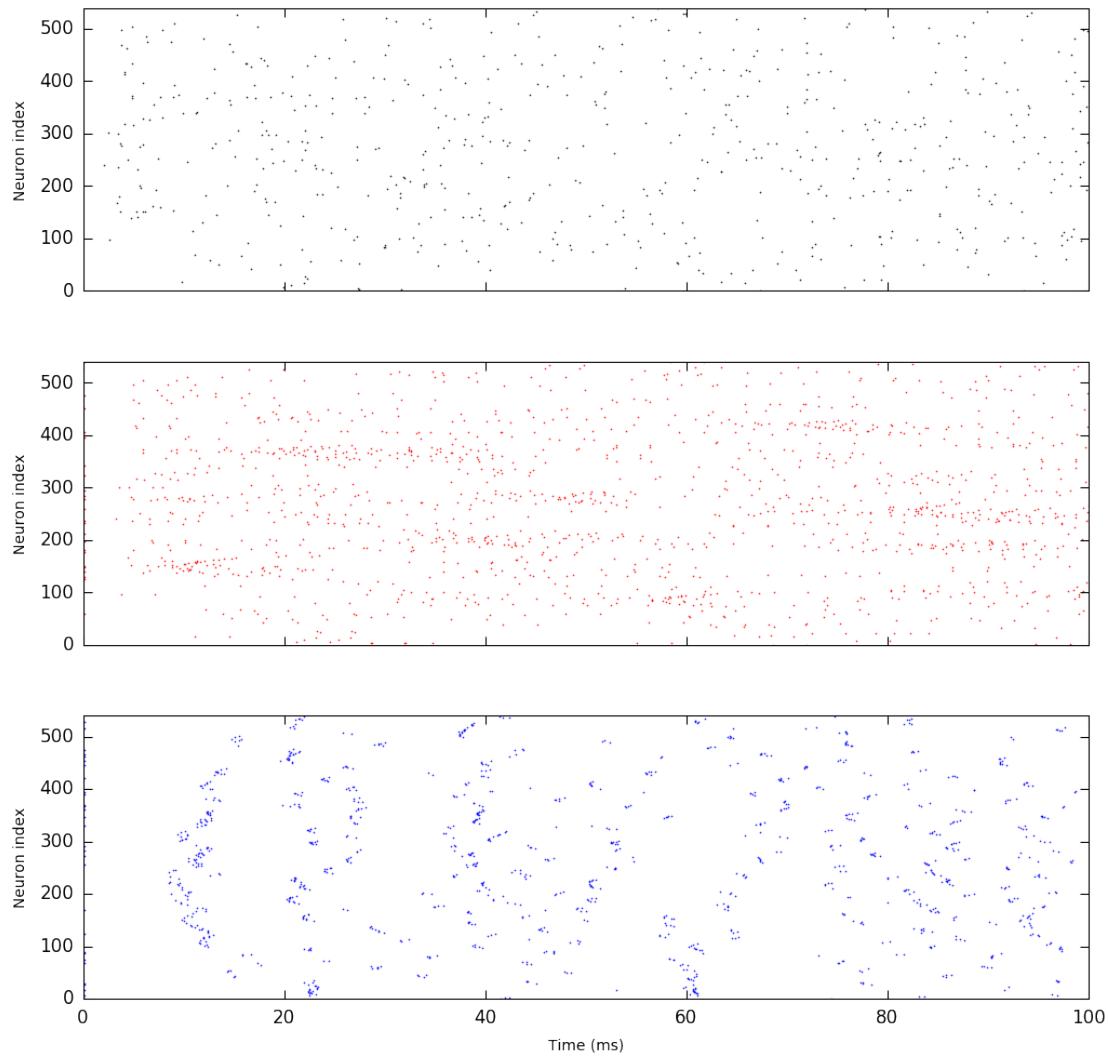
On fait maintenant une exploration des paramètres

```
>>> net = RRNN()
... bw_values = 180*np.logspace(0, -4, 5, base=2)
... for bw in bw_values:
...     fig, ax = plt.subplots(figsize=(8,8))
...     net = rec_net(time=time, b_input=bw)
...
...     df, spikesE, spikesI = net.model()
...     fig = Figure(Panels(net.spikesP.spike_trains, xticks=False, yticks=True),
...                  Panels(net.spikesE.spike_trains, xticks=False, yticks=True,
...                         title='RRNN'))
```

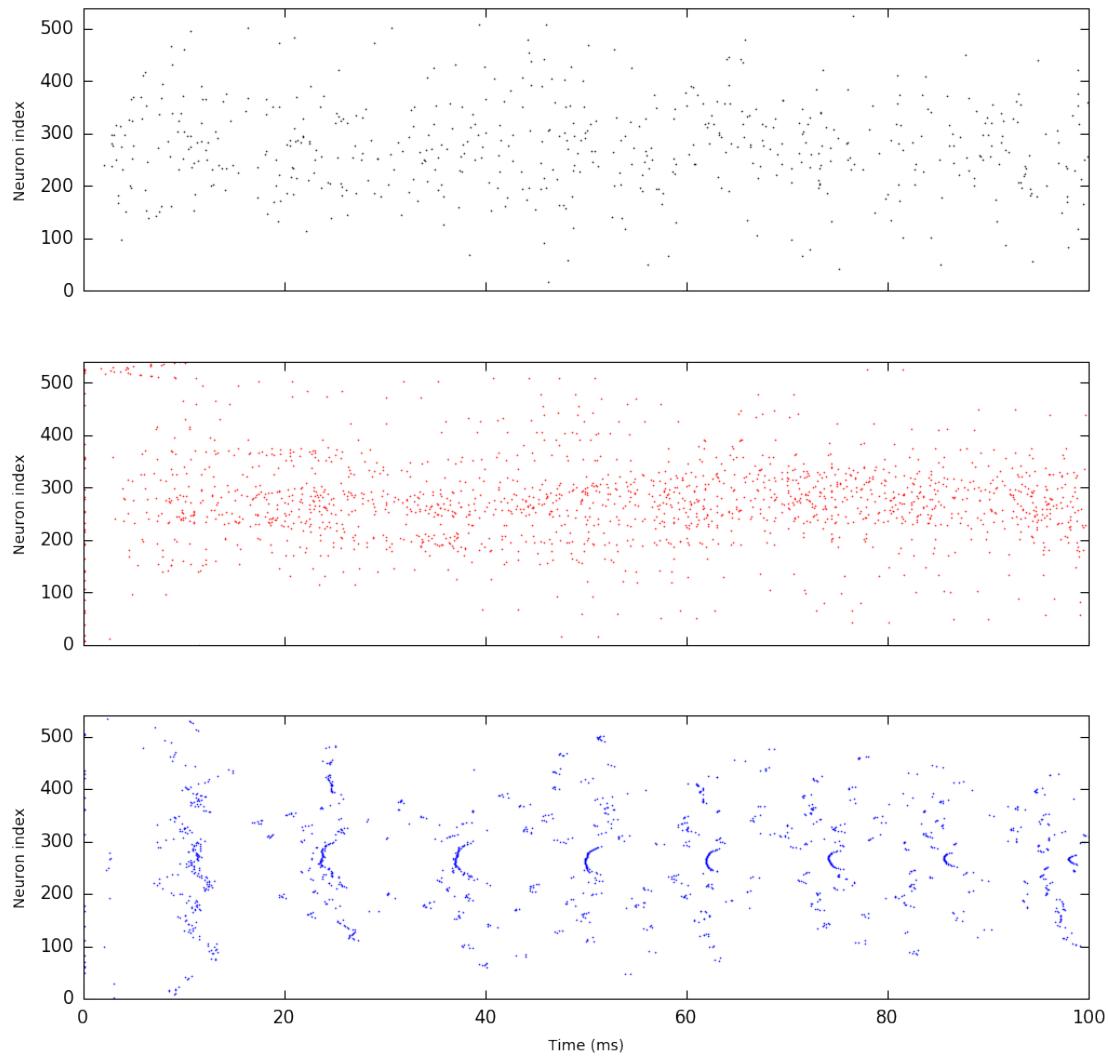
```
...     Panel(net.spikesI.spiketrains, xlabel="Time (ms)", xticks=T,
... title='-----b_input = {} -----'.format(str(bw)))
...
... plt.show()
```



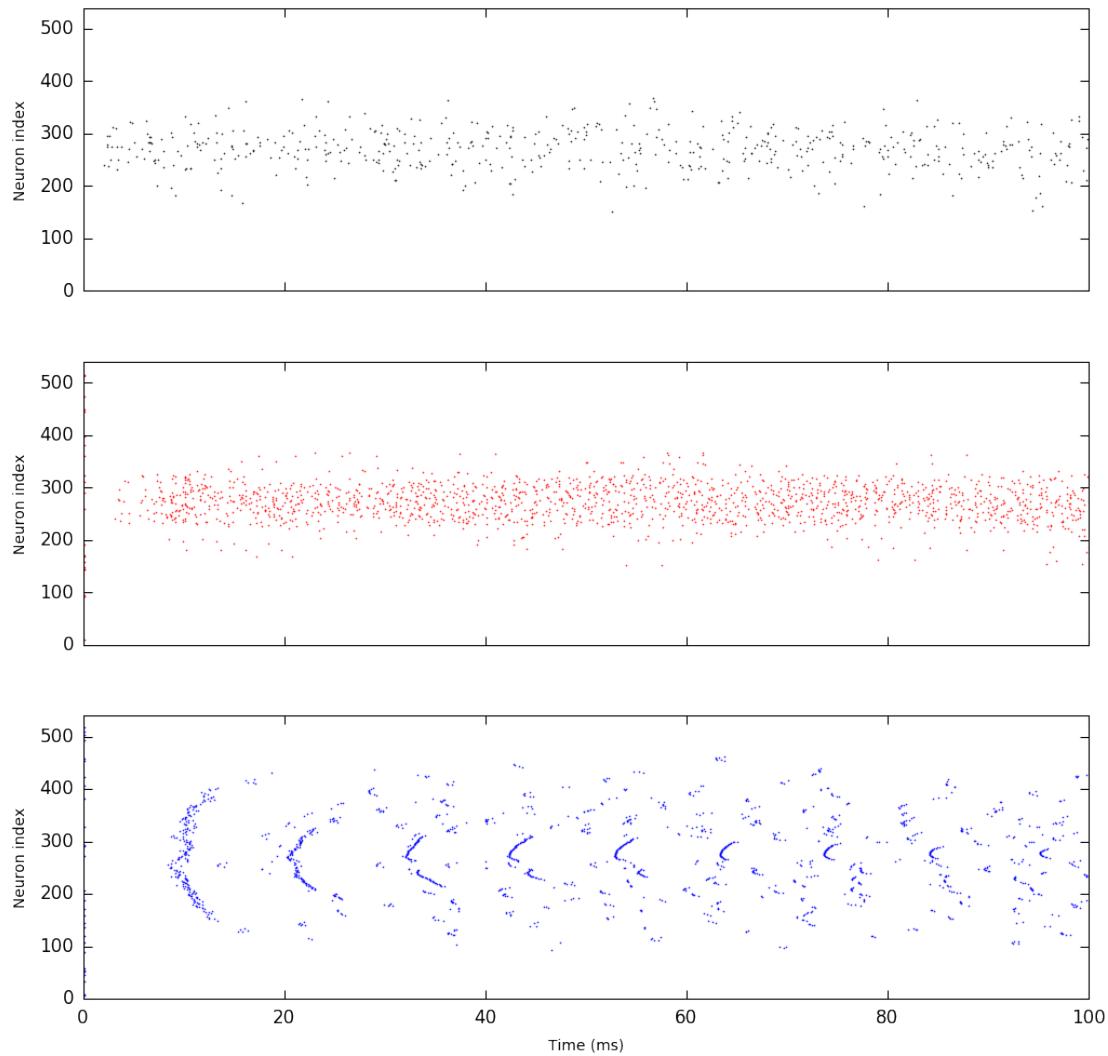
-----b_input = 90.0 -----

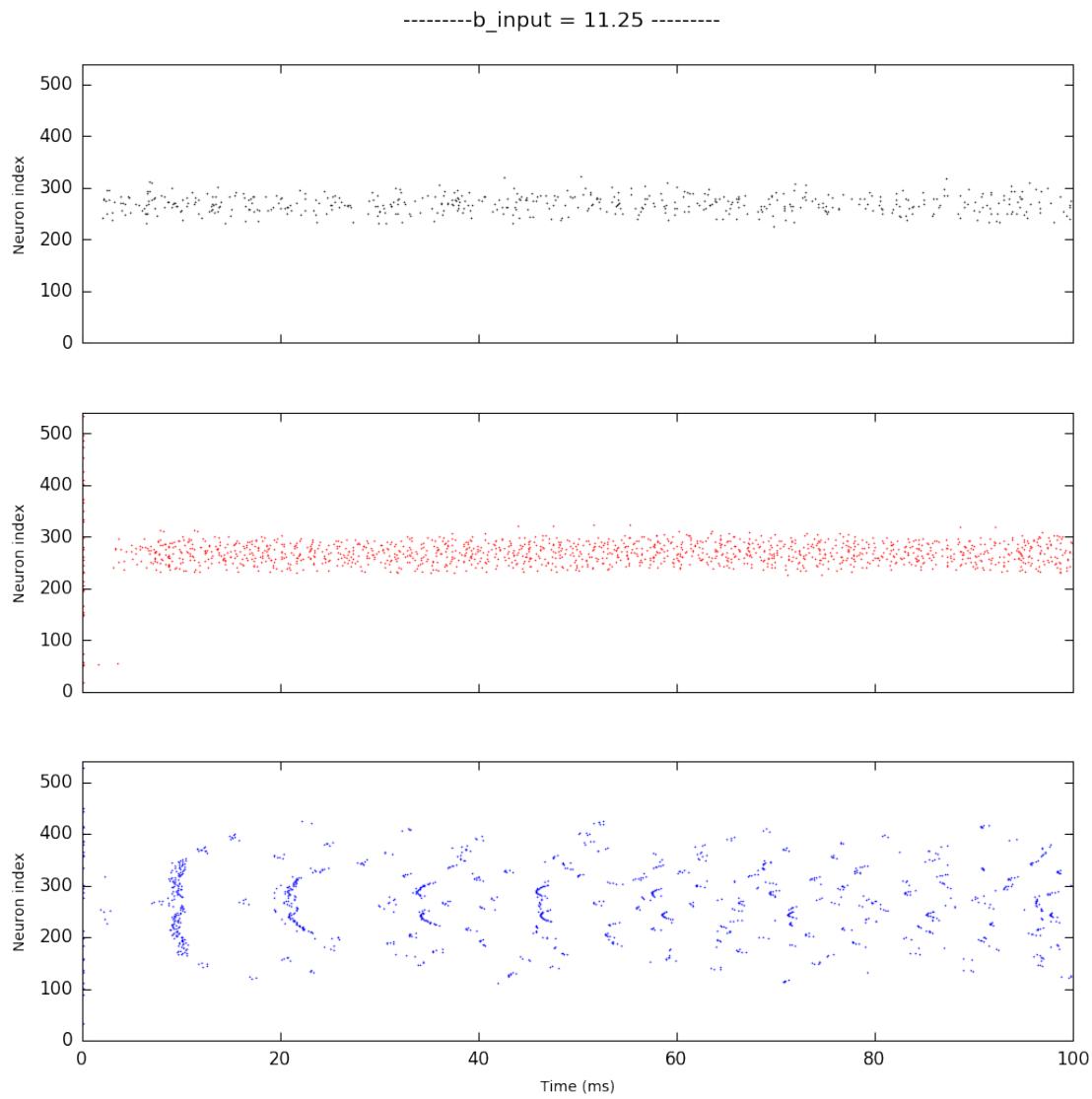


-----b_input = 45.0 -----



-----b_input = 22.5 -----



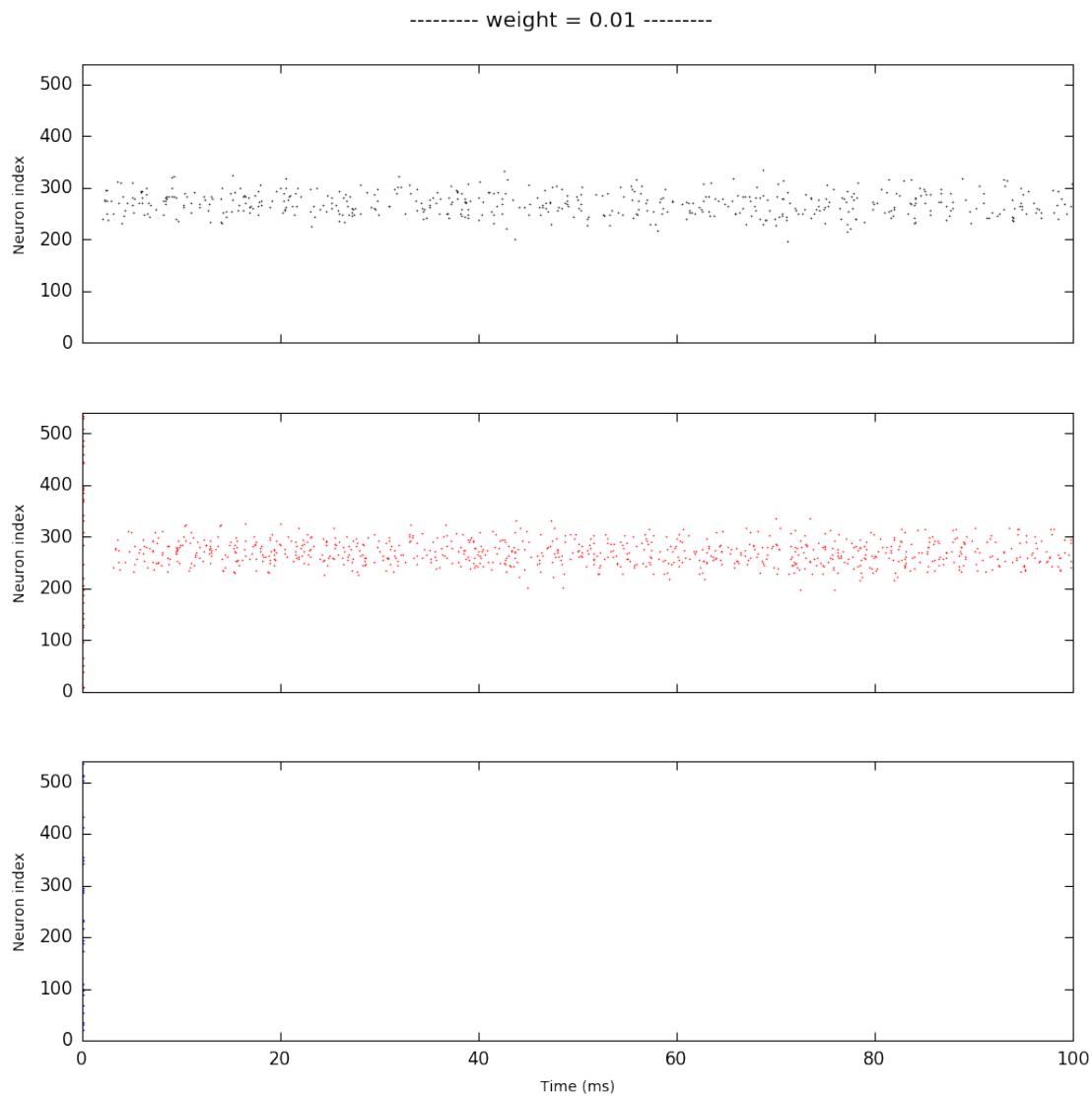


```

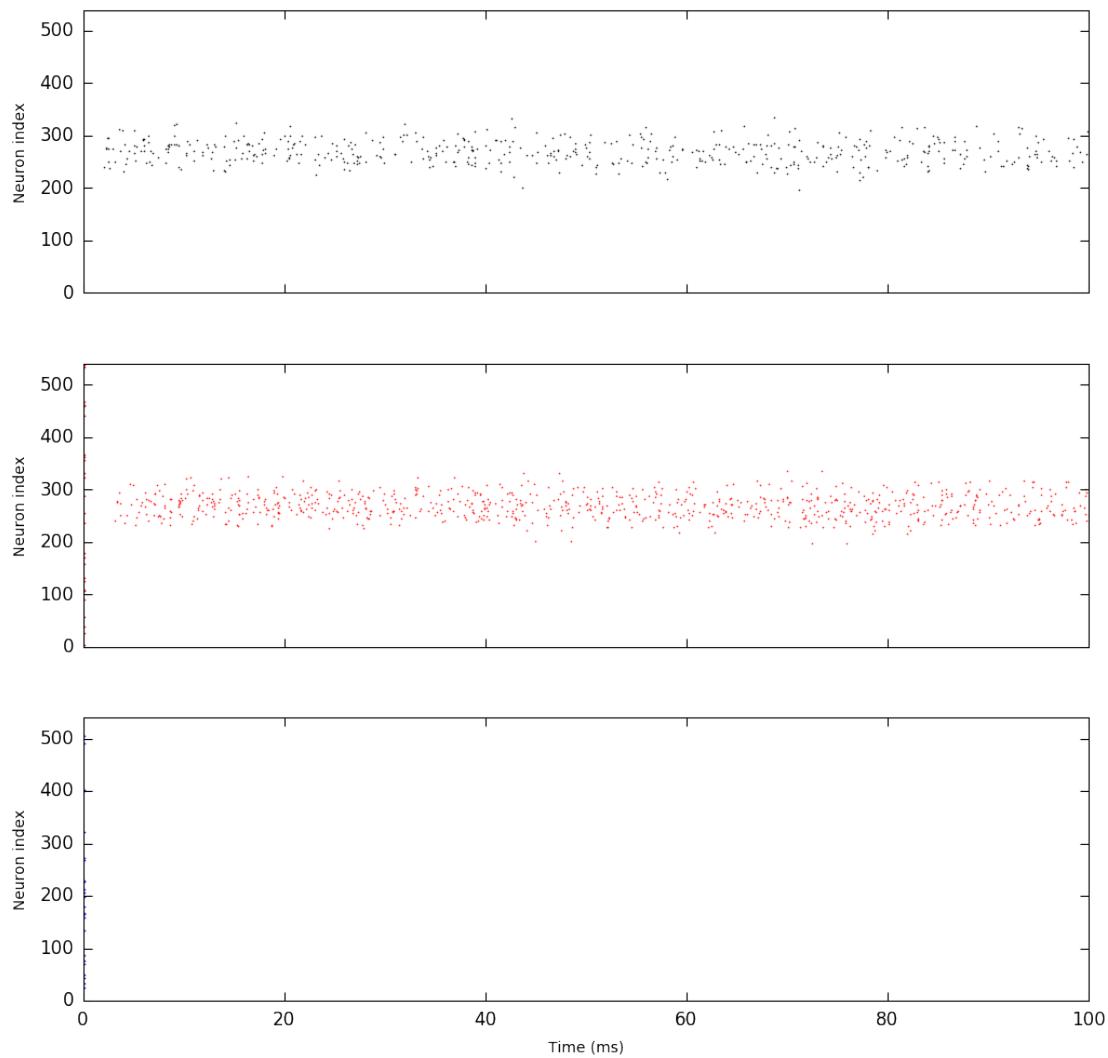
>>> net = RRNN()
... ws = net.w * np.logspace(-1, 1, 9)
... for w in ws:
...     fig, ax = plt.subplots(figsize=(8, 8))
...     net = rec_net(time=time, w=w)
...
...     df, spikesE, spikesI = net.model()
...     fig = Figure(Panel(net.spikesP.spiketrains, xticks=False, yticks=True,
...                         Panel(net.spikesE.spiketrains, xticks=False, yticks=True,
...                         Panel(net.spikesI.spiketrains, xlabel="Time (ms)", xticks=True,
...                         title='----- weight = {} -----'.format(str(w)))
...

```

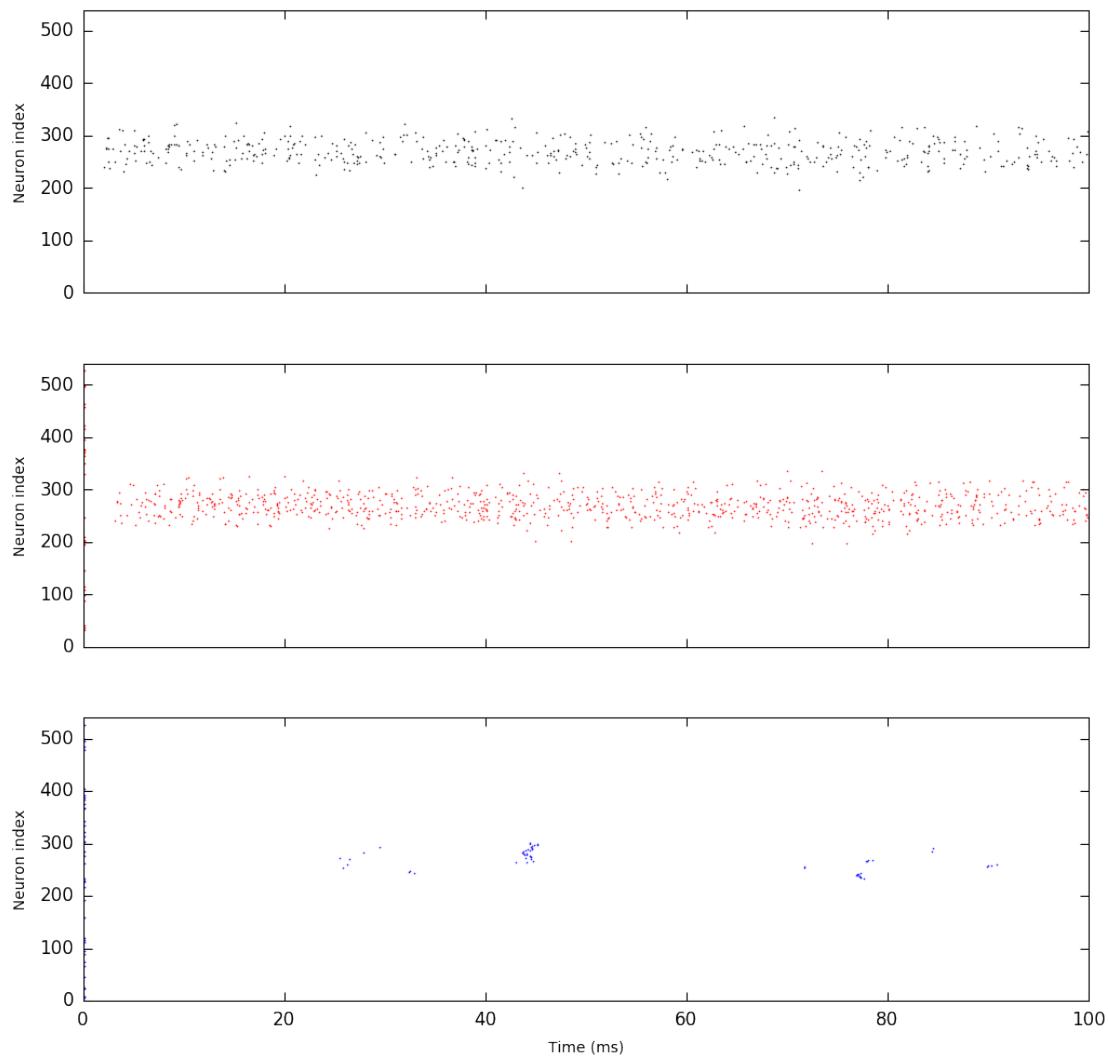
```
...     plt.show()
```



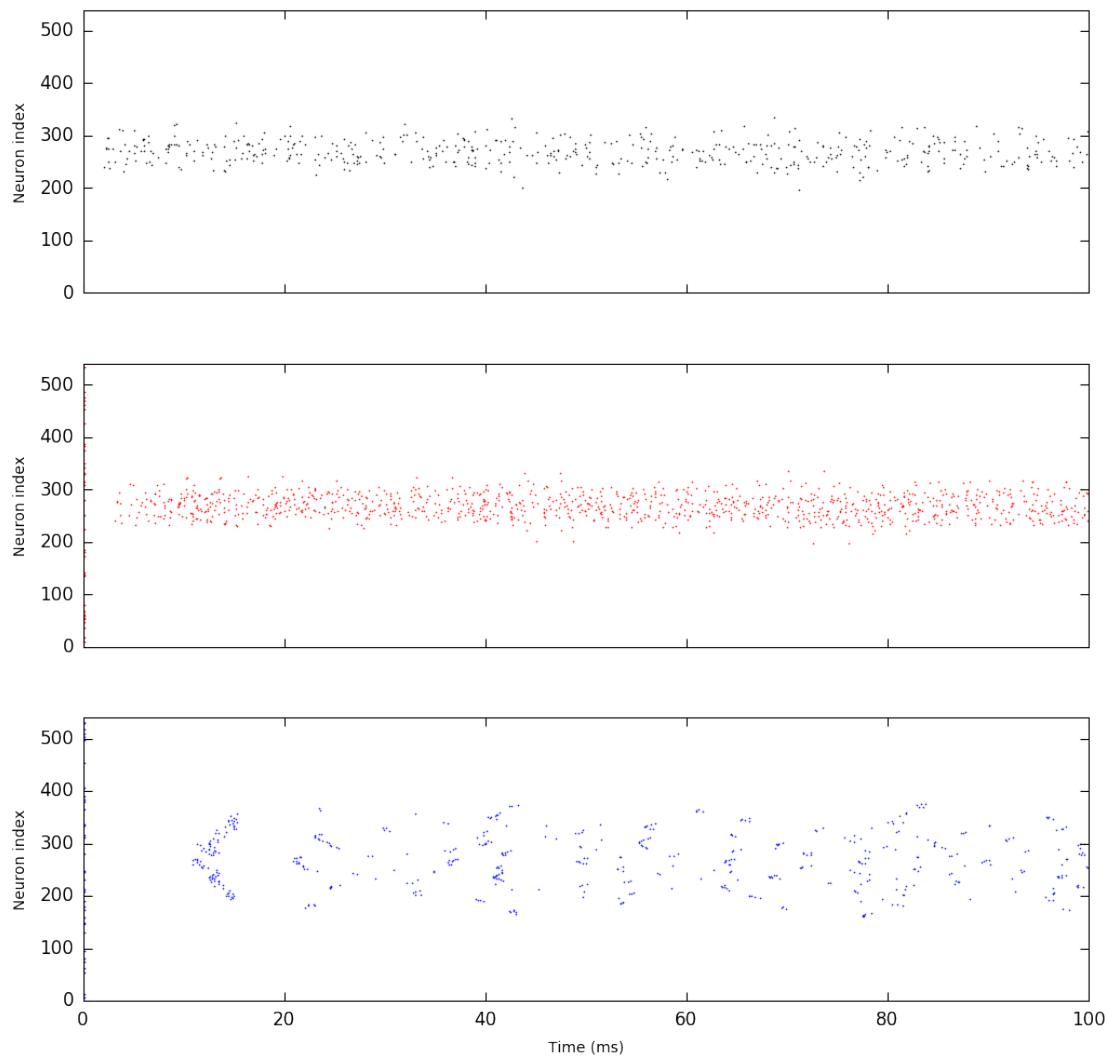
----- weight = 0.0177827941004 -----

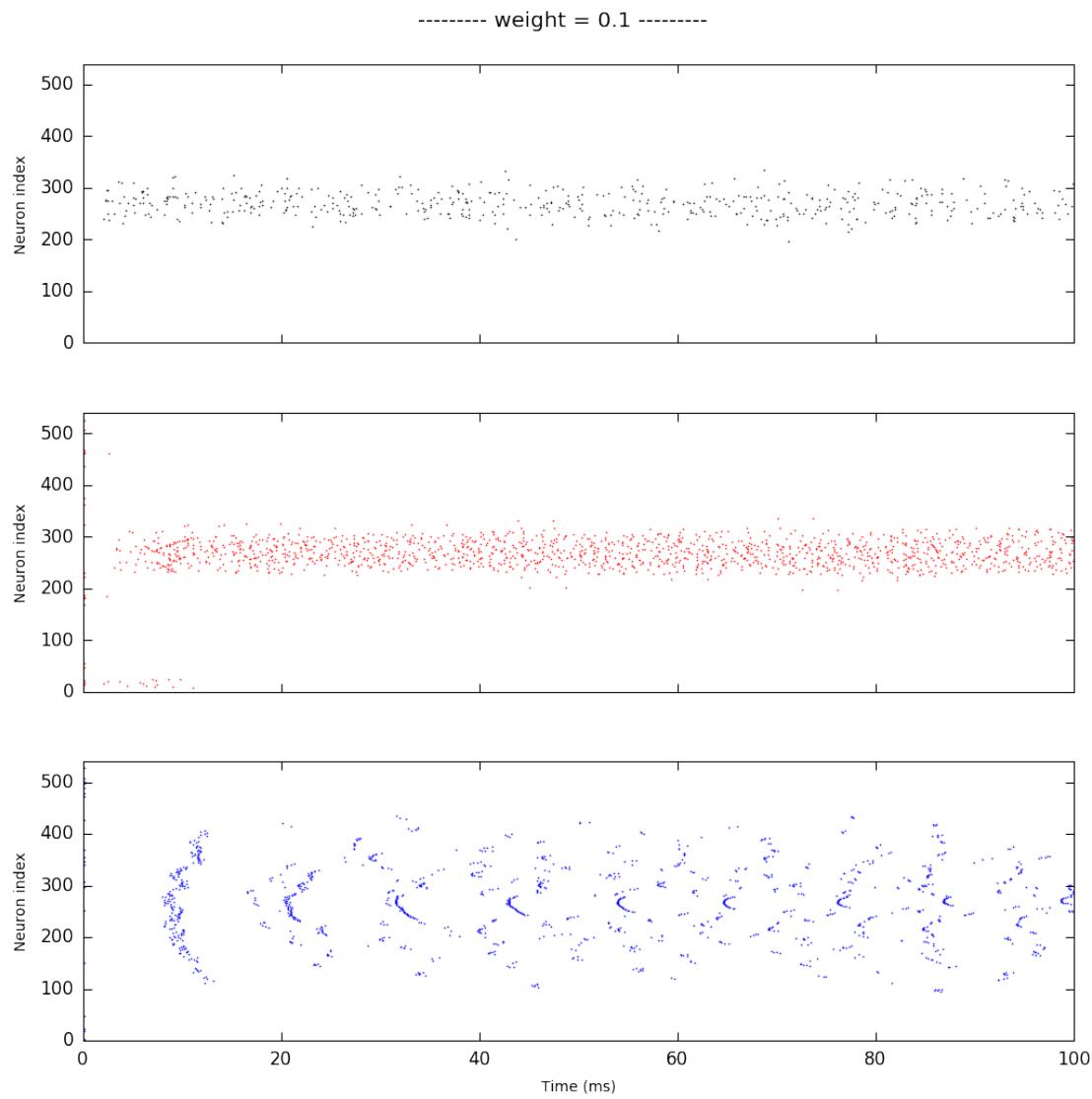


----- weight = 0.0316227766017 -----

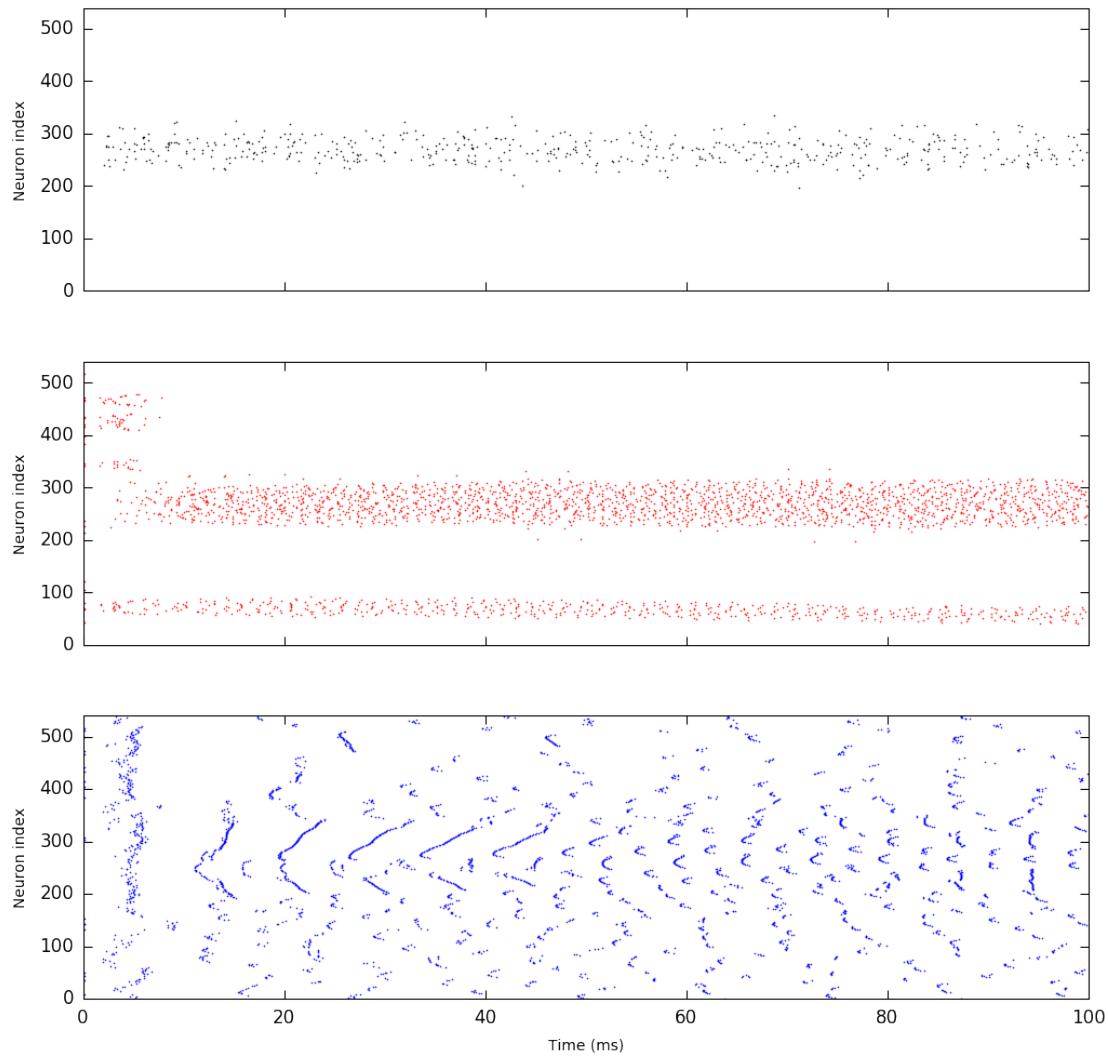


----- weight = 0.056234132519 -----

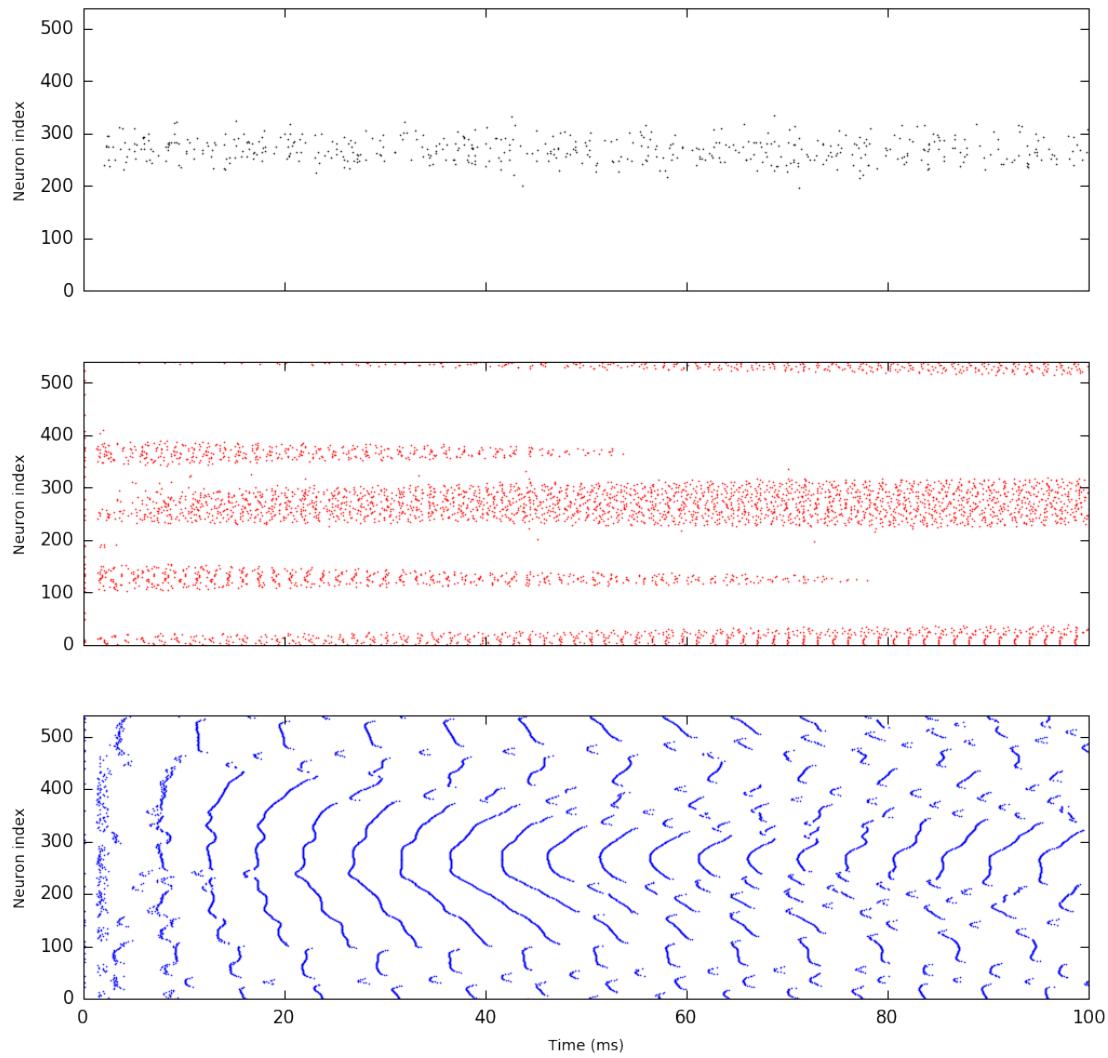




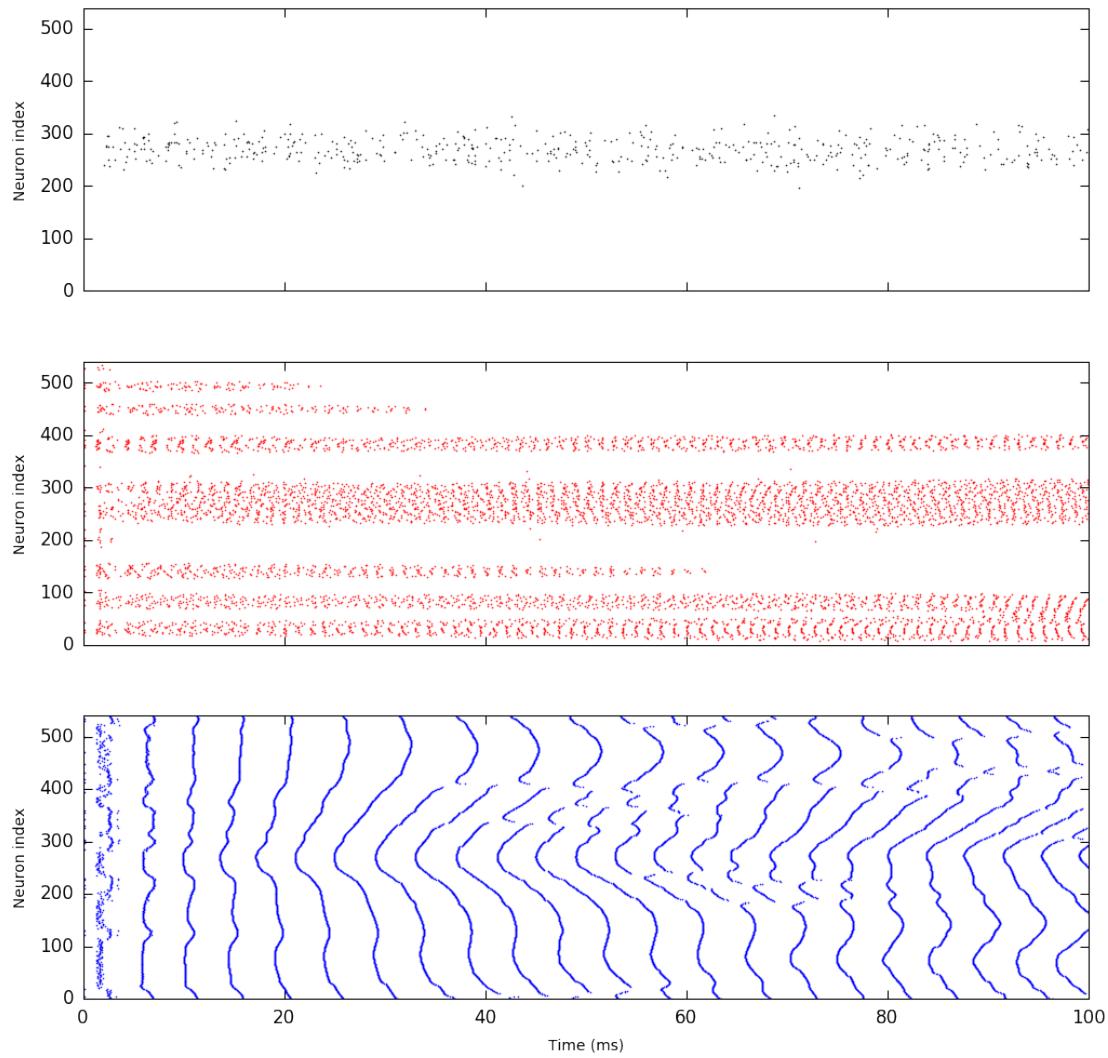
----- weight = 0.177827941004 -----

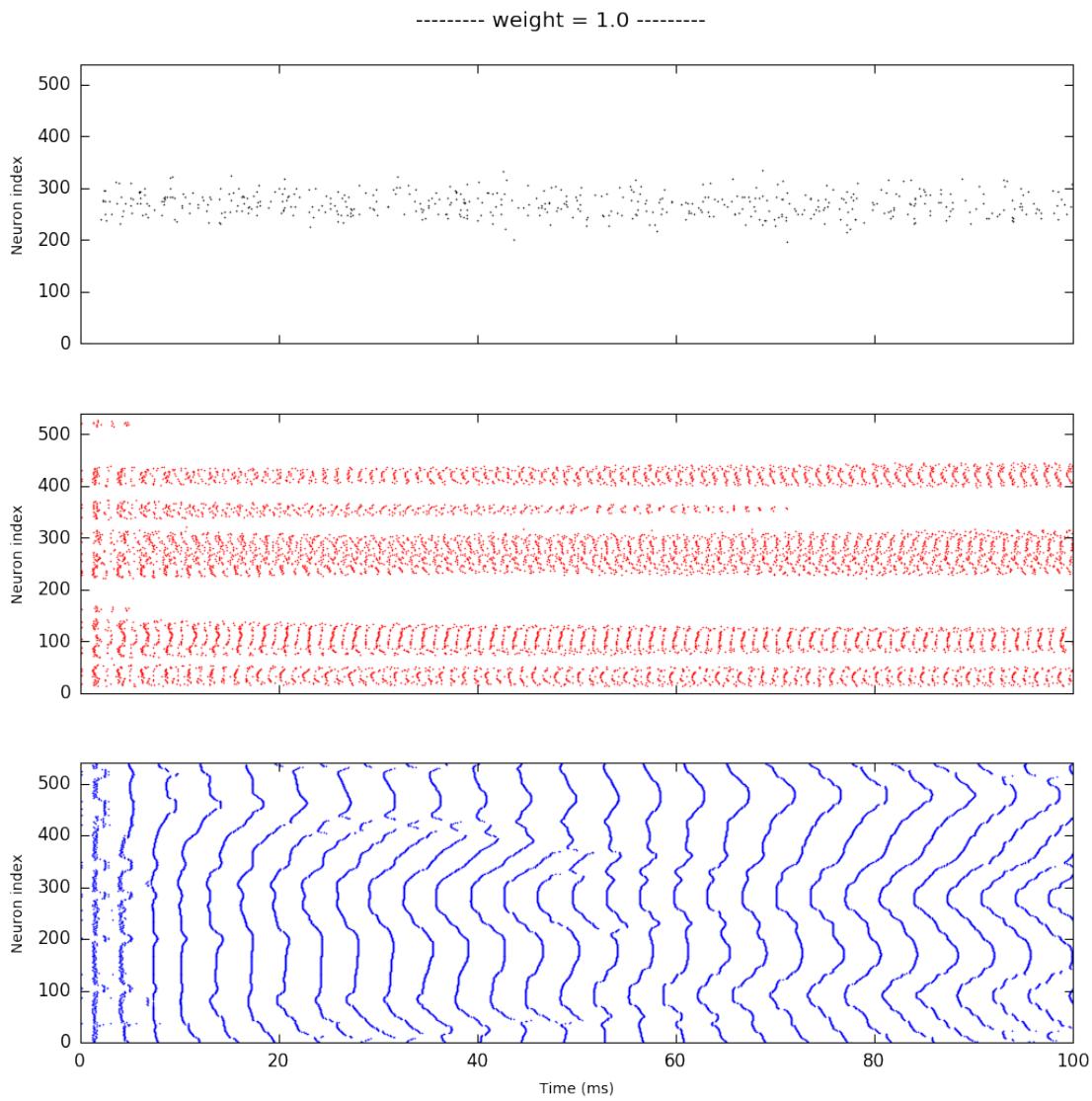


----- weight = 0.316227766017 -----



----- weight = 0.56234132519 -----



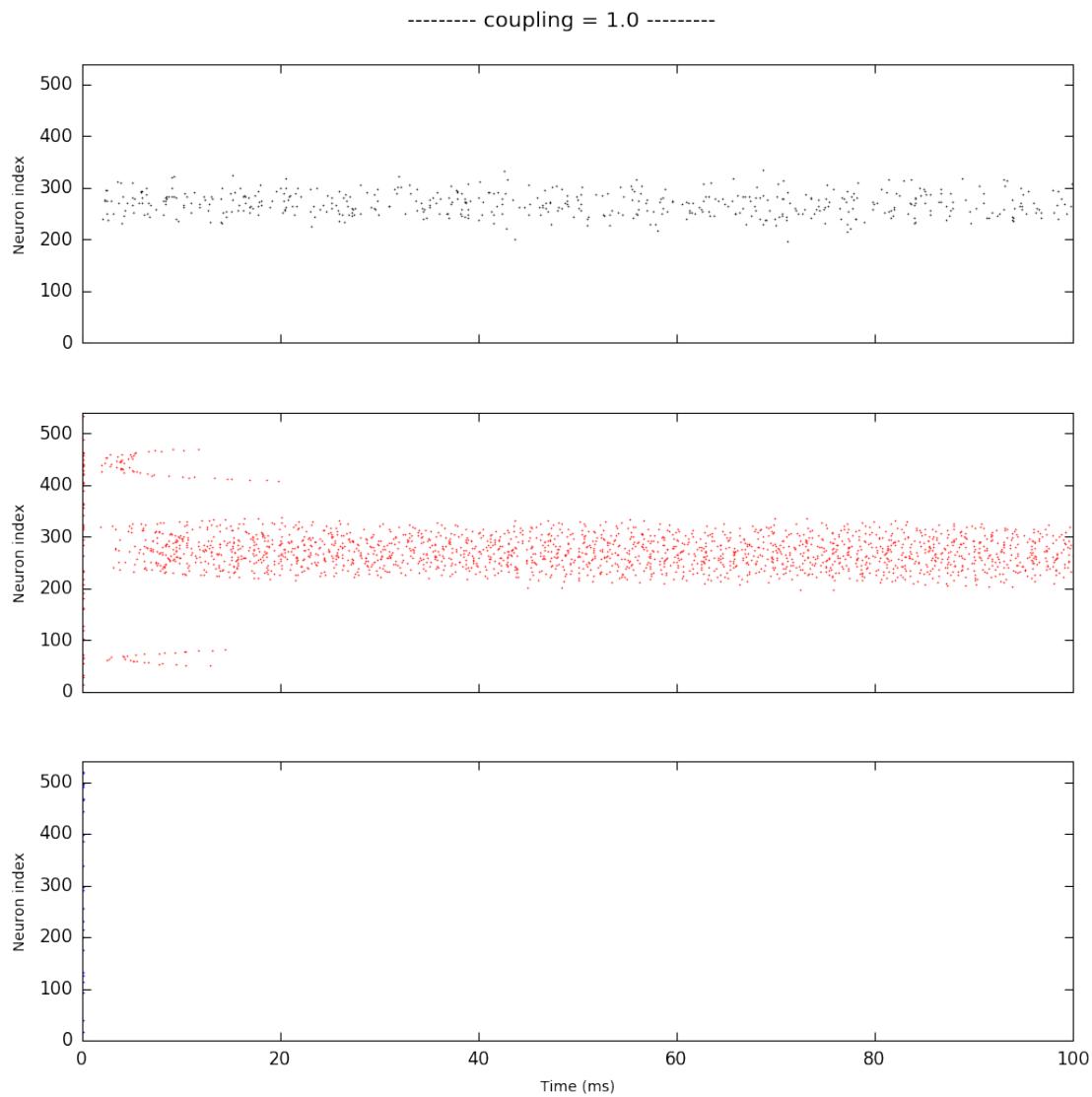


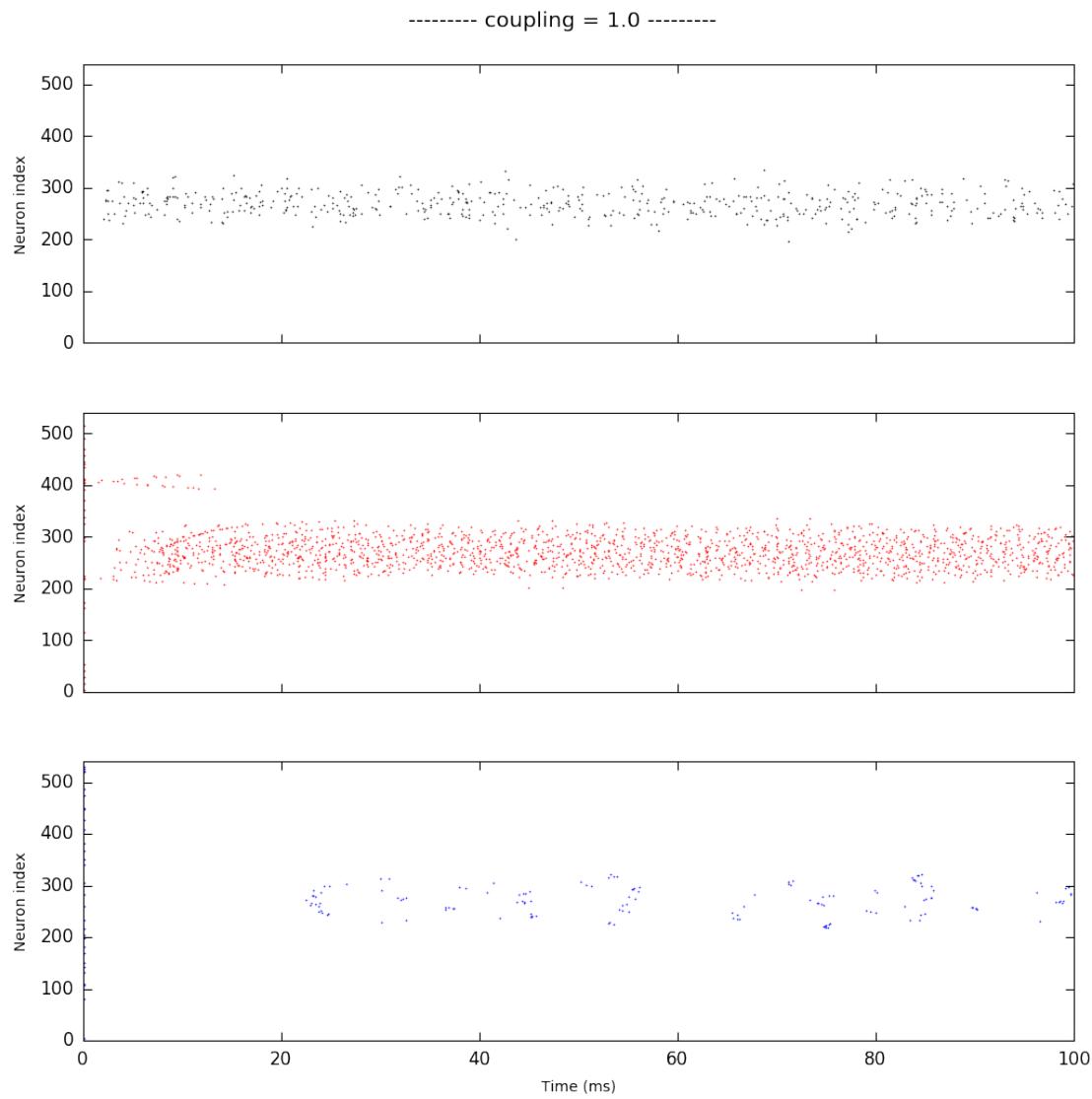
```

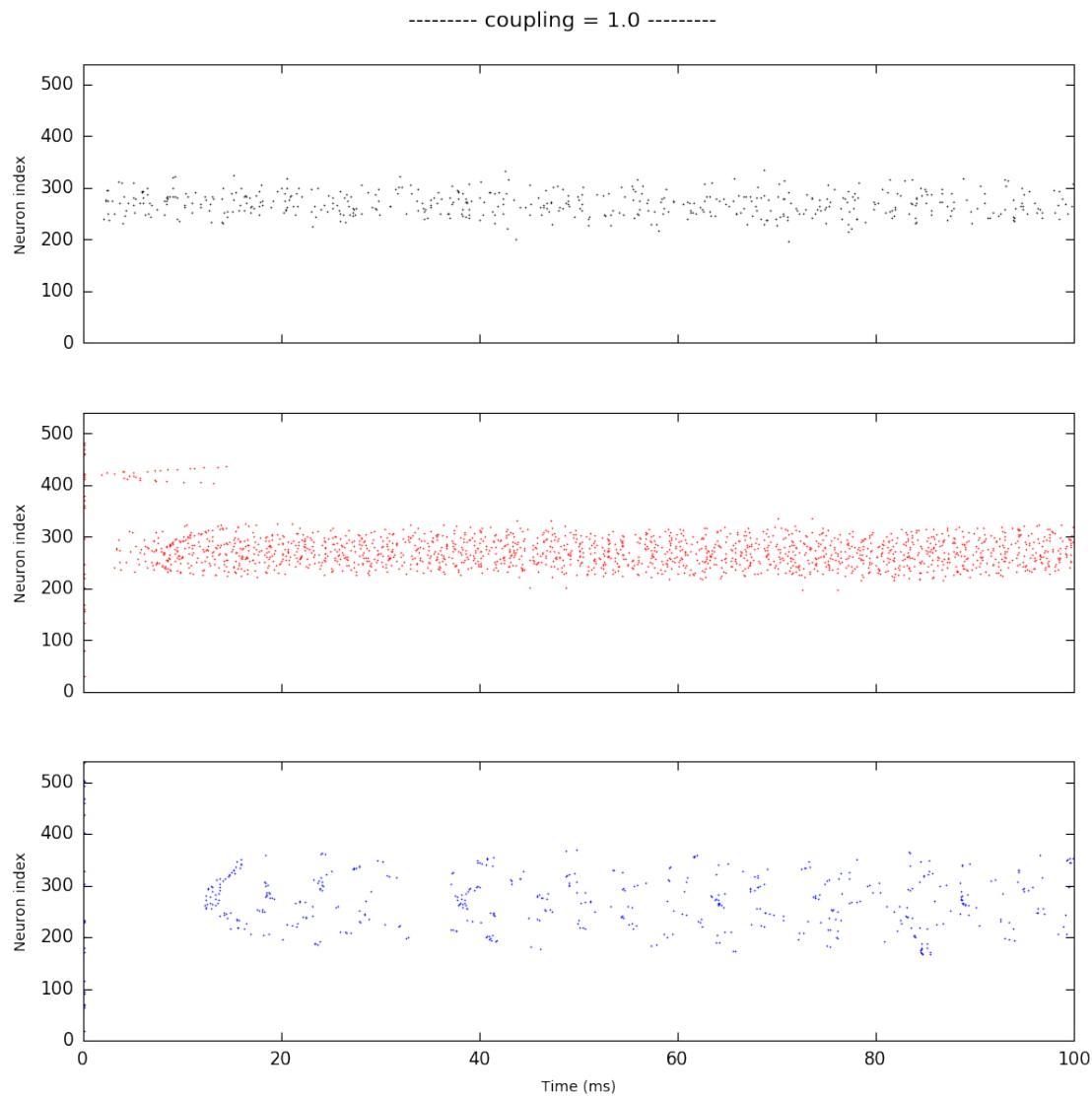
>>> net = RRNN()
... gs = net.g * np.logspace(-1, 1, 9)
... for g in gs:
...     fig, ax = plt.subplots(figsize=(8, 8))
...     net = rec_net(time=time, g=g)
...
...     df, spikesE, spikesI = net.model()
...     fig = Figure(Panel(net.spikesP.spiketrains, xticks=False, yticks=True,
...                         title='---- coupling = {} ----'.format(str(w)))
...                 Panel(net.spikesE.spiketrains, xticks=False, yticks=True,
...                         title='---- coupling = {} ----'.format(str(w)))
...                 Panel(net.spikesI.spiketrains, xlabel="Time (ms)", xticks=True,
...                         title='---- coupling = {} ----'.format(str(w)))
...                 title='---- weight = {} ----'.format(str(g)))
...     fig.show()

```

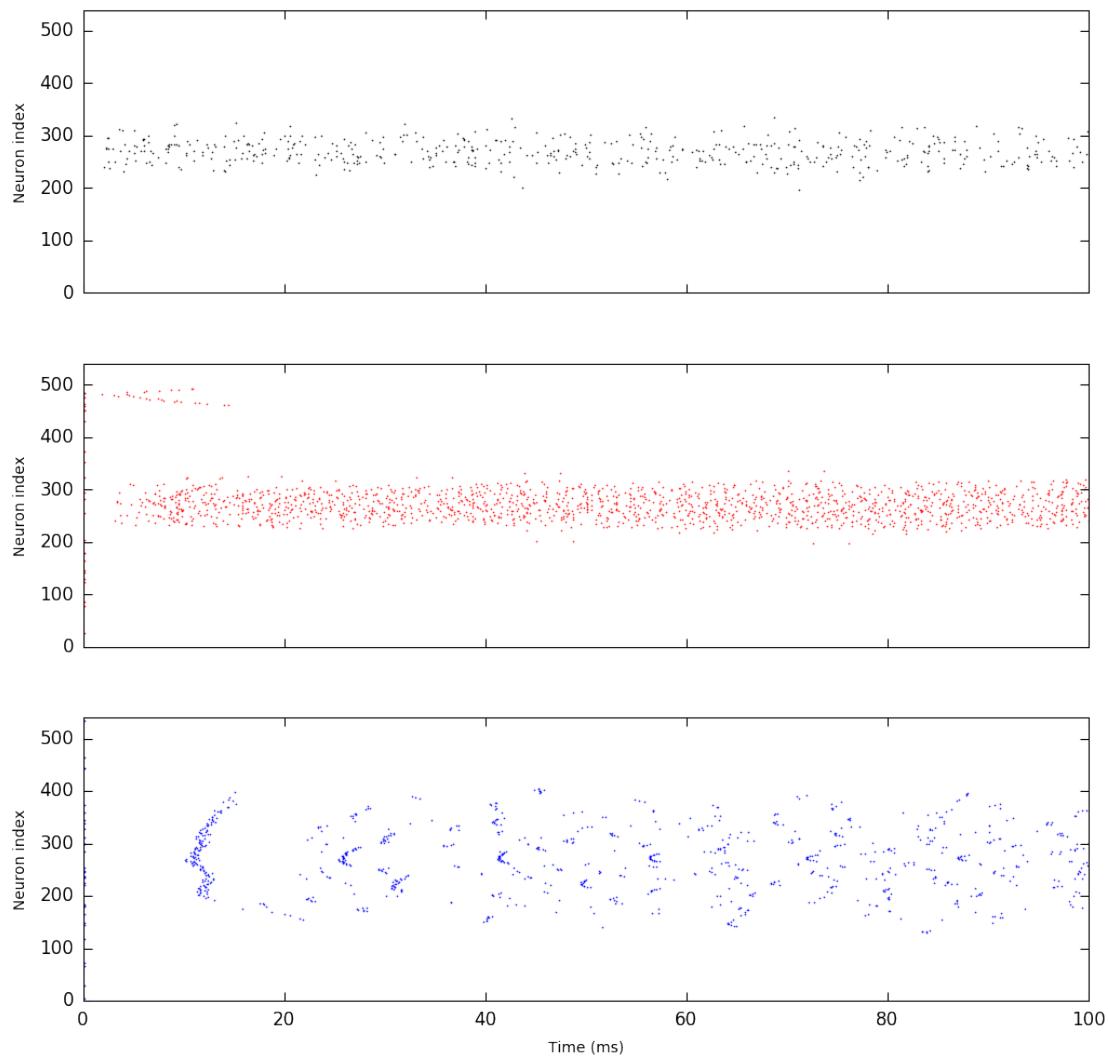
```
...     plt.show()
```



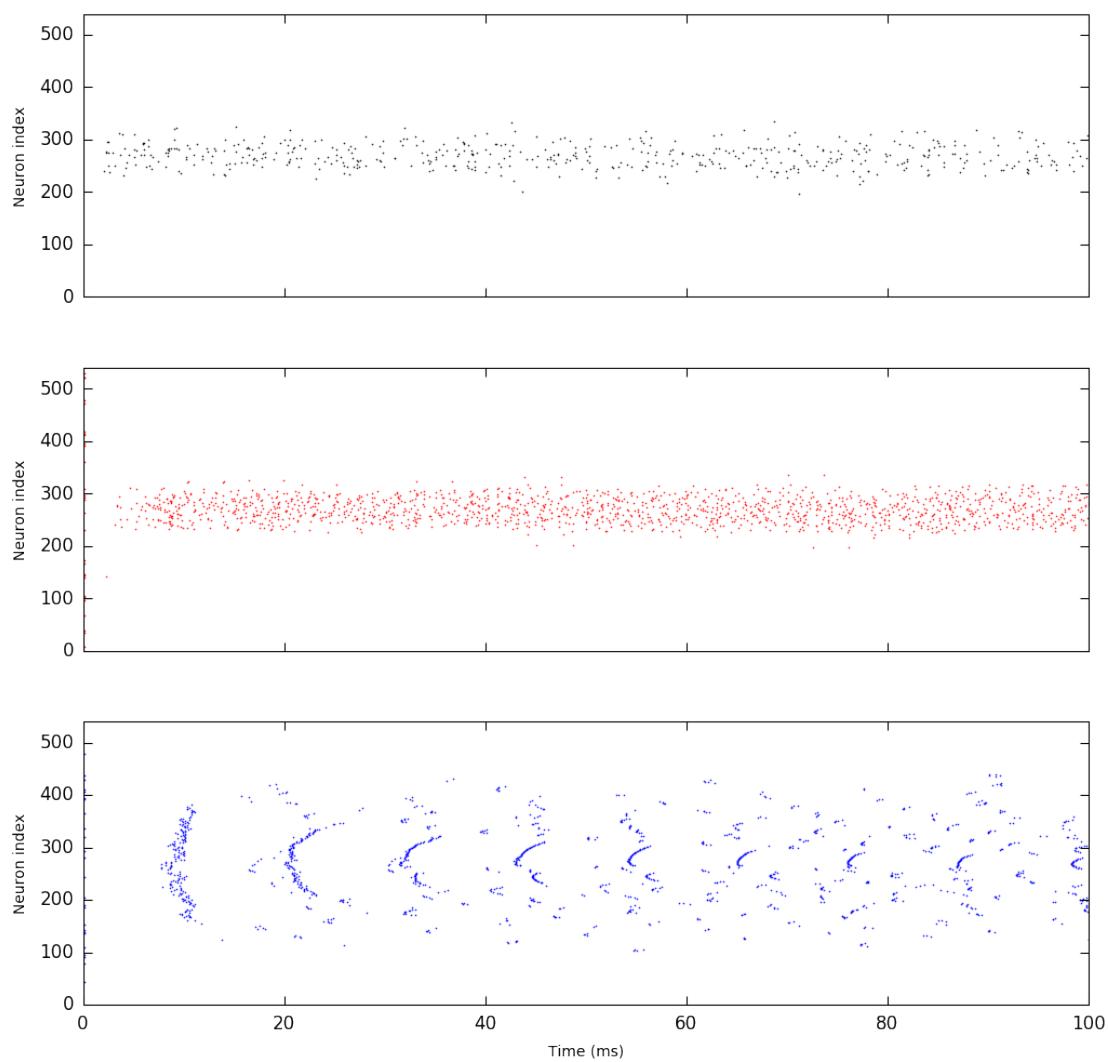


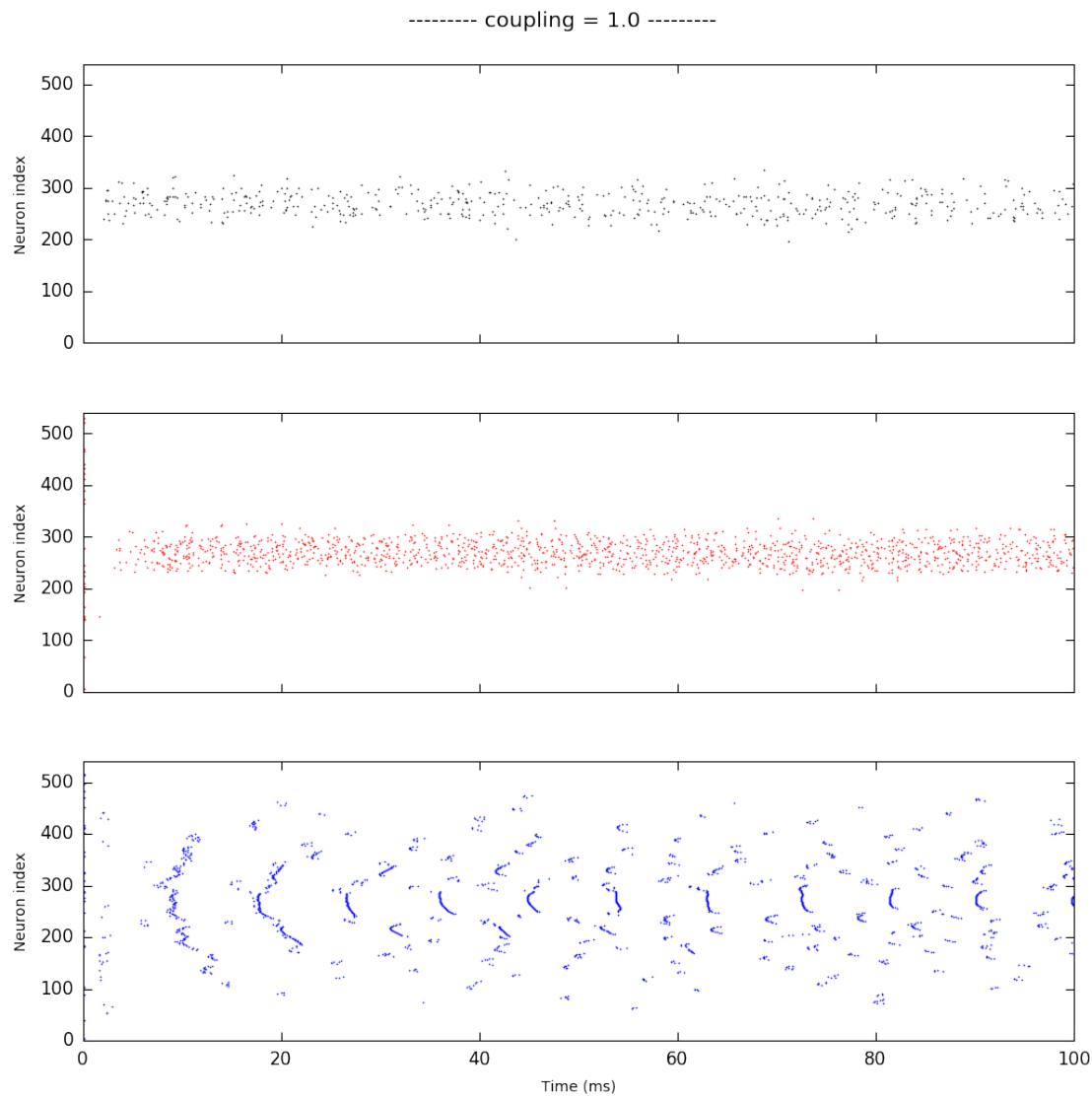


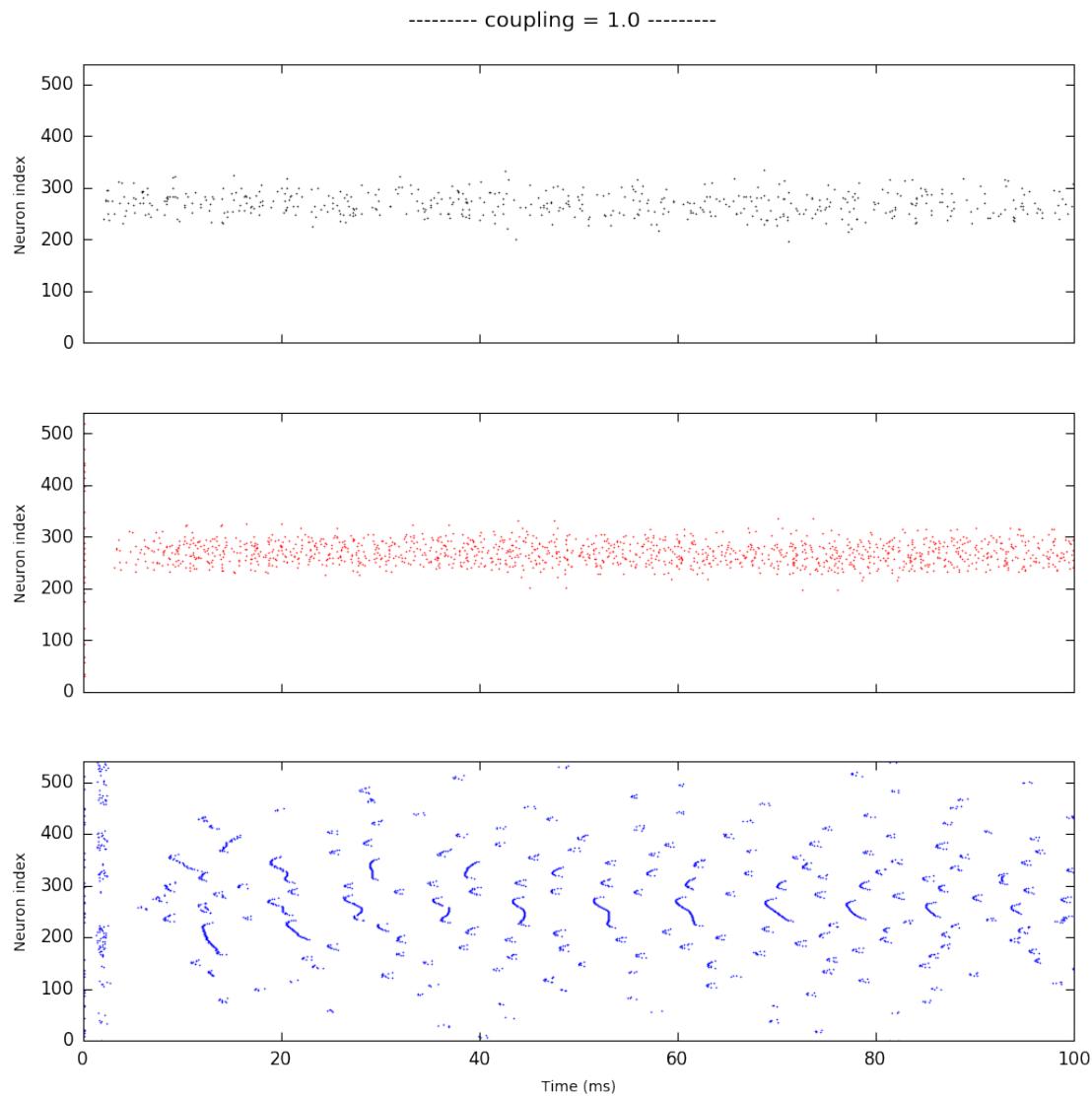
----- coupling = 1.0 -----



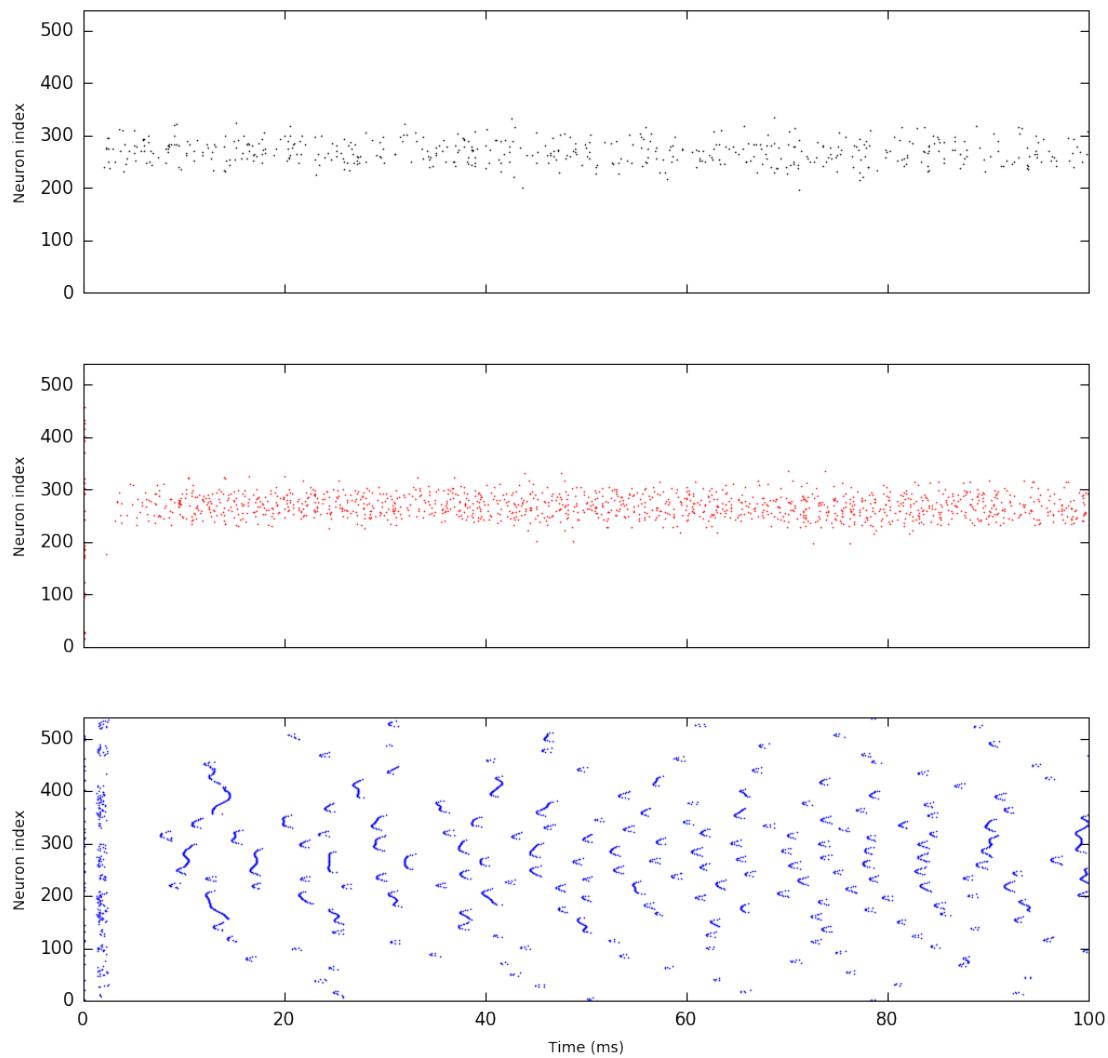
----- coupling = 1.0 -----

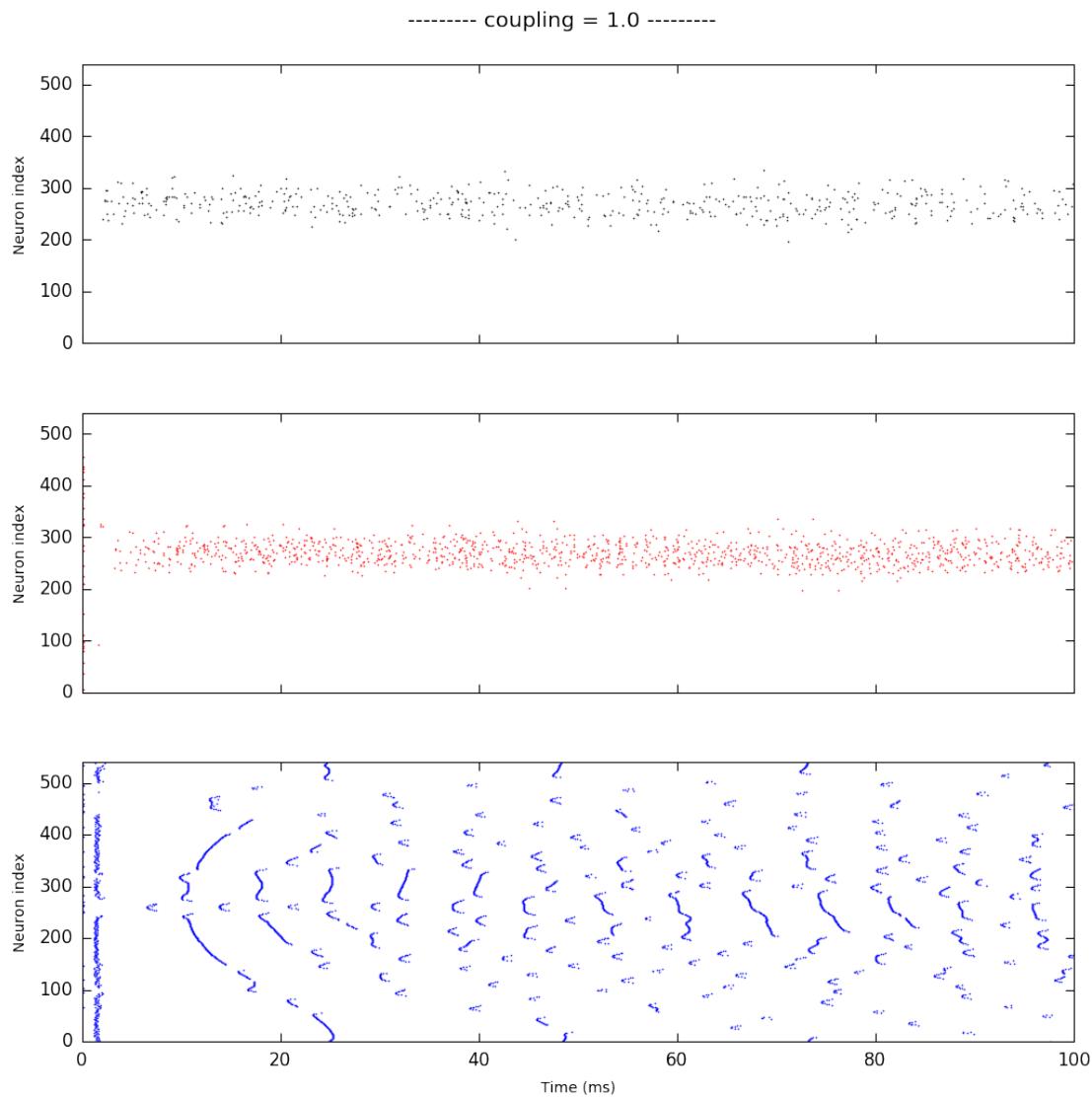






----- coupling = 1.0 -----





3.2.3 Courbes d'accord d'un ring recurrent

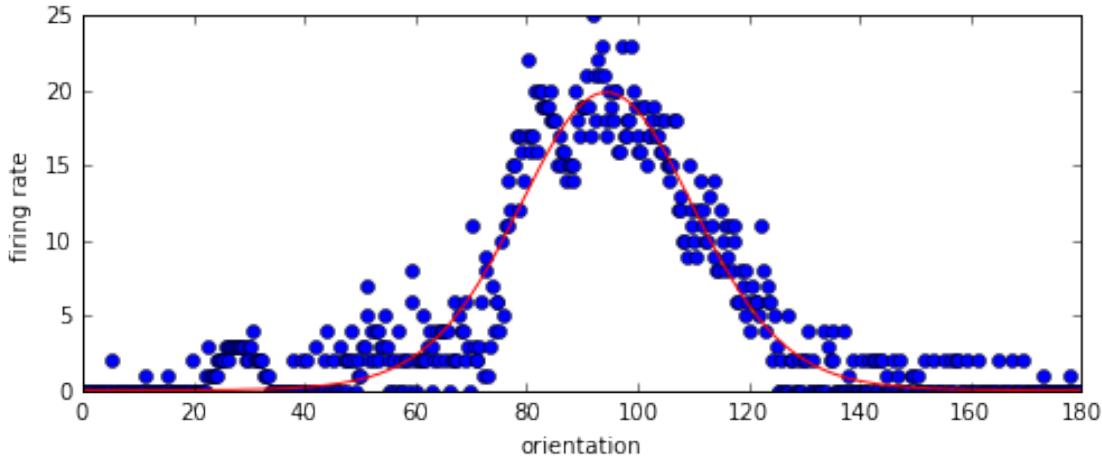
(1 : quelle) Pour démontrer notre démarche, nous allons maintenant appliquer des entrées sélectives à un réseau de connectivité aléatoire, trouver la courbe de selectivité (de type von Mises) qui correspond à la meilleure courbe d'accord sur les fréquences de décharge.

(2 : comment)

(3 : résultat) On observe une transmission parfaite de l'information lorsque les poids internes sont nuls (couplage nul) puis graduellement une diffusion de cette information. XXX Maintenant, on va étudier le comportement d'un réseau avec une connectivité en ring.

— Fitting network response with Von Mises

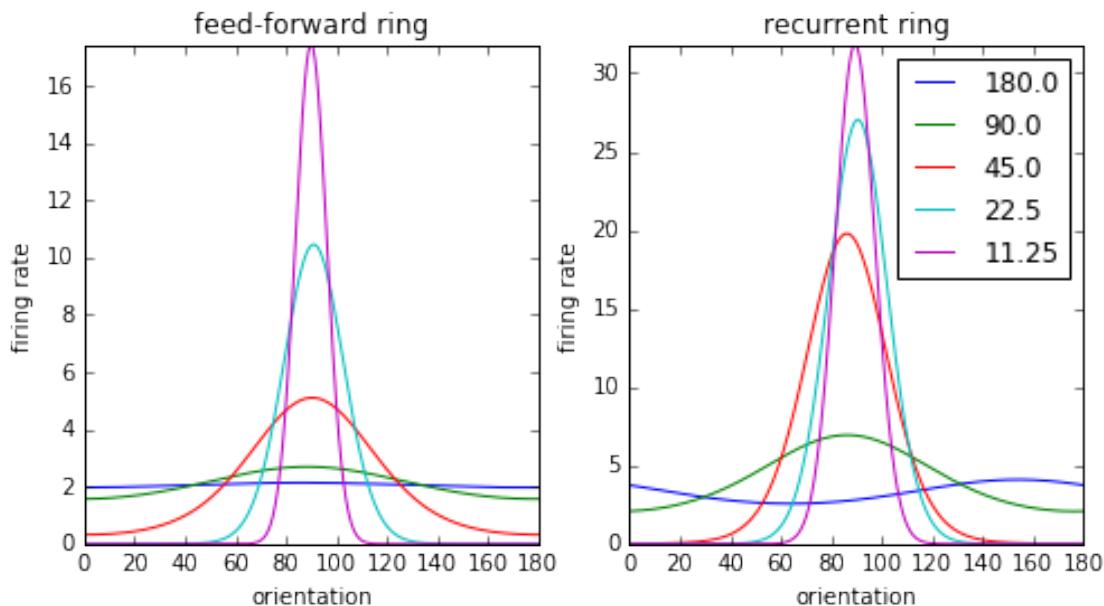
```
>>> time = 100
... net = RRNN(time=time)
... w, g, w_in = net.w, net.g, net.w_in
... def rec_net(time=time, w=w, g=g,
...             b_input=40, b_exc_inh=50, b_exc_exc=4, b_inh_exc=50, b_inh_inh=10,
...             c=1):
...     net = RRNN(time=time, g=g, w=w, c=c)
...     net.sim_params['b_input'] = b_input
...     net.sim_params['b_exc_inh'] = b_exc_inh
...     net.sim_params['b_exc_exc'] = b_exc_exc
...     net.sim_params['b_inh_exc'] = b_inh_exc
...     net.sim_params['b_inh_inh'] = b_inh_inh
...     return net
...
... net = rec_net(time=time)
... df, spikesE, spikesI = net.model()
...
... theta, fr, result = net.fit_vonMises(spikesE)
...
... fig, ax = plt.subplots(figsize=(8,3))
... ax.plot(theta*180/np.pi, fr, 'bo')
... #plt.plot(x, result.init_fit, 'k--')
... ax.plot(theta*180/np.pi, result.best_fit, 'r-')
... ax.axis('tight')
... ax.set_xlabel('orientation')
... ax.set_ylabel('firing rate')
... ax.set_ylim(0)
... plt.show()
```



```

>>> bw_values = 180*np.logspace(0, -4, 5, base=2)
... fig, axs = plt.subplots(1, 2, figsize=(8, 4))
... for i in range(2):#, w in enumerate([0., .32]):
...     for bw_value in bw_values:
...         if i==1:
...             net = rec_net(time=time)
...         else:
...             net = RRNN(time=time, w=0.)
...             net.sim_params['b_input'] = bw_value
...             df, spikesE, spikesI = net.model()
...             theta, fr, result = net.fit_vonMises(spikesE)
...             #print(result.best_fit.mean())
...             axs[i].plot(theta*180/np.pi, result.best_fit, label=str(bw_value))
...
...             axs[i].set_xlabel('orientation')
...             axs[i].set_ylabel('firing rate')
...             axs[i].axis('tight')
...             axs[i].set_ylim(0)
...     axs[0].set_title('feed-forward ring')
...     axs[1].set_title('recurrent ring')
...
... plt.legend(loc='best')
... plt.show()
...
... fig.savefig('../figs/ring_feed-forward_vs_recurrent.png', dpi = 600)

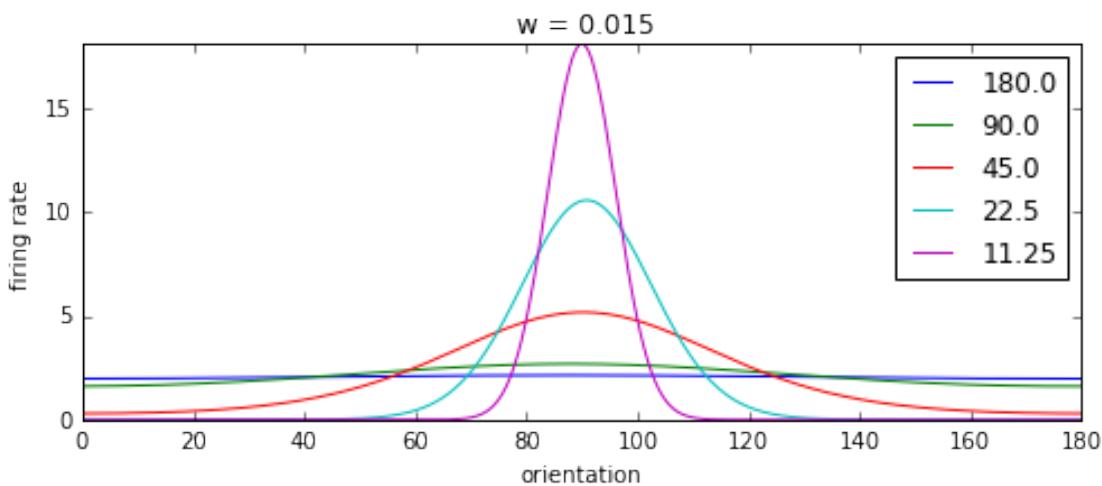
```



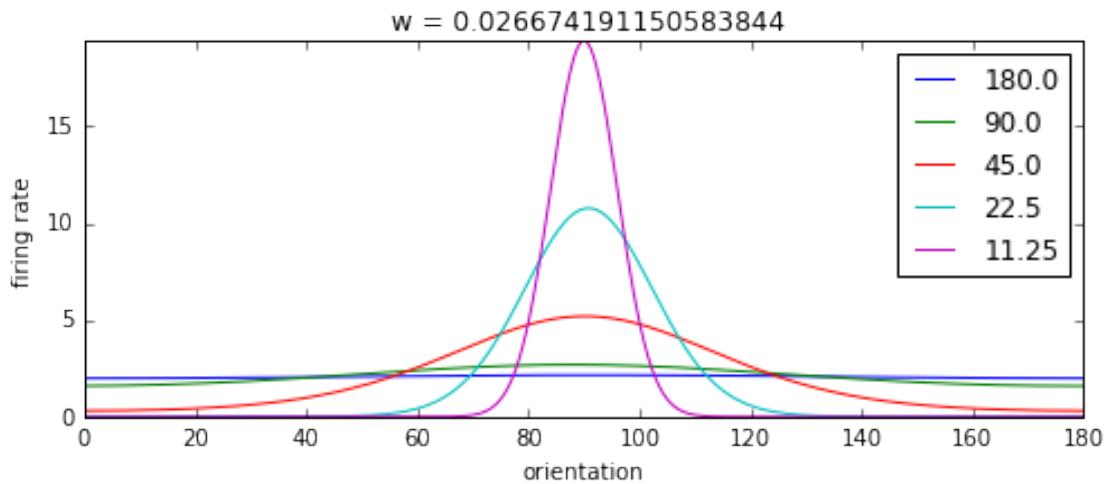
— fit en fonction de la bandwidth

```
>>> net = RRNN()
... ws = net.w * np.logspace(-1, 1, 9)
... bw_values = 180*np.logspace(0, -4, 5, base=2)
... for w in ws:
...     fig, ax = plt.subplots(figsize=(8,3))
...     for bw_value in bw_values:
...         net = rec_net(time=time, w=w, b_input=15, b_exc_inh=30, b_exc_ex=0)
...         net.sim_params['b_input'] = bw_value
...         df, spikesE, spikesI = net.model()
...         theta, fr, result = net.fit_vonMises(spikesE)
...         print(result.best_fit.mean(), result.best_fit.std())
...         ax.plot(theta*180/np.pi, result.best_fit, label=str(bw_value))
...
...         ax.set_title(' w = {}'.format(w))
...         ax.set_xlabel('orientation')
...         ax.set_ylabel('firing rate')
...         ax.axis('tight')
...         ax.set_ylim(0)
...         plt.legend()
...         plt.show()

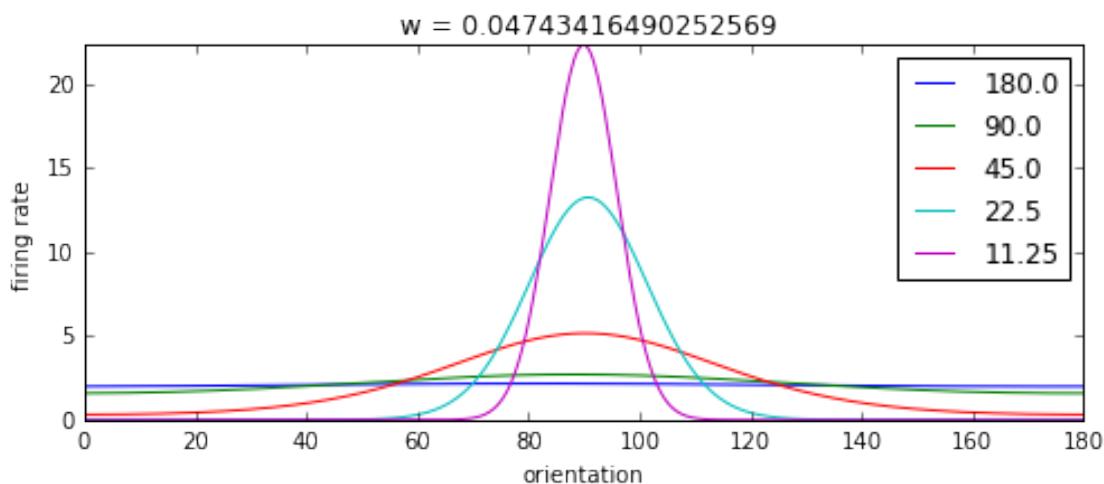
2.06484811963 0.0588356157063
2.11980842389 0.382812351605
1.96324217503 1.69320439908
1.76217304668 3.15238311366
1.55245822896 4.17594743067
```



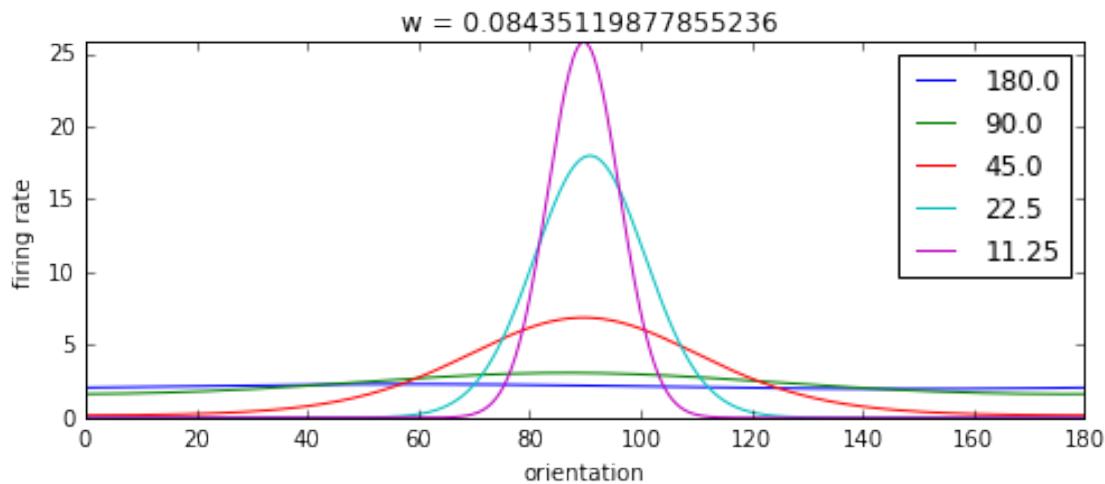
2.06671575694 0.0609751021058
 2.10320673398 0.387101557429
 1.97369501864 1.69272286763
 1.76568037822 3.18893030834
 1.62473918018 4.42669187387



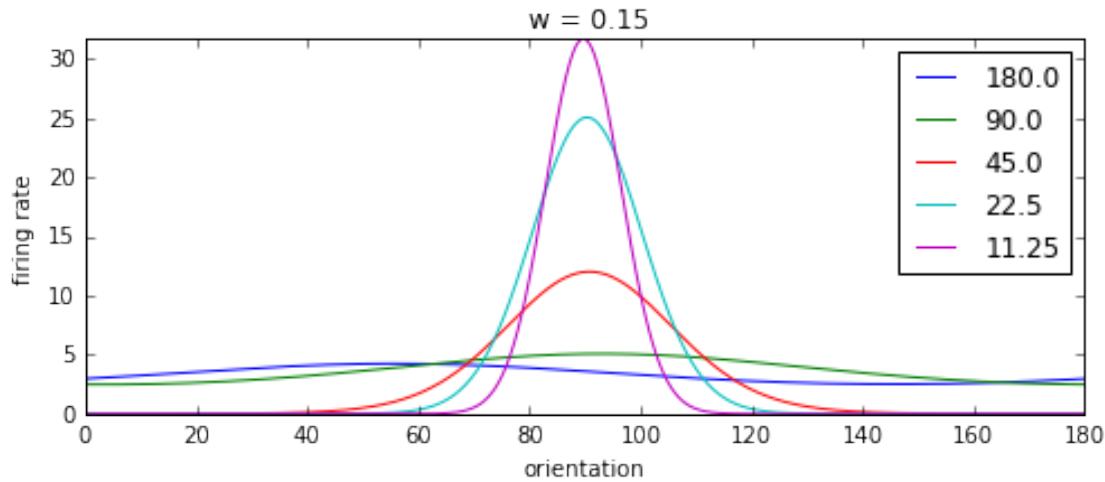
2.07410901984 0.062133989766
 2.10878287978 0.396174157696
 1.97882998123 1.68042676047
 2.00255601231 3.81607792362
 1.87883201424 5.10304996287



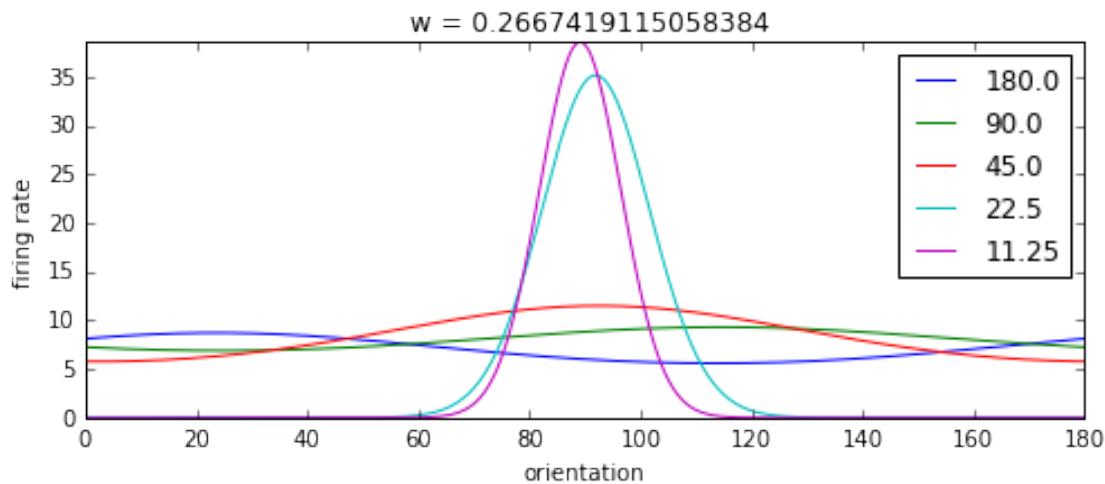
2.13872646967 0.116849316175
 2.29651969966 0.513820278243
 2.21906923955 2.29494256463
 2.54956379143 5.06509811984
 2.31106241496 6.05546169551



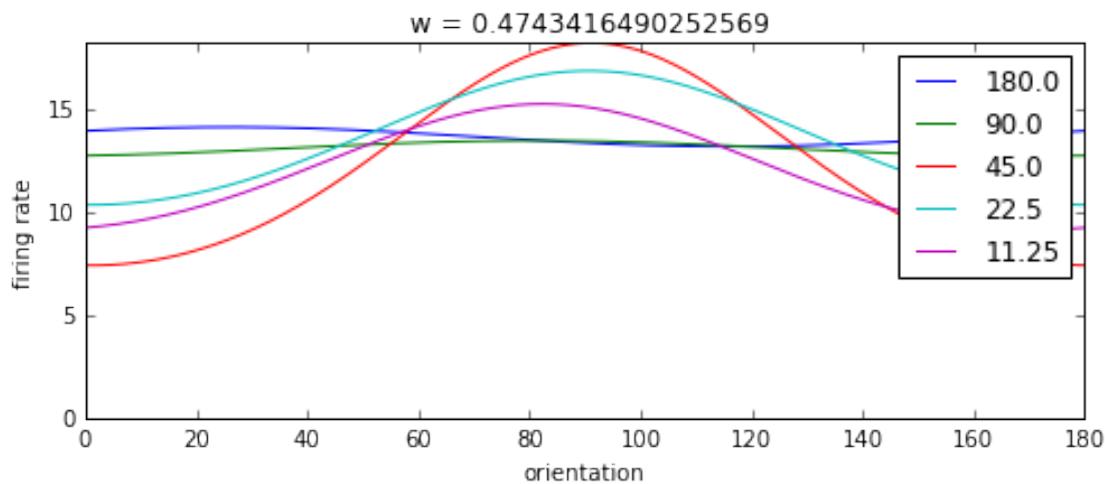
3.32886274965 0.612714672663
 3.66668734095 0.919737392652
 2.58908154481 3.85059804461
 3.51105873022 7.02905276606
 3.00734787959 7.62534118326



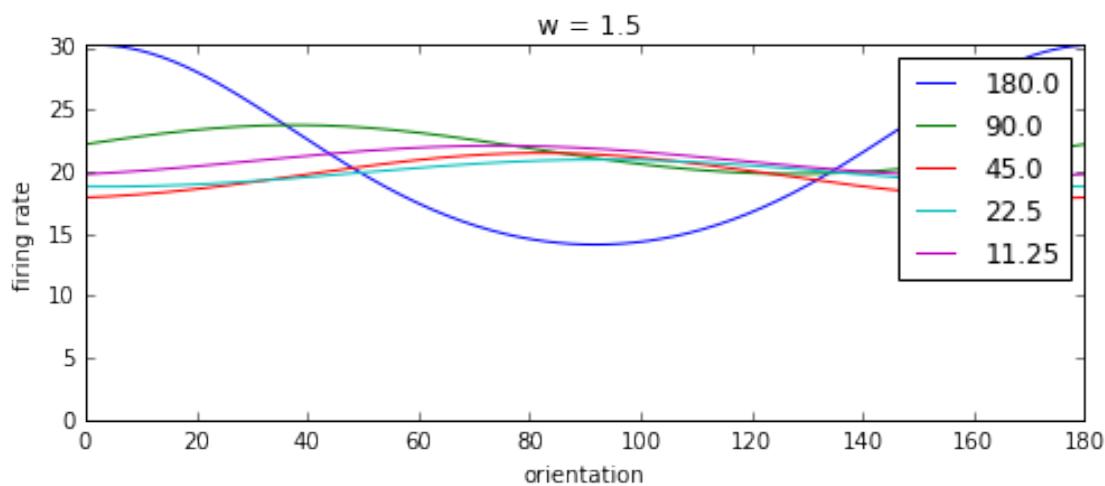
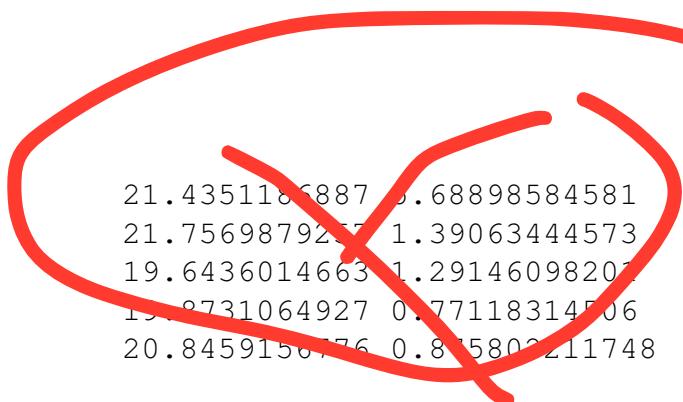
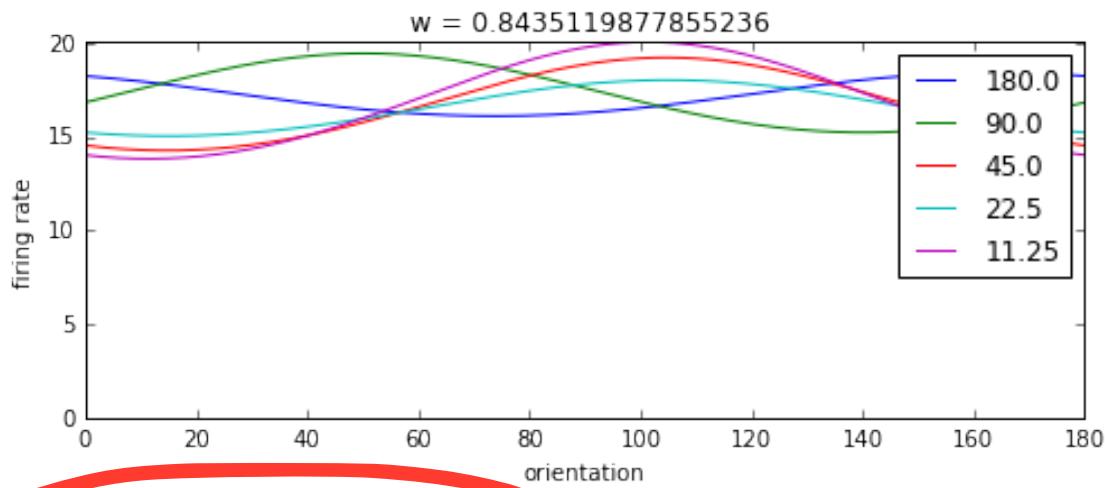
7.0702196689 1.10524333454
 8.04798744986 0.85577643094
 8.38372563761 2.02273903911
 4.79914195556 9.76387300073
 3.95701580824 9.58062701767



13.6665335655 0.334510419571
 13.1075521405 0.265466169102
 12.2145828302 3.81879347851
 13.4133141434 2.30532636651
 12.0153290216 2.15696024817



17.2391346558 0.822450060006
 17.2615127307 1.49040960112
 16.6368762854 1.75370442962
 16.4856706299 1.05578008597
 16.7766678563 2.20264383832

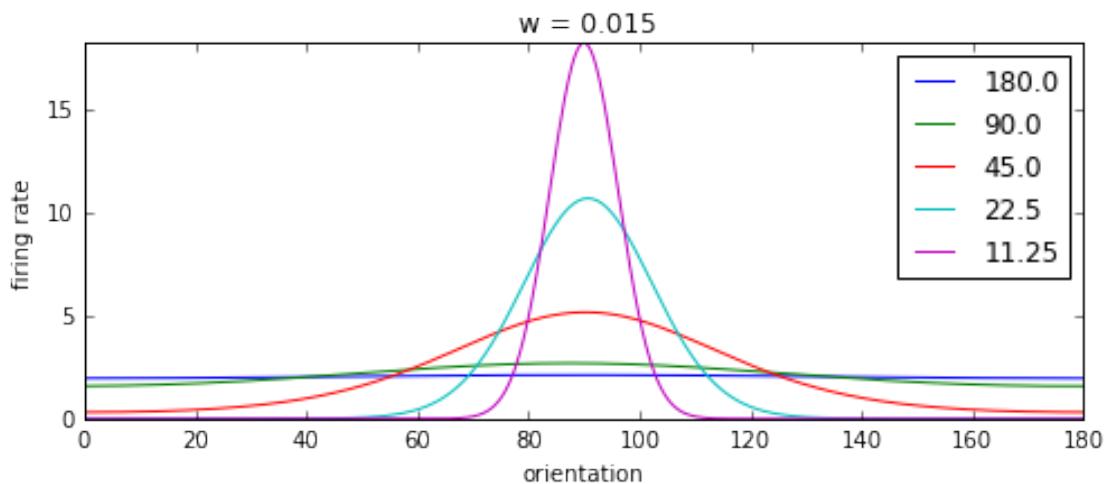


```

>>> for w in ws:
...     bw_values = 180*np.logspace(0, -4, 5, base=2)
...     fig, ax = plt.subplots(figsize=(8, 3))
...     for bw_value in bw_values:
...         net = RRNN(time=time, w=w, c=1)
...         net.sim_params['b_input'] = bw_value
...         net.sim_params['b_exc_inh'] = 30
...         net.sim_params['b_exc_exc'] = 5
...         net.sim_params['b_inh_exc'] = 30
...         net.sim_params['b_inh_inh'] = 5
...
...         df, spikesE, spikesI = net.model()
...         theta, fr, result = net.fit_vonMises(spikesE)
...         print(result.best_fit.mean(), result.best_fit.std())
...         ax.plot(theta*180/np.pi, result.best_fit, label=str(bw_value))
...
...         ax.set_title(' w = {}'.format(w))
...         ax.set_xlabel('orientation')
...         ax.set_ylabel('firing rate')
...         ax.axis('tight')
...         ax.set_ylim(0)
...         plt.legend()
...         plt.show()

2.03892548896 0.0576362680085
2.10145633305 0.392921301163
1.9742403908 1.68084471796
1.76332086899 3.17011437161
1.56357790916 4.19867203651

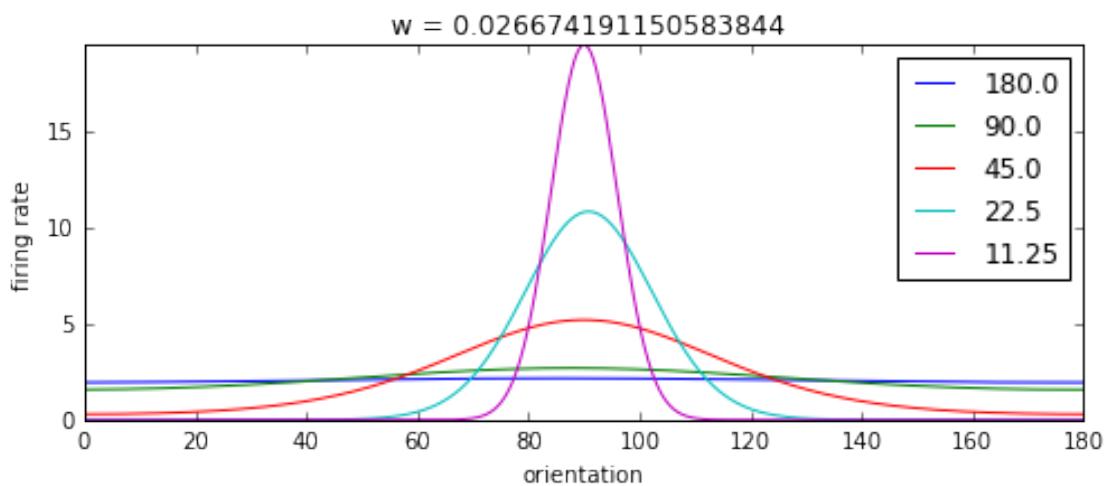
```



```

2.05376149347 0.0772960561586
2.09582684686 0.394702912316
1.96275242552 1.69758881835
1.76988960495 3.20318096702
1.62704361621 4.43960733999

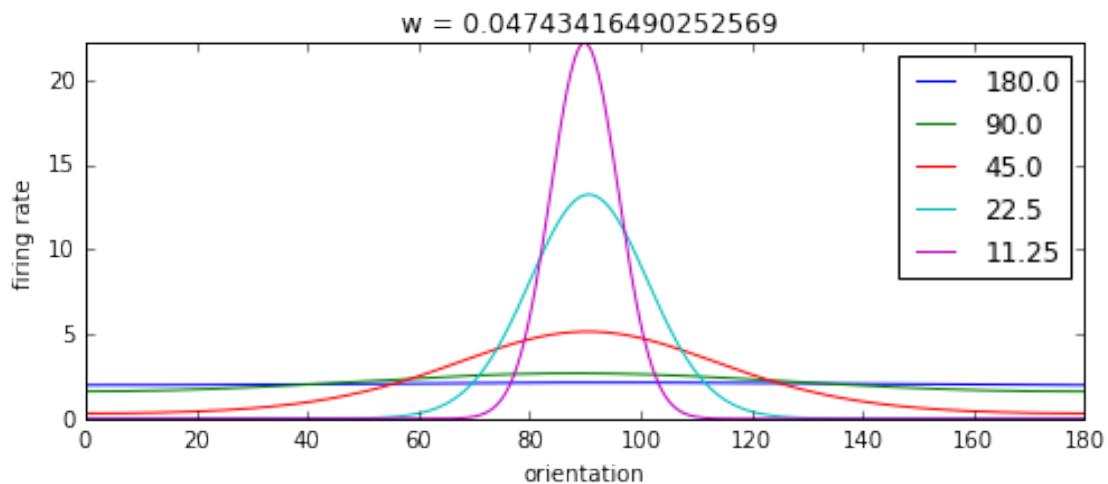
```



```

2.06114644739 0.054169230559
2.11409241771 0.379808563861
1.97067583949 1.6772113525
2.00496707669 3.81818510172
1.88371922264 5.09216723952

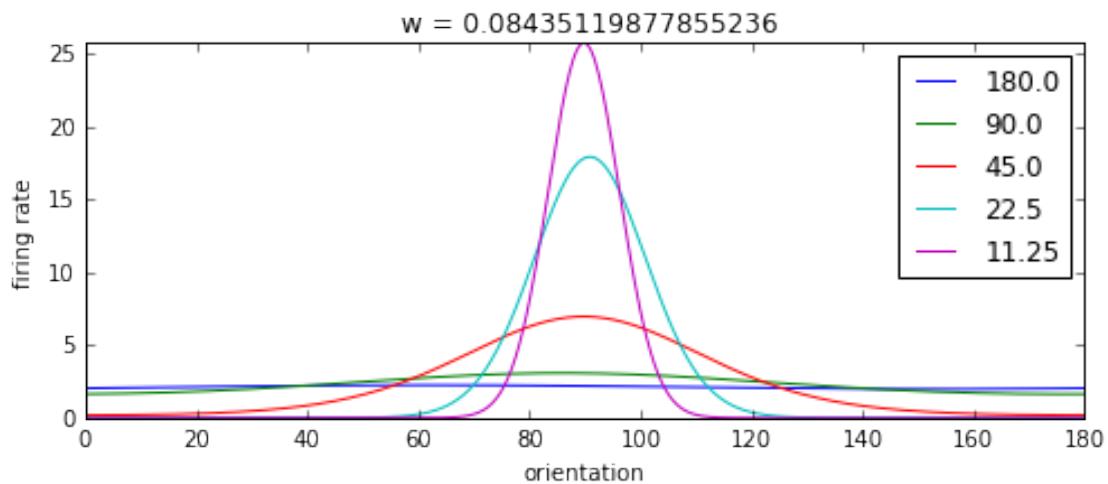
```



```

2.11660185958 0.097904333241
2.30240326428 0.511395099635
2.28414016769 2.32250268428
2.5498997864 5.0568738799
2.30599747826 6.04732688357

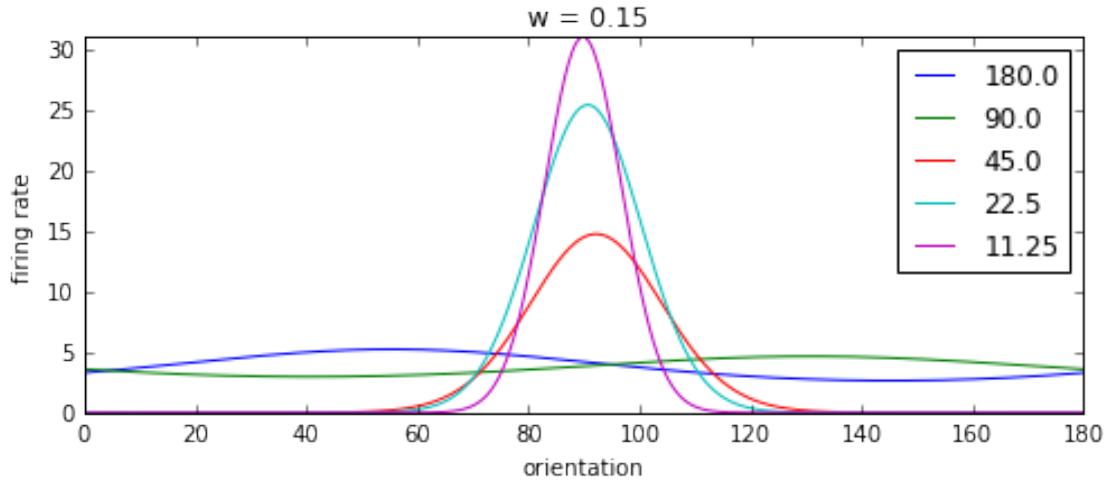
```



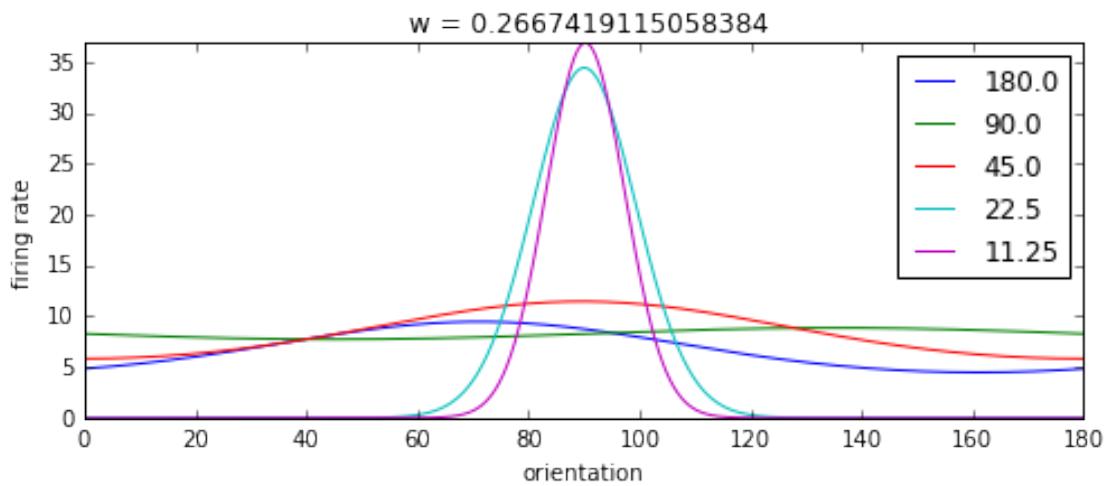
```

3.84193340308 0.91512342419
3.77482458031 0.596730720132
2.49447147282 4.41928112271
3.47515157773 7.07242810187
2.99977629593 7.52880625125

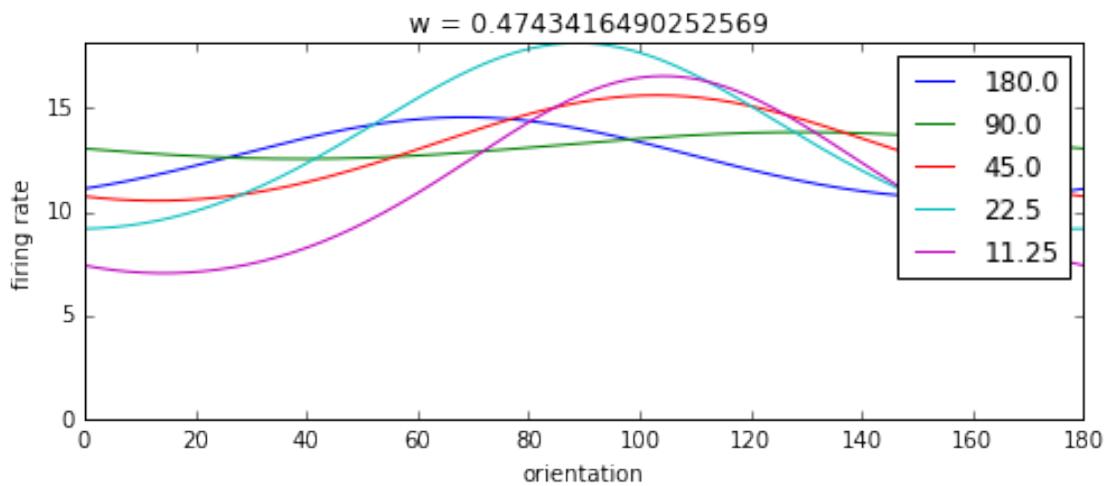
```



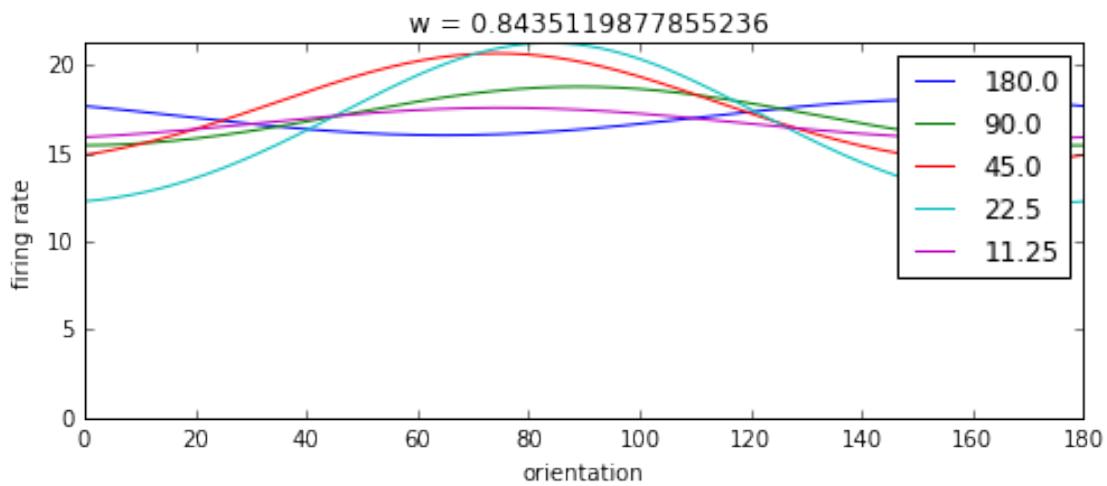
6.73325229599 1.75918801507
 8.27972805587 0.398315176691
 8.39369456563 1.9885446837
 4.50670921966 9.41203936595
 3.63589224771 9.00989985922



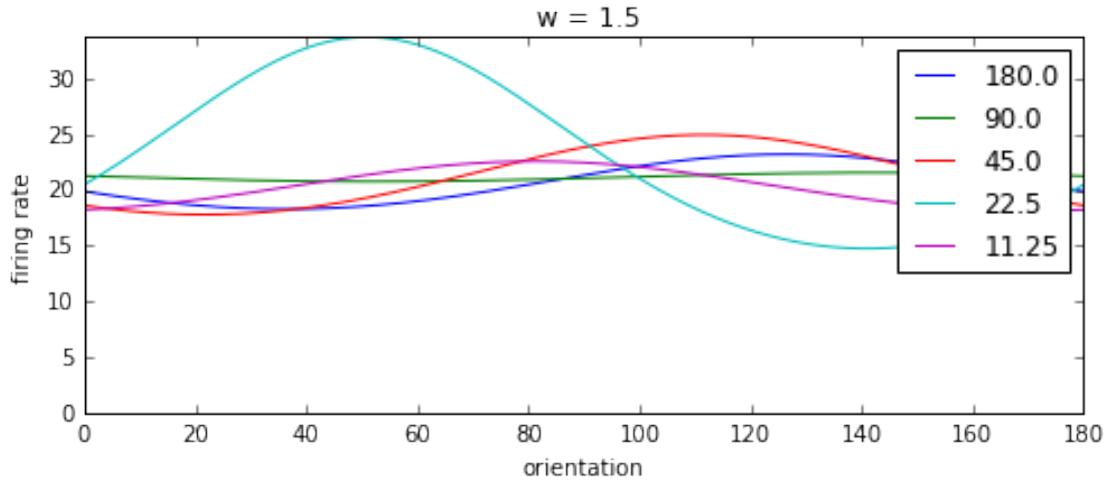
12.5291090355 1.38242942646
 13.1887481895 0.455725802122
 12.9473316832 1.8011879544
 13.2714304335 3.1707952455
 11.2790338901 3.36136869458



17.0352742142 0.719561873374
 17.0771816637 1.1782941147
 17.464259764 2.17259164025
 16.411652502 3.20421417974
 16.6860642997 0.628550080798



20.6948386861 1.74076438928
 21.1833464848 0.282447127588
 21.2414772666 2.53676684582
 23.2881413554 6.70577785233
 20.3154845461 1.57419181739

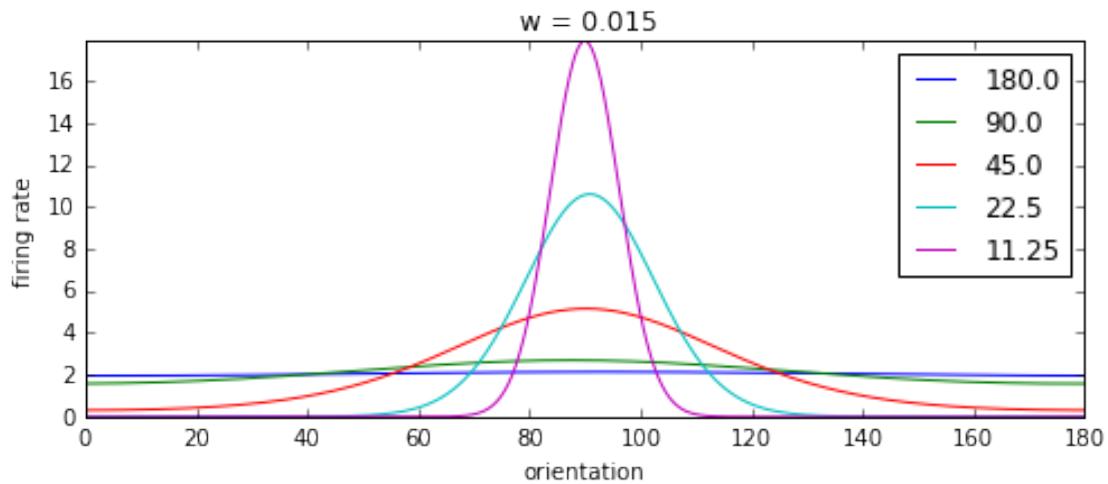


```

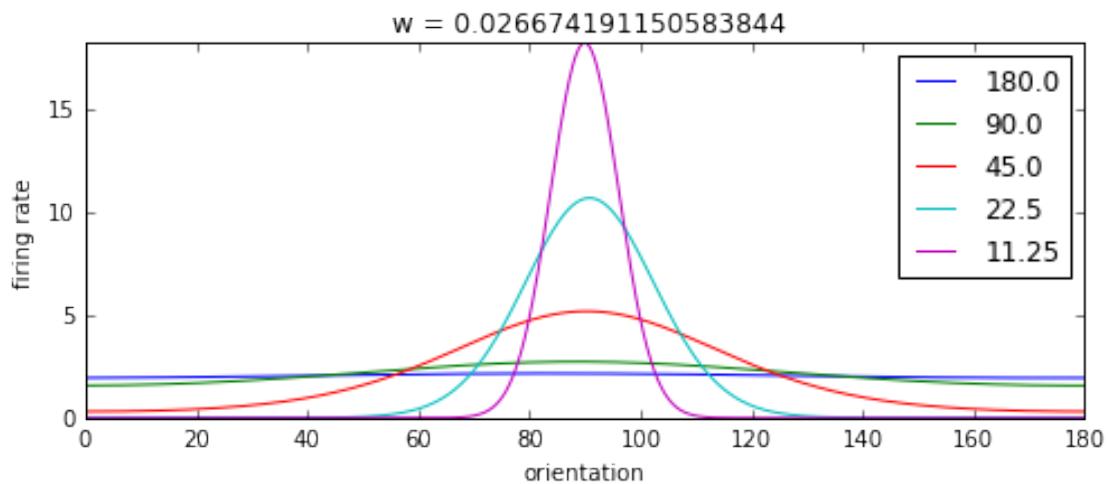
>>> for w in ws:
...     bw_values = 180*np.logspace(0, -4, 5, base=2)
...     fig, ax = plt.subplots(figsize=(8,3))
...     for bw_value in bw_values:
...         net = rec_net(time=time, w=w, b_input=bw_value, b_exc_inh=5, b_e
...         df, spikesE, spikesI = net.model()
...         theta, fr, result = net.fit_vonMises(spikesE)
...         print(result.best_fit.mean(), result.best_fit.std())
...         ax.plot(theta*180/np.pi, result.best_fit, label=str(bw_value))
...
...     ax.set_title(' w = {}'.format(w))
...     ax.set_xlabel('orientation')
...     ax.set_ylabel('firing rate')
...     ax.axis('tight')
...     ax.set_ylim(0)
...     plt.legend()
...     plt.show()

2.04264283045 0.0639917796249
2.09582765678 0.394001260571
1.96500125331 1.67691935458
1.75321700159 3.14802280527
1.54551392648 4.13594659408

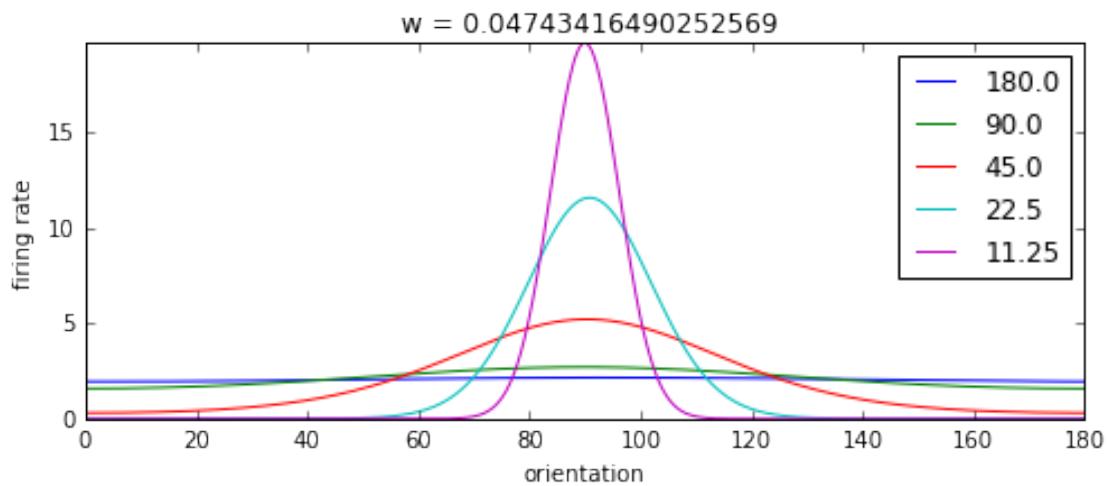
```



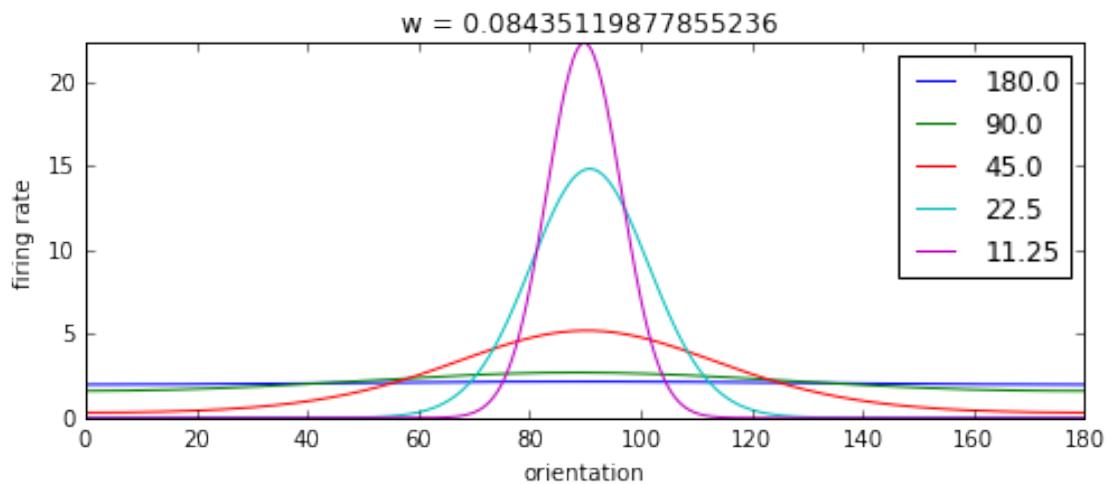
2.0556066113 0.0780127203066
 2.11800859486 0.410739535012
 1.97947988443 1.69595390546
 1.77405754348 3.18451912217
 1.56058785078 4.20471024525



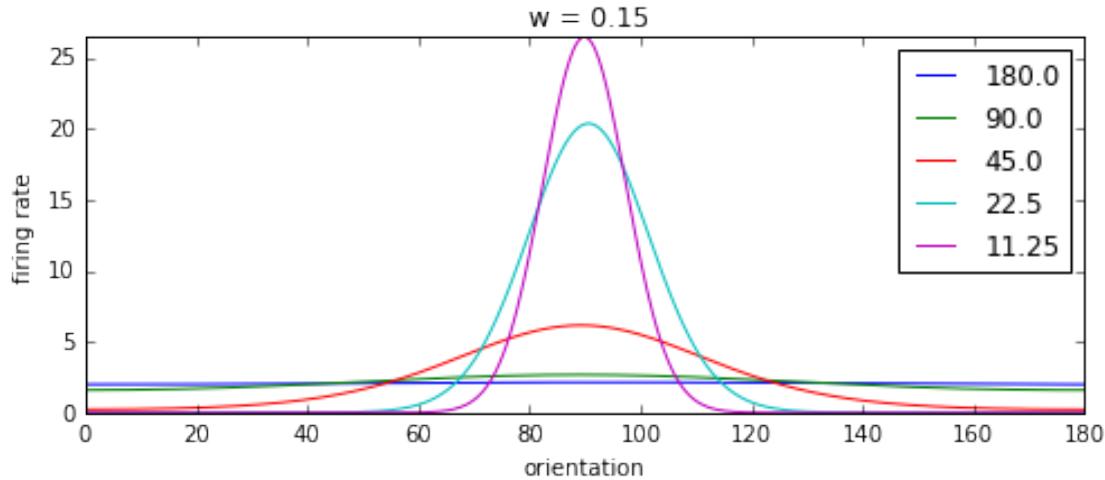
2.05376431209 0.0725434003952
 2.10711187475 0.398563003619
 1.97969820213 1.7021274528
 1.83735819419 3.39431271191
 1.69362028031 4.5454905231



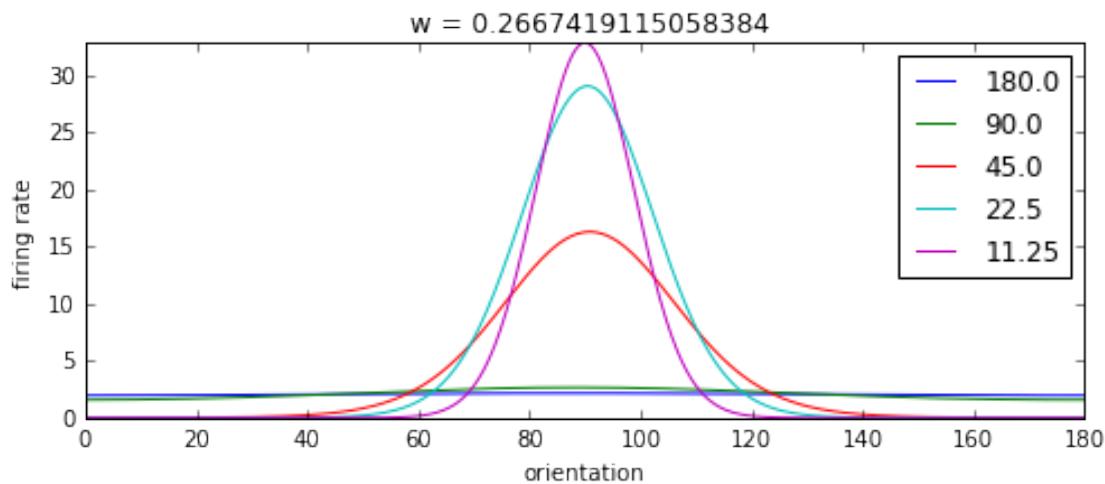
2.07225829399 0.0624159188469
 2.11789045305 0.387062862322
 1.98883876491 1.69478827563
 2.2418843706 4.27564225337
 2.09325122363 5.34569825168



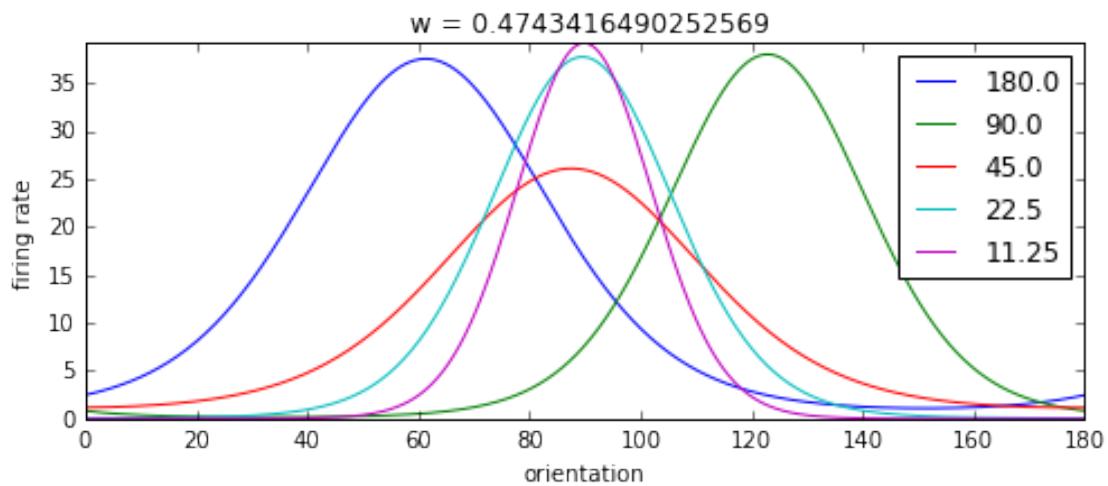
2.07410162507 0.0522894646018
 2.09944106904 0.382468875646
 2.14103412153 2.04227705866
 3.14943974428 5.91501830497
 2.78179229806 6.6397879596



2.06116009597 0.0684562592367
 2.08659836299 0.385977770623
 3.63910024158 5.25652803222
 4.90441190053 8.6826337188
 4.08272867366 8.8008876121



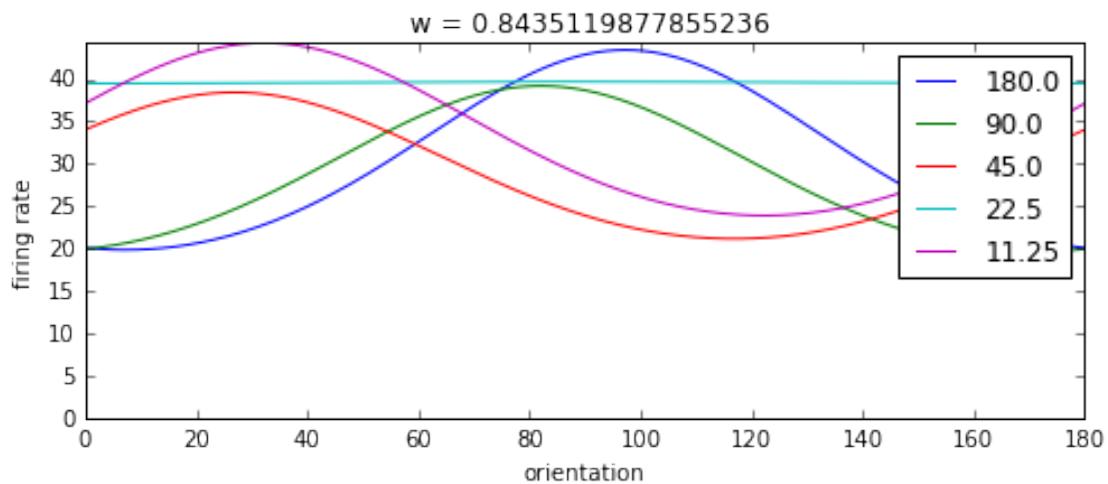
12.4693233057 12.5084588952
 9.7685135618 12.5721111628
 9.36234632942 8.61501425584
 8.85821269216 12.3070993236
 6.75596578086 11.7975914797



```

30.468331526 8.33325327602
28.5809413908 6.89390674039
29.1558254448 6.10167717602
39.5870369802 0.0746316320478
33.3037848139 7.19106001476

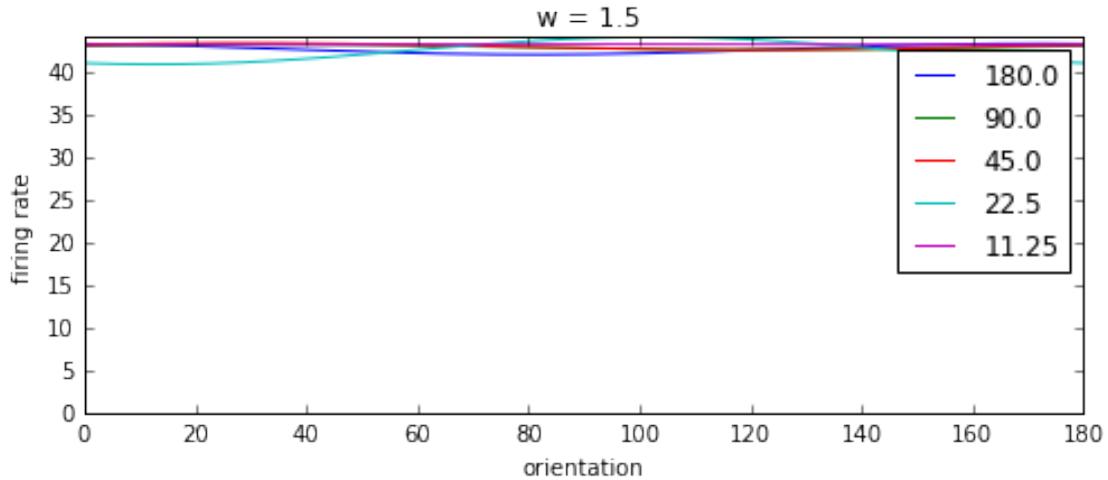
```



```

42.6203768255 0.453942779891
42.8537035268 0.226654648083
42.9203753072 0.288356964973
42.4131592573 1.13167839867
43.1555555564 0.028970070221

```

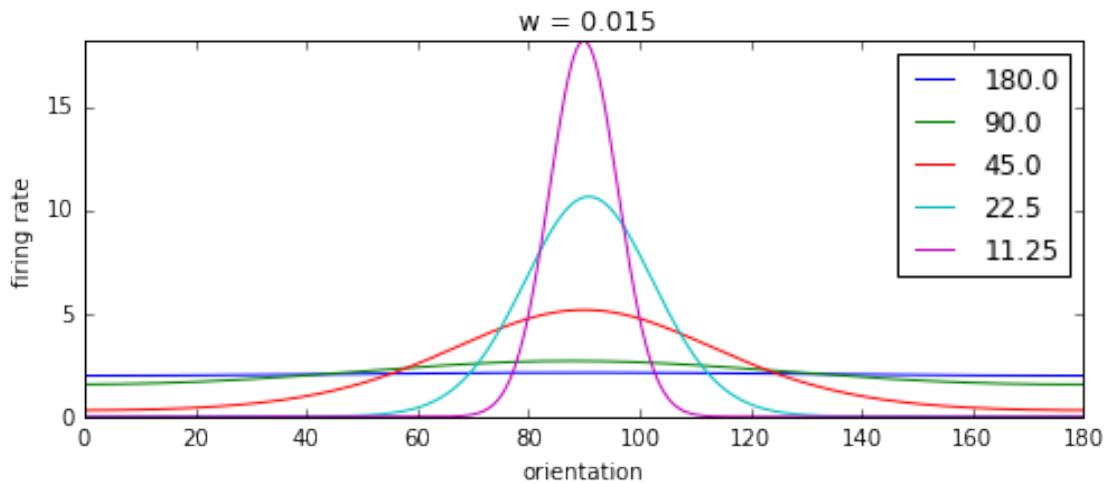


```

>>> for w in ws:
...     bw_values = 180*np.logspace(0, -4, 5, base=2)
...     fig, ax = plt.subplots(figsize=(8,3))
...     for bw_value in bw_values:
...         net = rec_net(time=time, w=w, b_input=bw_value, b_exc_inh=45, b_
...         df, spikesE, spikesI = net.model()
...         theta, fr, result = net.fit_vonMises(spikesE)
...         print(result.best_fit.mean(), result.best_fit.std())
...         ax.plot(theta*180/np.pi, result.best_fit, label=str(bw_value))
...
...     ax.set_title(' w = {}'.format(w))
...     ax.set_xlabel('orientation')
...     ax.set_ylabel('firing rate')
...     ax.axis('tight')
...     ax.set_ylim(0)
...     plt.legend()
...     plt.show()

2.05743540842 0.0530515547288
2.09027474158 0.40209250802
1.97427963555 1.682634724
1.76054752422 3.16120463727
1.55538684829 4.18352943088

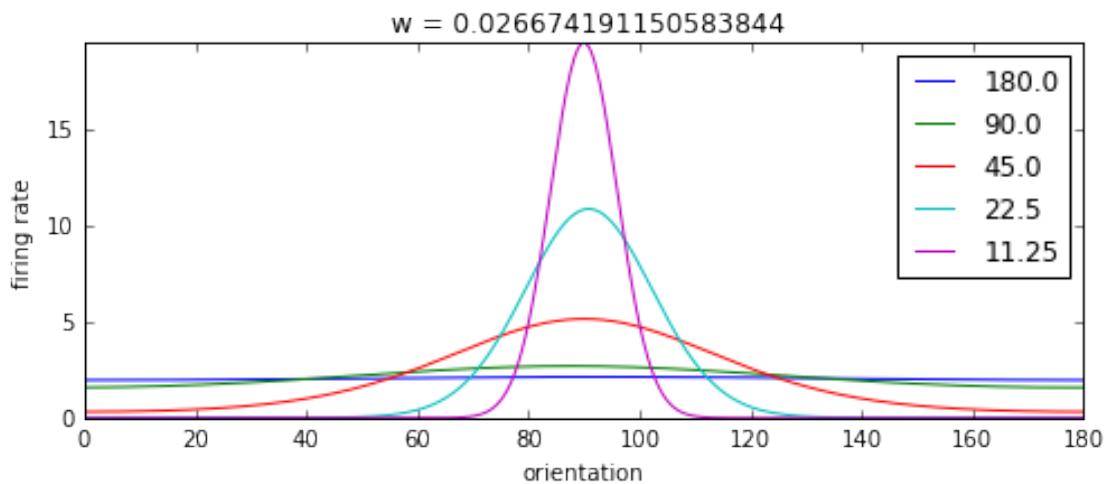
```



```

2.05374373407 0.0596066828978
2.10897219055 0.394690269482
1.98253198832 1.6802581835
1.78130343128 3.22511647092
1.63659563667 4.4612572075

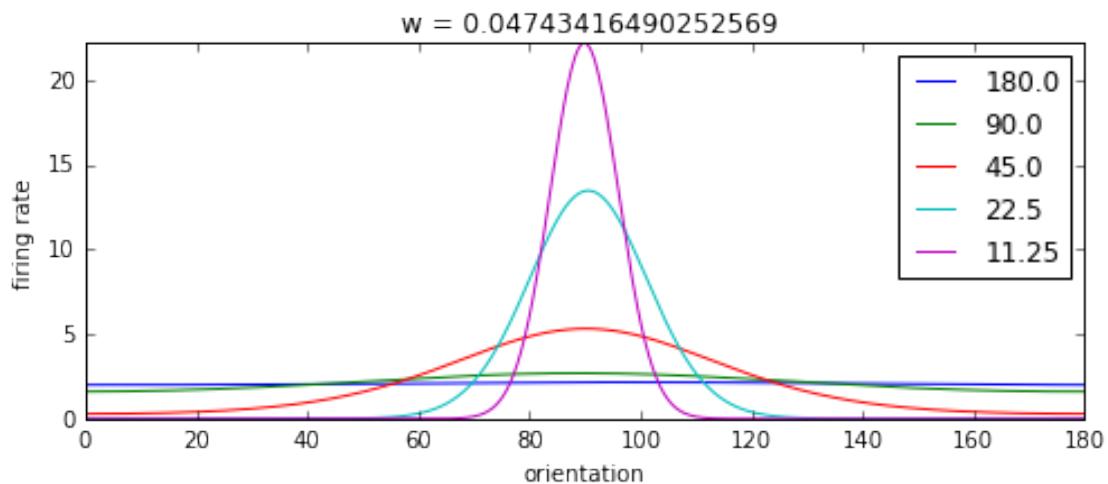
```



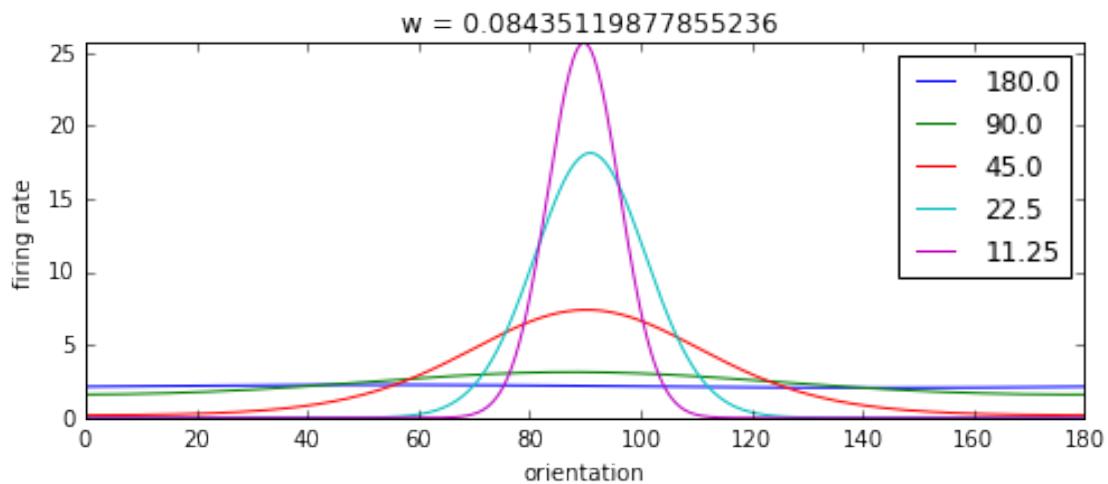
```

2.07410878776 0.0566203374558
2.11070507293 0.384707527962
1.98507433799 1.74141484412
2.02236291256 3.87150663454
1.89426888531 5.10262958768

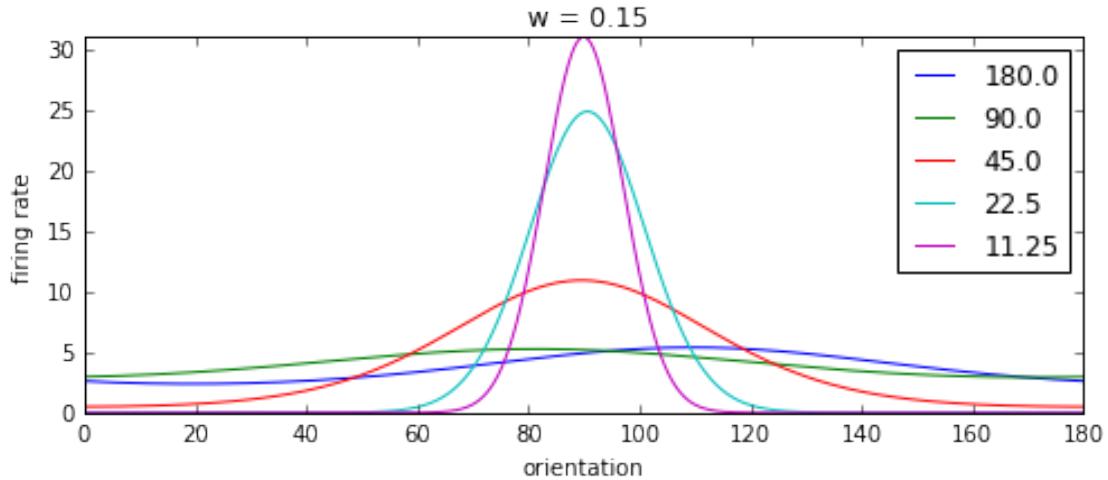
```



2.16472699771 0.071774780373
 2.30415758252 0.543972751422
 2.39523993643 2.46881644906
 2.56593710155 5.10343726916
 2.31090342809 6.03916777046



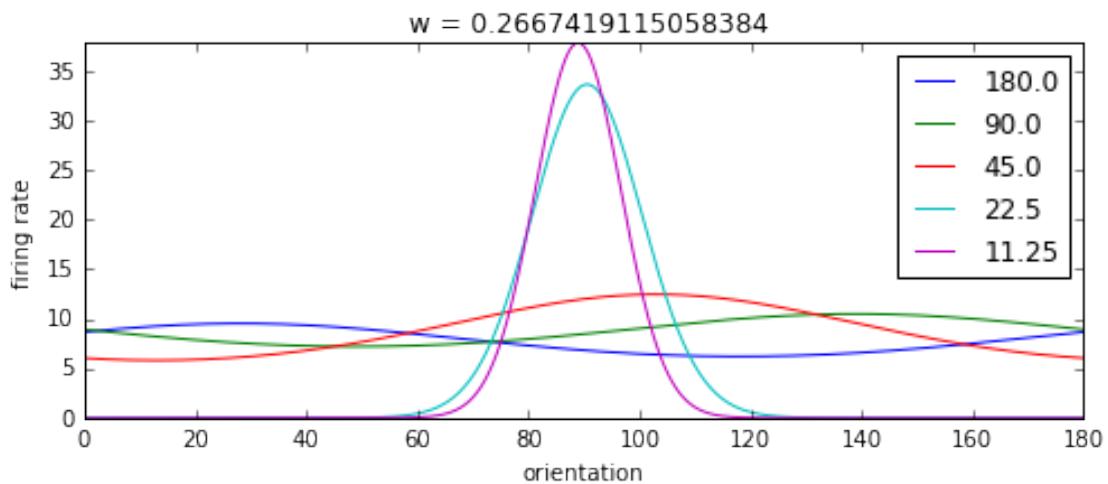
3.74433790152 1.06564866602
 4.01786858912 0.828277787621
 3.95214846504 3.61347070823
 3.65126315858 7.10320188127
 3.0901425552 7.62465545192



```

7.77228996797 1.17549953862
8.7641926298 1.16289087819
8.81012673306 2.35091446205
4.69539608689 9.4163181235
4.09261134411 9.59853115466

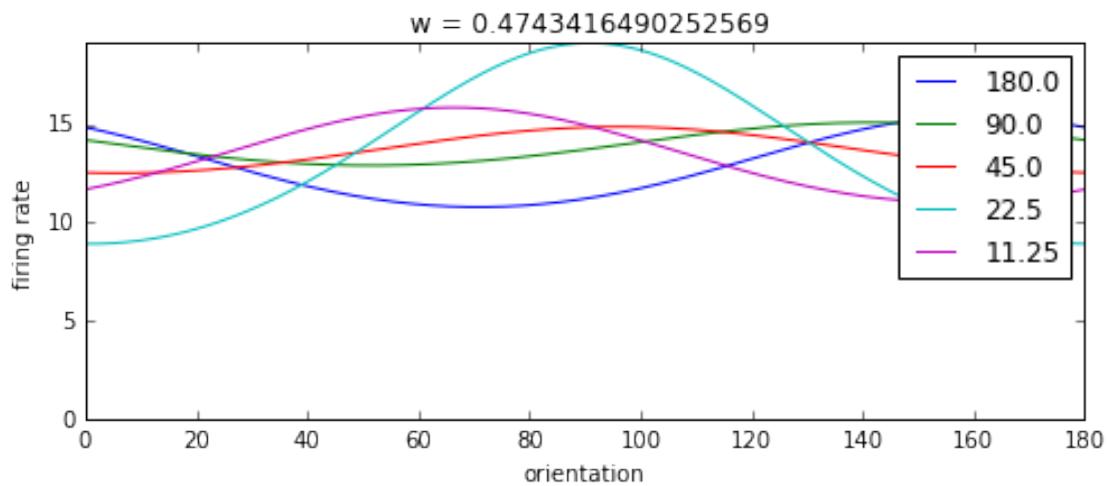
```



```

13.0019470405 1.64190531951
13.9718432794 0.772832183637
13.6617121623 0.834634646734
13.5269540731 3.61250311995
13.3216181161 1.70794664241

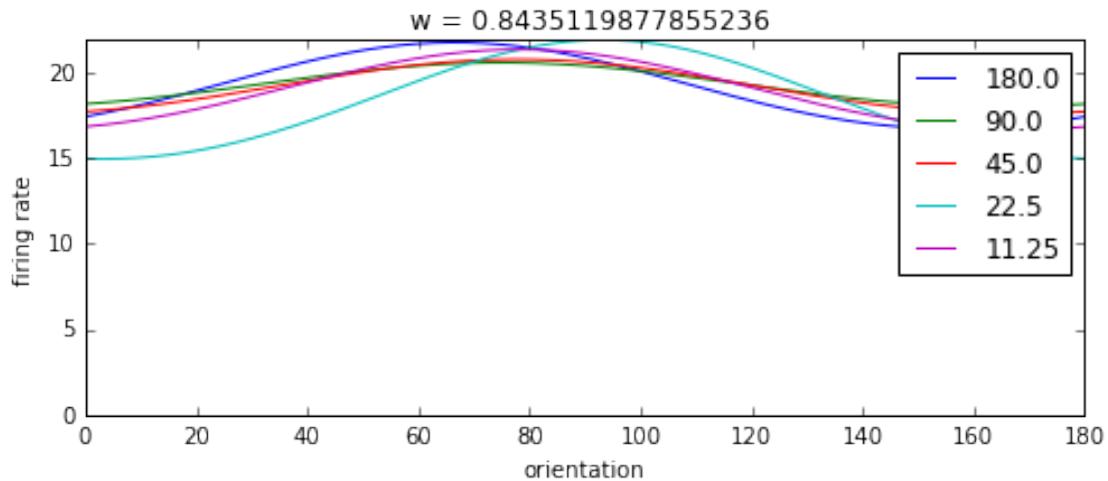
```



```

19.1654822422 1.787791675
19.2875108641 0.91285046295
19.1615710242 1.1274626117
18.2746667711 2.46182262748
18.9478834973 1.66037308076

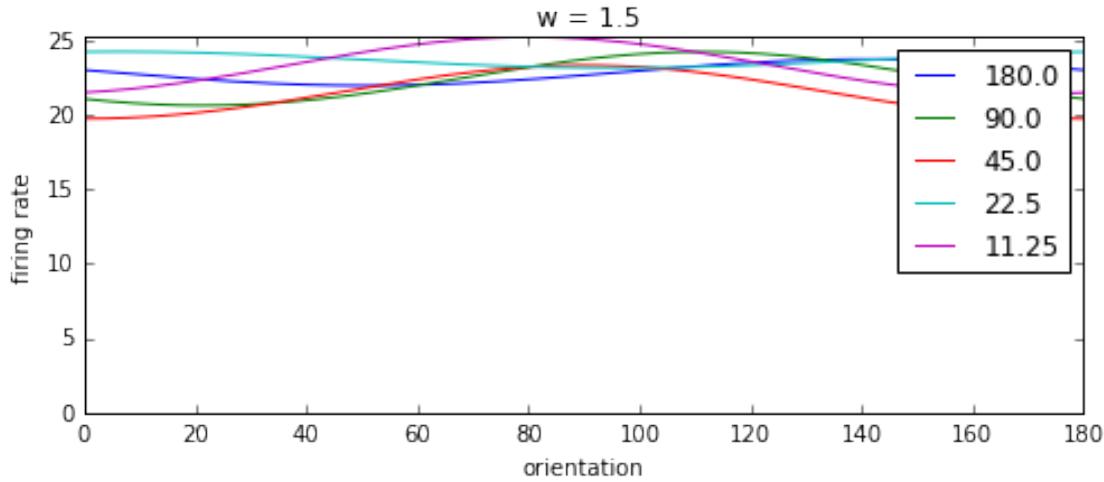
```



```

22.8315609968 0.624474627005
22.3829132672 1.27412309028
21.4855707315 1.26919600803
23.6831537966 0.386393289998
23.2563548282 1.35349634968

```



— fit en fonction du coupling

```

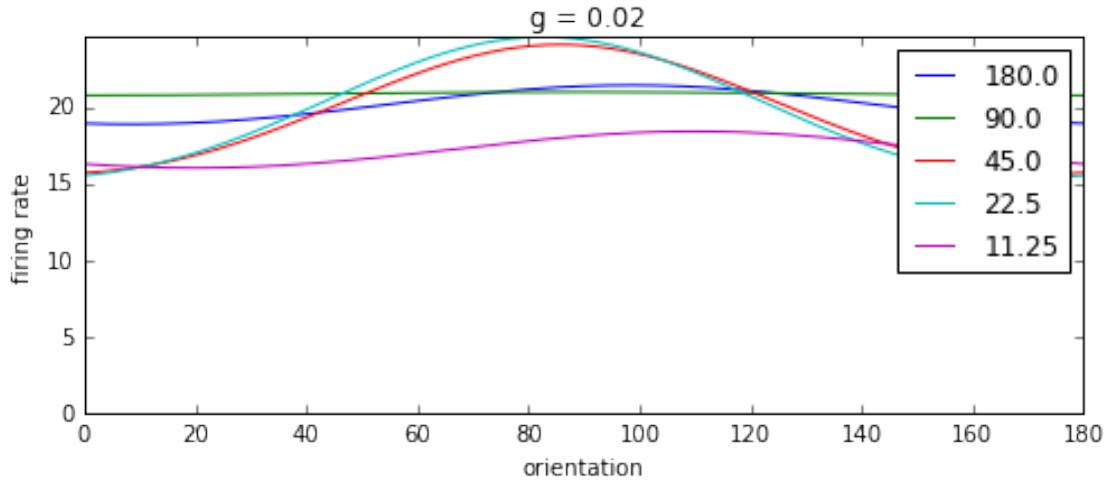
>>> net = RRNN()
... gs = net.g * np.logspace(-2, 2, 20)
... bw_values = 180*np.logspace(0, -4, 5, base=2)
... for g in gs:
...     fig, ax = plt.subplots(figsize=(8,3))
...     for bw_value in bw_values:
...         net = rec_net(time=time, g=g, b_input=bw_value)
...         df, spikesE, spikesI = net.model()
...         theta, fr, result = net.fit_vonMises(spikesE)
...         print(result.best_fit.mean(), result.best_fit.std())
...         ax.plot(theta*180/np.pi, result.best_fit, label=str(bw_value))
...
...     ax.set_title(' g = {}'.format(g))
...     ax.set_xlabel('orientation')
...     ax.set_ylabel('firing rate')
...     ax.axis('tight')
...     ax.set_xlim(0)
...     plt.legend()
...     plt.show()

```

```

20.1392524148 0.903201519378
20.8833332857 0.0768537833486
19.6528286702 2.97787938991
19.7254676372 3.24644838706
17.1976841586 0.846131489833

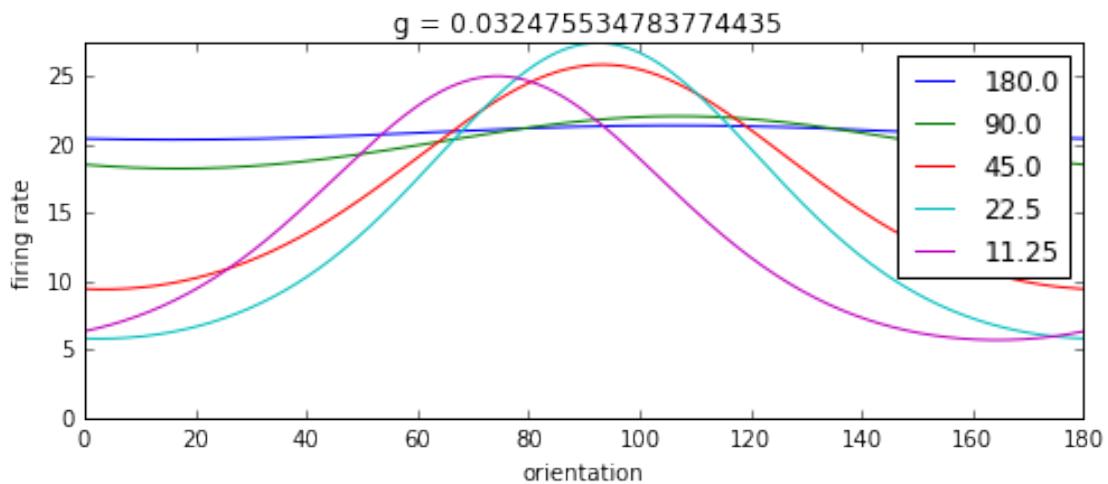
```



```

20.888906004 0.369670916498
20.126777031 1.35878182899
16.6035357315 5.80476599361
14.5791989667 7.61218223
13.6200785813 6.79533205278

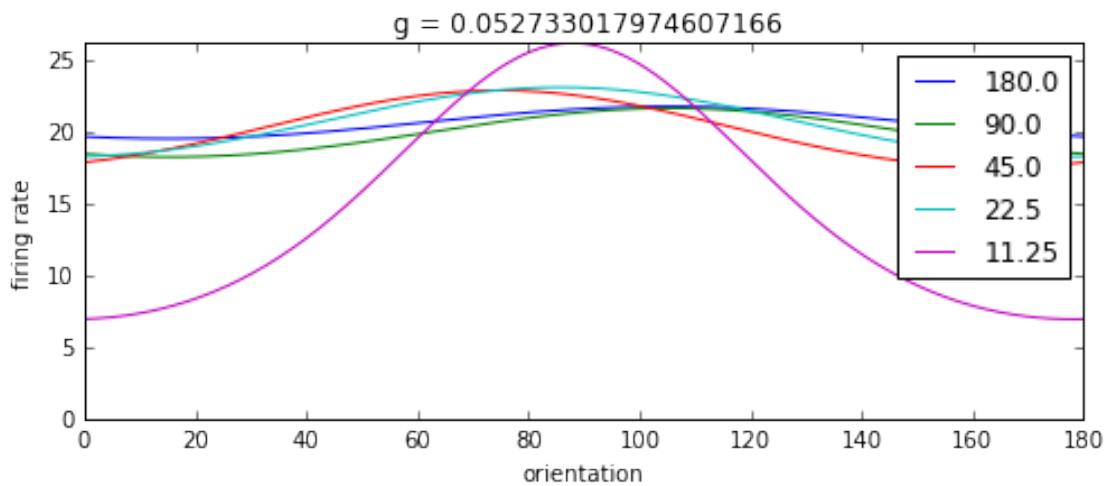
```



```

20.6428687917 0.796840868169
19.9176480677 1.20253594585
20.1369078095 1.89096847931
20.6174393394 1.71881405539
15.0026690136 6.7816892379

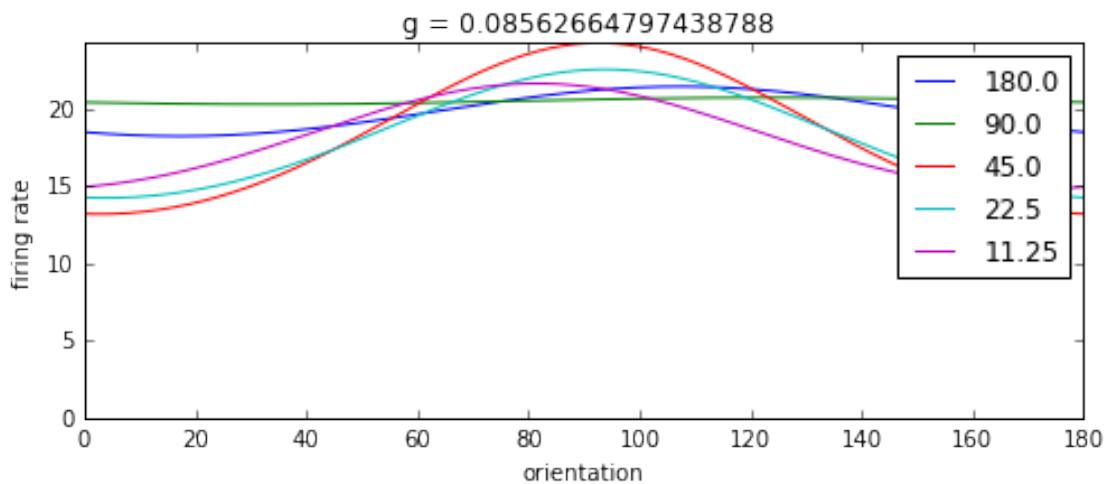
```



```

19.8171619123 1.13709274382
20.5240718813 0.154447303093
18.3087324154 3.91831845437
18.1607873594 2.94367088028
18.0765793301 2.41301116068

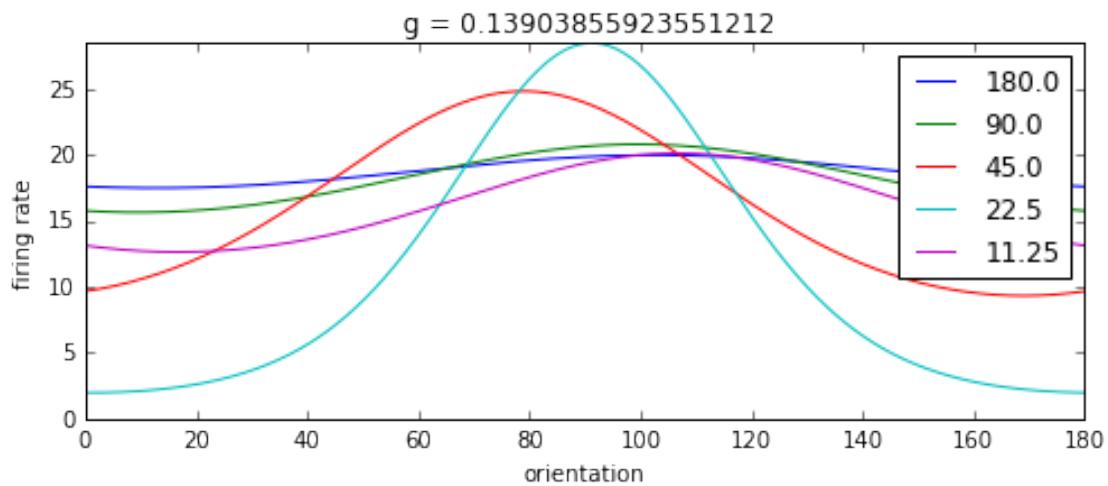
```



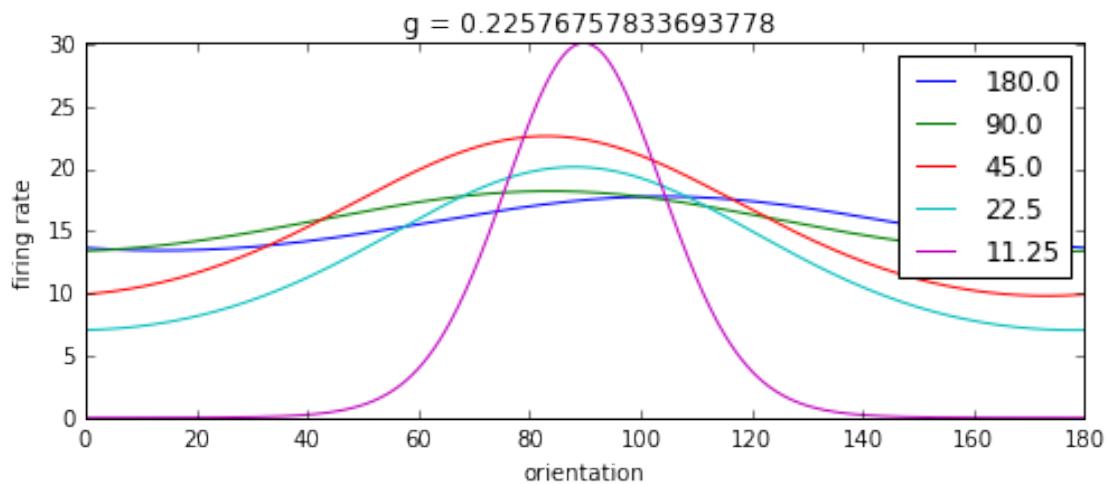
```

18.7228851851 0.882233890503
18.1364381413 1.82756465189
16.1386573439 5.47405295699
11.218276618 9.21096848304
16.1644317887 2.64785834561

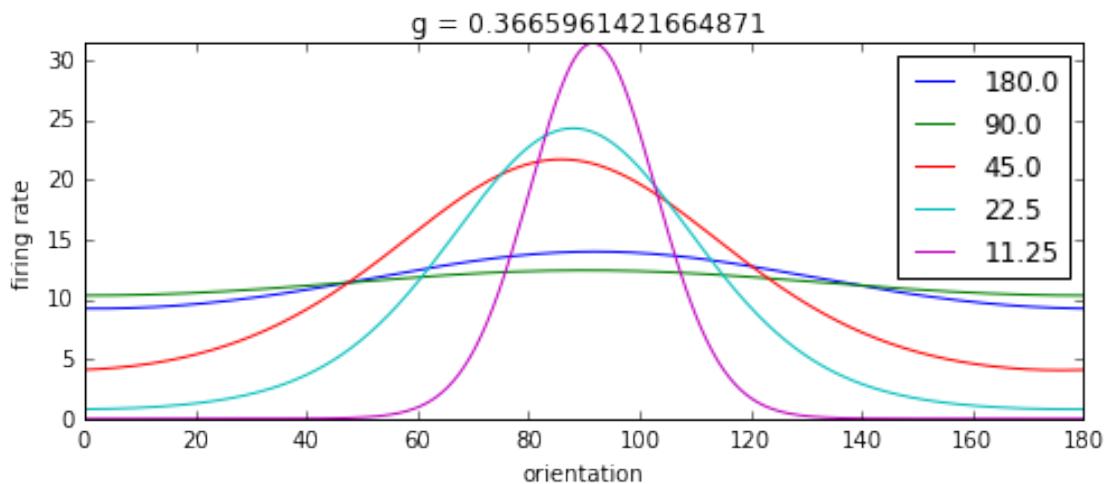
```



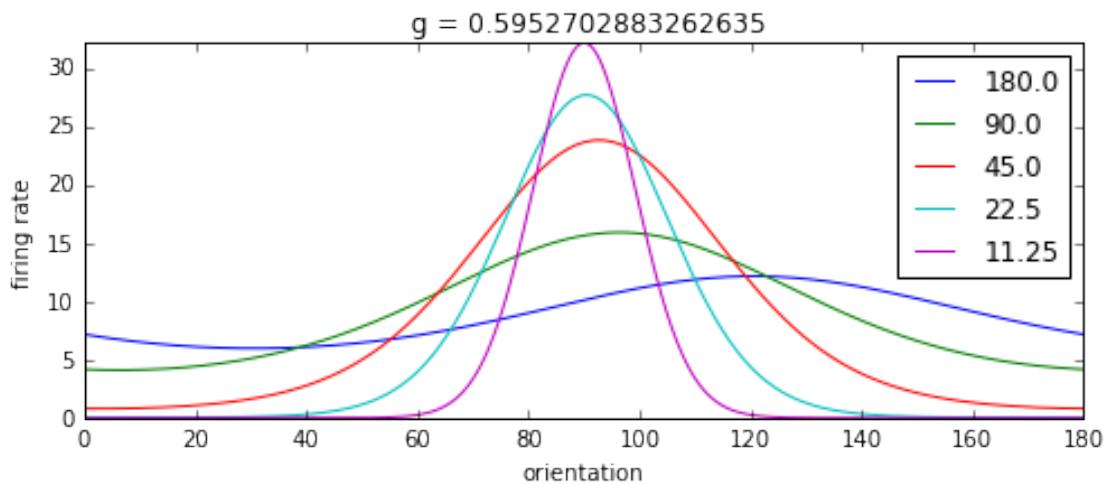
15.5271460628 1.5348127597
 15.6789887453 1.71709635933
 15.5347103274 4.52749226827
 12.7381354761 4.62126960189
 6.13428316509 9.50731366314



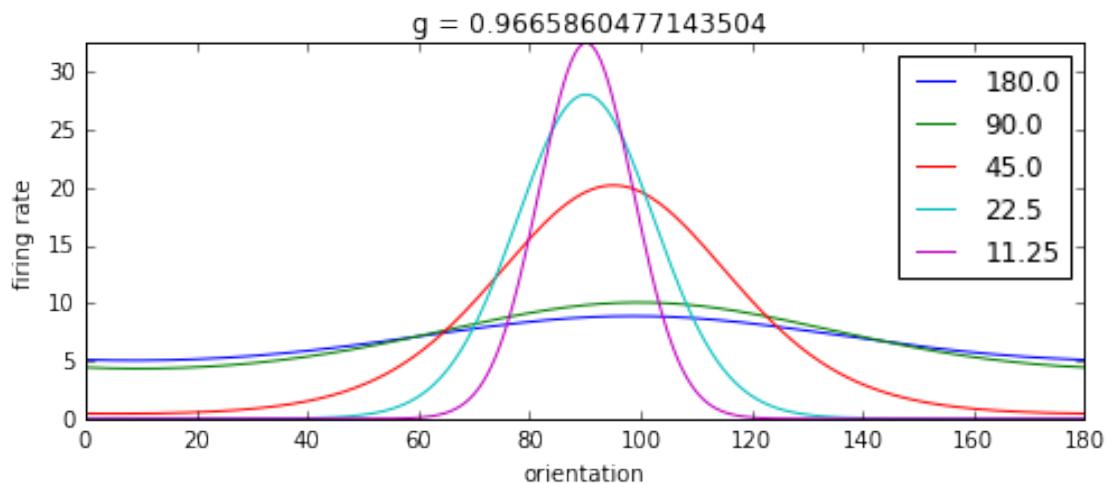
11.4737628745 1.68657003189
 11.3393750288 0.745584714635
 11.0988262513 6.19406719694
 8.23422636152 8.08196791428
 5.06295439604 9.25557656171



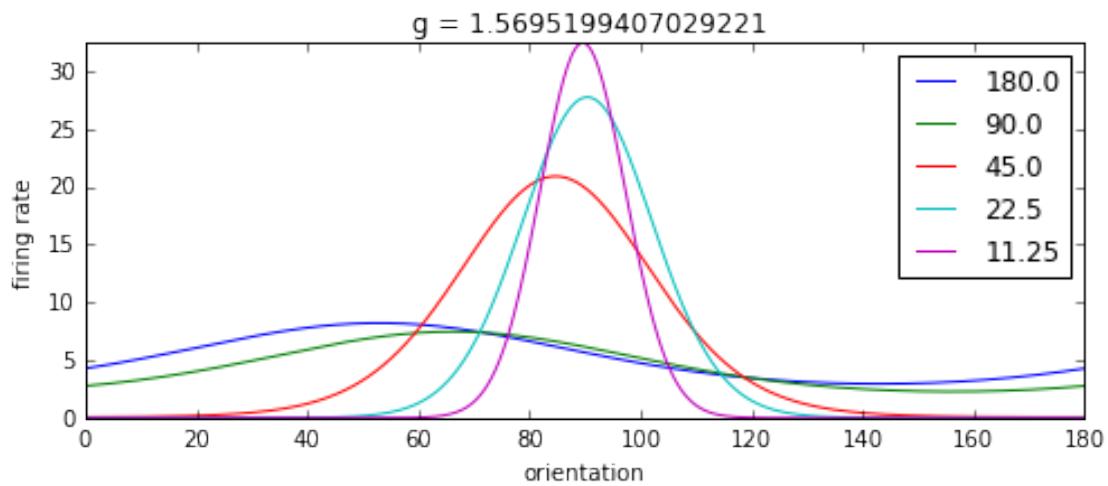
8.78112371665 2.18647331752
 9.0195334885 4.15742522186
 8.09264895747 7.9156692502
 5.89336016127 8.84237561819
 4.18081998439 8.77068010468



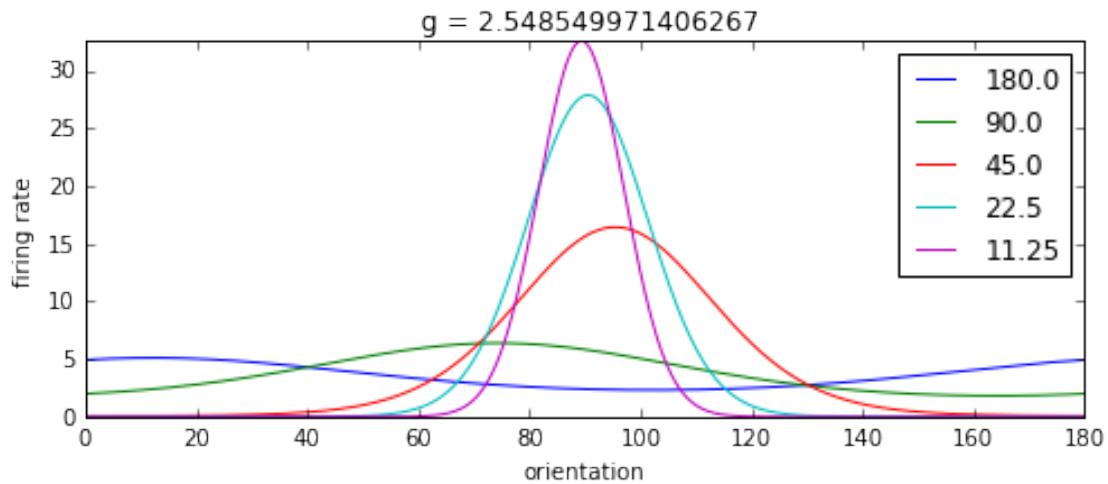
6.8262230029 1.36058241244
 6.90339689251 2.02281037916
 6.37596524353 6.75117499071
 4.96868967256 8.51816129057
 3.87504312902 8.57801860962



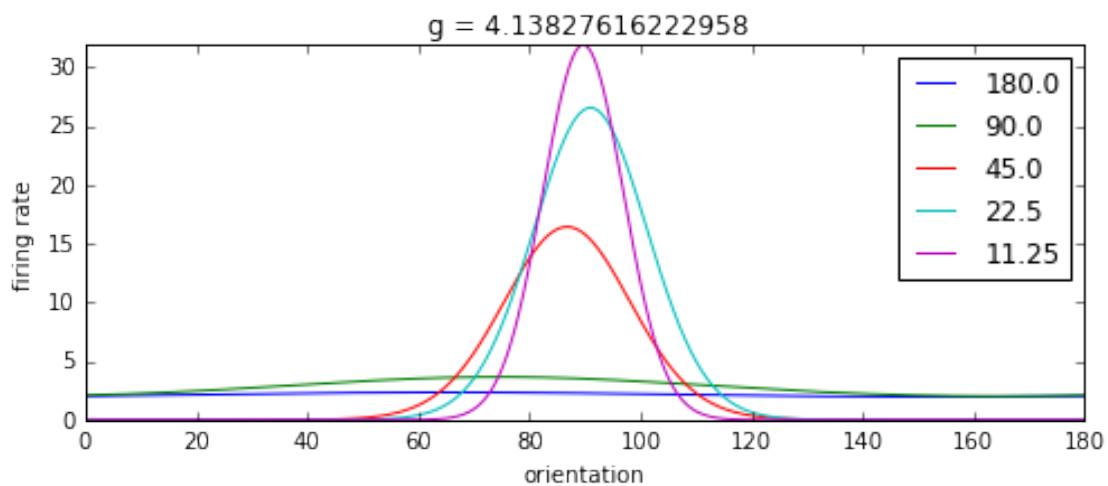
5.24459672886 1.85356566202
 4.4907036458 1.82535651753
 5.20432685627 6.88051103483
 4.61390790436 8.25425397023
 3.48657399246 8.21130079425



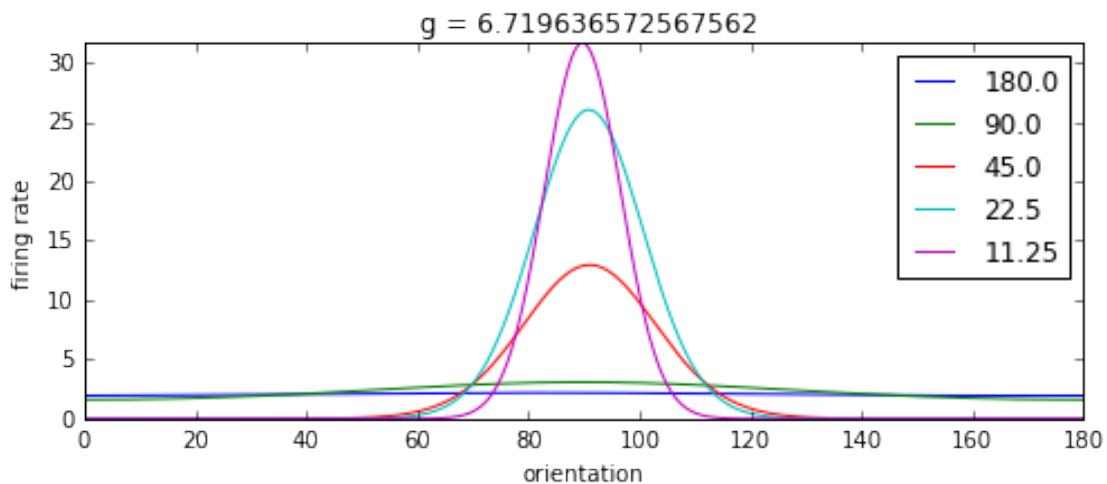
3.57482010898 0.986330860818
 3.74989717454 1.60902861566
 4.18937249217 5.43032060403
 4.27714853701 8.07360602043
 3.49301254665 8.23915534735



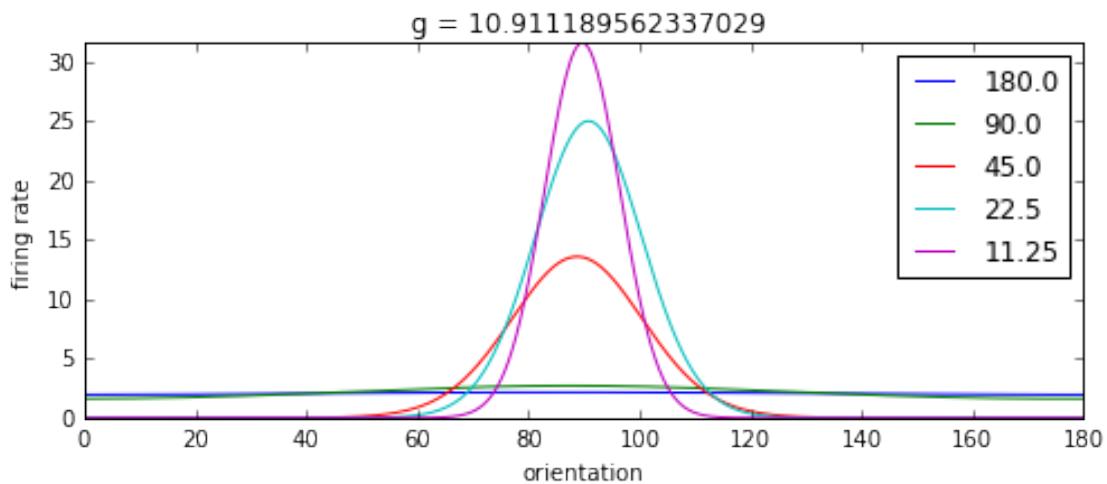
2.14431796644 0.136683501906
 2.77015589049 0.580100772004
 2.65093963461 4.83651247019
 3.83037244023 7.52058232478
 3.23875670417 7.88818131009



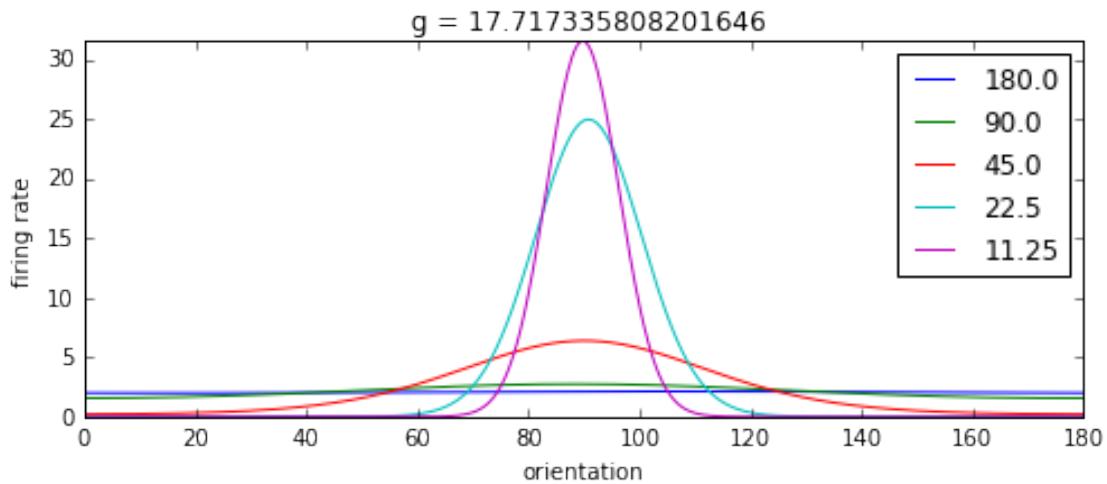
2.06671431739 0.0897620403987
 2.2871347857 0.521217863252
 2.19912459254 3.8815251782
 3.60210731815 7.26617434899
 3.05017366708 7.6638260327



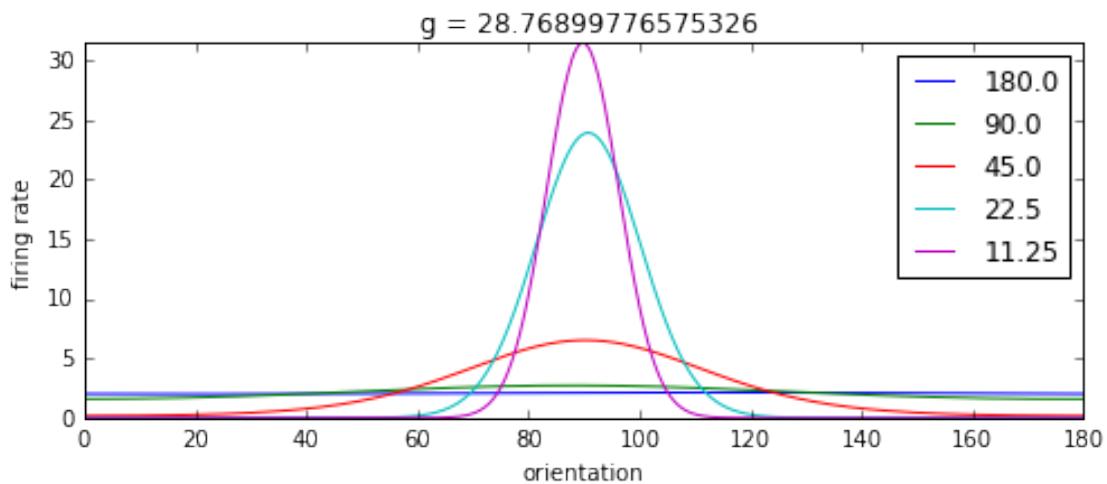
2.06116828387 0.072190052467
 2.13322108411 0.385811756815
 2.28410179346 4.05992949137
 3.39253291816 6.93462020647
 3.00249050813 7.60904395362



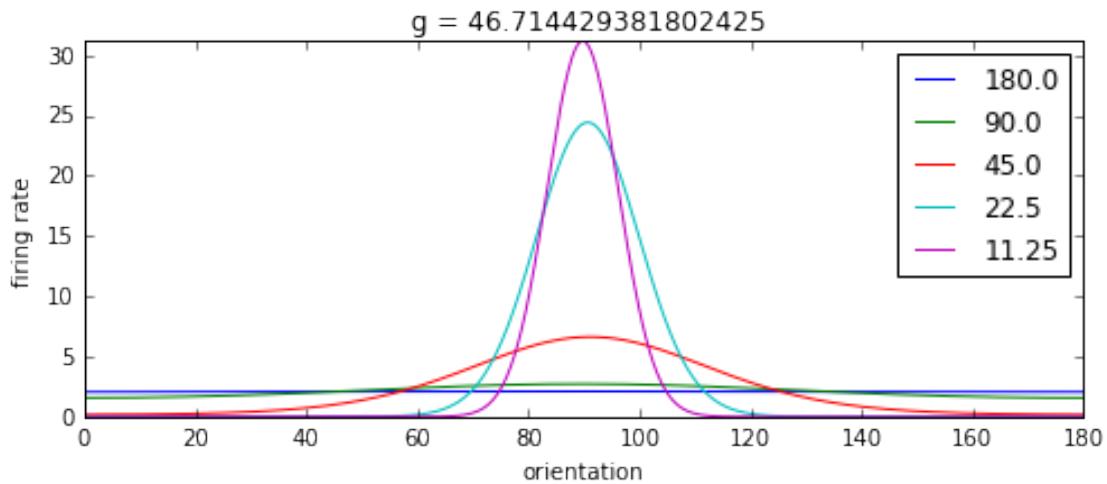
2.05928070362 0.0473847109201
 2.11098391057 0.420979149611
 2.17666708816 2.12020098975
 3.36389767442 6.89875555987
 2.87177295822 7.45547757786



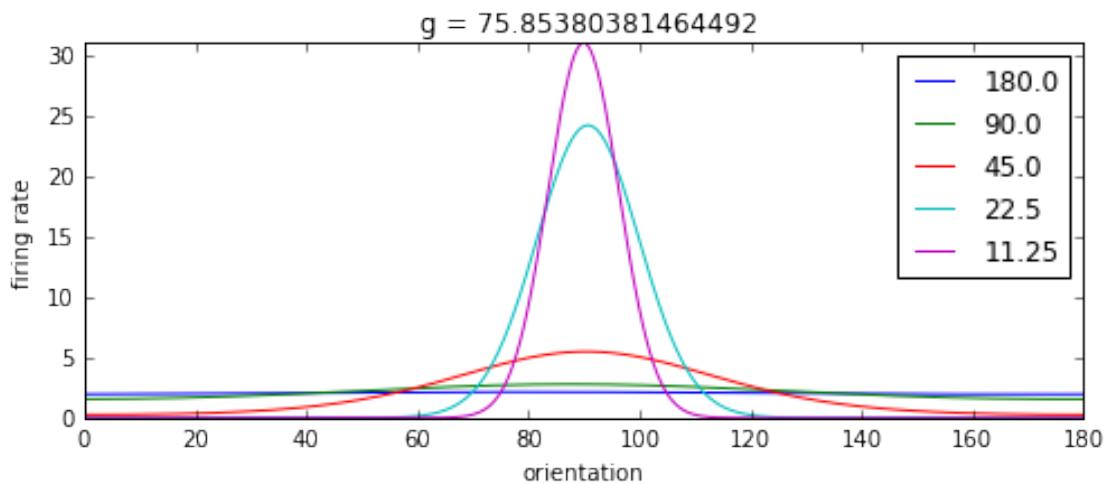
2.07038622128 0.0459911985056
 2.11817359304 0.402060573031
 2.13880103029 2.17547930949
 3.1668144625 6.56925725281
 2.86781967145 7.44189625357



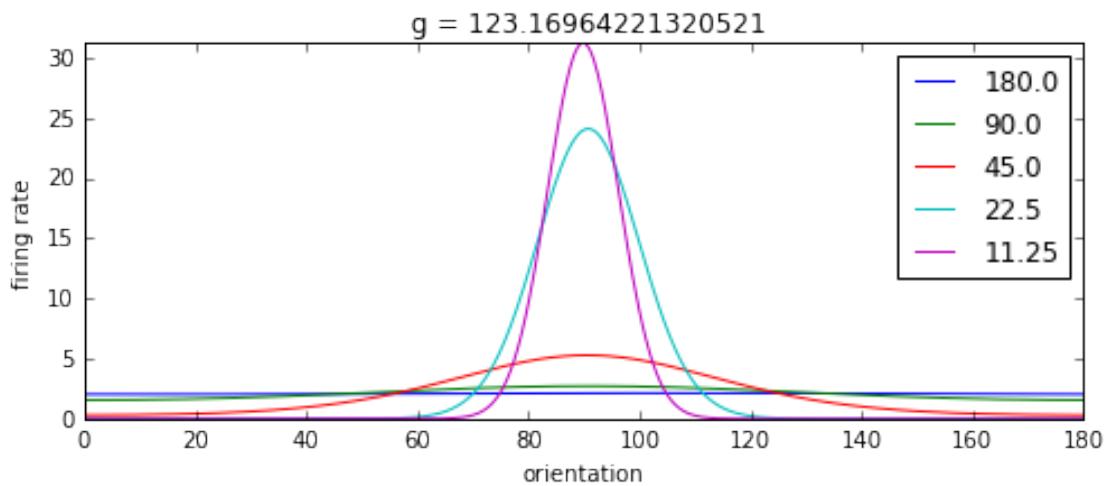
2.08148494459 0.0179682254101
 2.10906406555 0.410212083906
 2.1631636608 2.21287139573
 3.19561977209 6.67954731823
 2.79427456556 7.32369957113



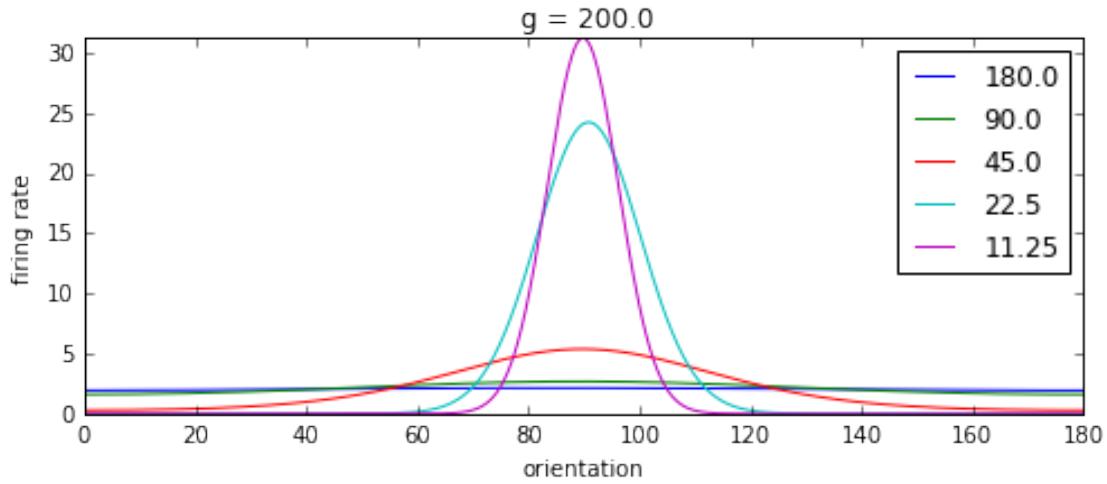
2.05743800846 0.0680524702321
 2.12924508107 0.436589493129
 2.00220425366 1.81102946714
 3.15677702345 6.62040249717
 2.76680299815 7.28047747826



2.08334934807 0.0390211148859
 2.09579562689 0.411218284666
 1.9926142849 1.72576799739
 3.11363258837 6.55865214181
 2.78957803468 7.32388433178



2.04448734192 0.0719478896313
2.11585389082 0.379604081486
2.00512279568 1.76317427747
3.18647042828 6.63901857306
2.79686162288 7.33339429981



Chapitre 4

Discussion et perspectives

4.1 Conclusion

4.1.1 Résultats obtenus

Nous avons vu qu'une modélisation de larges réseaux neuronaux résulte en fait de nombreux choix. Certains de ces choix sont faits par rapport au stimulateur et à l'interface de programmation à utiliser. Aussi, la grande diversité de modèles neuronaux existants implique que des décisions sont également à prendre quant au choix d'un de ces modèles. Ensuite, nous avons exposé le développement de l'architecture du réseau récurrent aléatoire, qui n'est rien d'autre qu'un "ring" dépourvu de topologie, ainsi que son exploration. Celle-ci nous a permis d'observer quatre états du réseau, déjà montrés par Brunel, qui sont autant de dynamiques de son activité [?]. Parmi ces quatre états, nous avons cherché à obtenir l'état balancé en manipulant le couplage interne excitation-inhibition, g . Pour obtenir cet état, nous avons procédé à une optimisation fonctionnelle du paramètre de couplage interne, sous contraintes de coefficient de variation et de gain. Nous avons alors introduit de la topologie au sein du réseau, pour implémenter un ring. Et, nous avons ensuite débuté l'étude de son comportement lorsqu'il est soumis à une entrée représentant un contenu en orientations d'un stimulus visuel.

Malheureusement, les résultats obtenus actuellement sont encore insuffisants pour effectuer toute interprétation ou prédition. Il reste encore à observer l'effet d'une entrée représentant un contenu en orientations d'un stimulus visuel sur la réponse du "ring". Nous procéderons alors à un ajustement des résultats avec les données issues des expérimentations physiologiques prévues dans le projet de recherche.

4.2 Projet de thèse

4.2.1 Motivation et contexte

La perception, dont fait partie la détection d'orientations, n'est qu'un versant des traitements qu'effectuent le système nerveux. En effet, le système nerveux permet également à un individu d'agir sur son environnement, en fonction des informations sensorielles

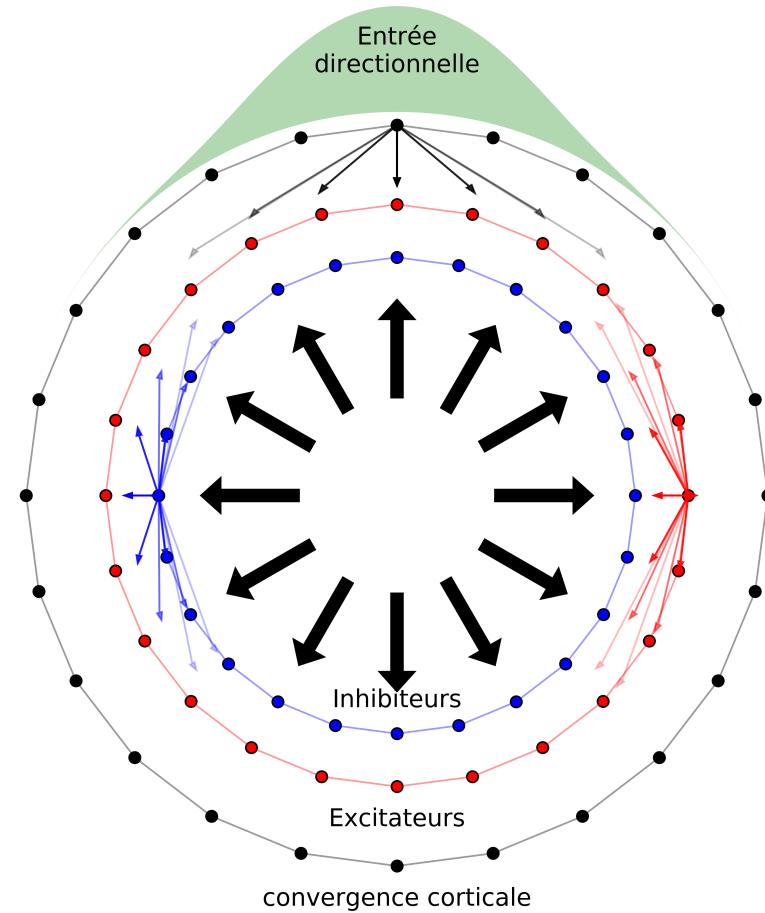
qu'il reçoit. La nature de ces informations, du moins une infime partie, a été étudiée au cours du projet de stage grâce à une approche computationnelle. Cette approche nécessitait des modèles de neurones avec un minimum de bioréalisme, afin de capturer les mécanismes neuronaux sous-tendant la perception de l'orientation. Nous pensons que ces modèles sont également adaptés à une étude des mécanismes physiologiques sous-tendant la décision motrice.

Les décisions motrices sont au cœur des opérations effectuées par le cerveau. Comme le montrent les dernières avancées en machine learning, le domaine de l'apprentissage par renforcement apporte des concepts utiles à l'étude des décisions motrices optimales. Le principe d'échantillonage aléatoire de l'espace de réalisation est central. En effet, lorsque le résultat de certaines actions est inconnu, l'échantillonage de l'espace moteur est une bonne stratégie. Cet échantillonage consiste en un choix aléatoire d'actions qui va être biaisé par l'arrivée progressive d'informations sur l'environnement. Cette approche a été initialement développée dans des environnements "quadrillés" (comme pour le jeu d'échecs ou le jeu de go) [?], cependant la généralisation de cette stratégie à des environnements plus complexes, où les associations situation-action sont continues, est maintenant chose courante dans le domaine de la robotique et du contrôle. Des schémas sensori-moteurs efficaces peuvent être appris de cette manière dans le cas où l'environnement se constitue de cibles statiques ou mobiles [?], et diverses implémentations neuromimétiques ont été proposées ces dernières années [?][?]. Néanmoins, beaucoup de problèmes restent à résoudre pour comprendre l'ensemble du flux d'actions, des premiers traitements sensoriels jusqu'à l'exécution motrice finale. Aussi, la plupart des théories ont échouées à expliquer l'adaptabilité (i.e. la généralisation d'une tâche) des systèmes sensori-moteurs biologiques. En effet, malgré des années d'études, beaucoup de modèles de décision proposés dans la littérature restent évasifs sur le substrat neuronal des mécanismes d'adaptation des décisions motrices.

Deux approches sont généralement utilisées pour étudier les processus de décision sensori-motrice en Neurosciences : la première se concentre sur les tâches de catégorisation perceptive discrète (2AFC notamment) et un Drift-Diffusion-Model (DDM) qui postule l'existence d'une accumulation d'évidences sensorielles bruitées jusqu'à qu'un seuil soit atteint. Ces modèles permettent la prédiction de distributions de précision de réponse et de temps de réaction (RT). Cependant, les DDMs sont des modèles ad hoc descriptifs et ne peuvent pas rendre compte de décisions plus complexes. La seconde approche traite traditionnellement de tâches de contrôle moteur continu. Dans ce flux continu d'informations sensorielles, le cerveau permet des comportements contrastés en fonction de différentes échelles temporelles, du simple réflexe à un apprentissage et une prise de décision élaborée. Par exemple, la vision d'un objet mobile peut donner lieu à une variété de décisions motrices telles que la préhension ou l'esquive, selon la nature et la trajectoire de l'objet.

Dans ce contexte, l'étude des mouvements oculaires permet de mettre à l'épreuve les modèles modernes de décision et d'adaptation motrice. L'orientation visuelle (i.e. le choix de l'endroit où le regard se porte), réalisée de nombreuses fois par minute chez l'homme et l'animal, est effectivement un des actes moteurs les plus élémentaires. Ce type de mouvement a été très étudié au cours des cinquante dernières années, et le but de ce projet est de mettre au défi les concepts clés de l'apprentissage par renforcement, en utilisant une

large connaissance de l'expérimentation et de la modélisation portant sur les mouvements oculaires. Récemment, des chercheurs ont modélisé ce type de tâche à l'aide d'inférences Bayésiennes . Ces dernières représentent explicitement les dynamiques des croyances de l'individu sur sa relation au monde et, ainsi, elles peuvent rendre compte des mécanismes de l'adaptation motrice, ce qui manquait dans les précédents travaux [?][?][?][?].



Réseau de neurones organisé en “ring” pour la détection d’une direction d’un regard.

4.2.2 Théorie

Notre but est de créer une base à une compréhension globale d'une simple adaptation sensorimotrice. L'approche de modélisation sera basée sur le paradigme de minimisation de l'énergie libre développée par Karl Friston, lequel s'adapte bien à l'étude des mouvements oculaires [?]. Cette approche hérite et étend de précédentes formalisations, telles que le filtre de Kalman ou le filtrage bayésien variationnel, par une formalisation de principe de la surprise dans le système étudié. Ces dernières années, les approches de la perception visuelle, par le principe d'énergie libre (FEP pour Free Energy principle),

ont donné naissance à un cadre de travail unifié pour le développement de modèles de décision sensorimotrice, parmi lesquels :

- 1) L'exploration visuelle : Le FEP considère les mouvements oculaires comme un sondage de l'environnement [?]. Ainsi, l'exploration par le biais de la saccade contribue à la construction d'une vision intégrée de l'espace et des objets environnants ;
- 2) La prédiction : idéalement, la précision des mouvements est maintenue en dépit de changements survenant dans l'environnement ou à l'intérieur de l'organisme. L'acte de vision, par exemple, équivaut à constamment extraire de nouvelles informations, venant de zones non fiables, ou variables, de la scène visuelle ;
- 3) L'apprentissage qui est le changement des relations élémentaires entre la perception et l'action, en ce qui concerne les attentes à propos d'objets de l'environnement.

Ainsi, différents aspects des mouvements oculaires sont bien expliqués par le codage prédictif, une approche du FEP. Idéalement, équiper un système (optimisé grâce au FEP) de moyens de produire une action sur son environnement, lui offre une nouvelle manière d'échantillonner son espace visuel. Ce type de stratégie est appelée le paradigme d'inférence active. Il a été prouvé que celui-ci est hautement généralisable et biologiquement plausible [?]. Cependant ce paradigme n'a pas été mis à l'épreuve dans la compréhension de l'adaptation motrice, notamment concernant le système oculomoteur.

4.2.3 Méthodes

L'originalité du projet repose sur l'étude de mouvements oculaires volontaires, en réponse à des informations visuelles, qui constitue un modèle d'apprentissage moteur utilisant le paradigme d'inférence active. Des données issues d'expériences comportementales conduites chez l'Homme seront utilisées pour tester le modèle. Ces expériences portaient principalement sur les modulations du comportement oculomoteur, comme la latence, l'amplitude ou la vitesse, observées dans un environnement changeant, en relation avec un protocole de renforcement dynamique. Ce jeu de données fournit un banc de test idéal pour sonder les décisions sensorimotrices ayant lieu sur différentes échelles temporelles, sur plusieurs niveaux de traitement et sur de nombreux mouvements (e.g. poursuites versus saccades [?][?][?]). De plus, la manipulation du délai [?], de l'attente sensorielle ou encore de la conséquence associée aux réponses motrices révèle une remarquable flexibilité des comportements oculomoteurs [?].

4.2.4 Résultats attendus

Les décisions sensorimotrices se déclinent sur différentes échelles temporelles, de dizaines de secondes à des heures, voire des jours (i.e. échelles temporelles d'adaptation et apprentissage), selon le type de prise de décision et le répertoire de mouvement. En particulier, bien que le DDM standard admet qu'une décision motrice soit prise après qu'un seuil soit franchi, aucune prédiction n'est faite à propos : 1) du contrôle du mouvement (i.e. sur le fait qu'il soit ou non modulé par des informations sensorielles) ; 2) de l'effec-teur concerné (oeil ou main) ; 3) de la valeur de mouvement (succès ou échec, bénéfique ou délétère, etc...);

Nous allons donc étudier et modéliser ces différentes variables du traitement de la décision. Ceci permet d'identifier trois axes majeurs dans le développement du projet :

- 1) Exploration de l'espace visuel : identifier les échelles temporelles caractéristiques sur lesquelles la décision motrice est faite. Les modèles biologiquement inspirés de l'activité de population [?][?] et de la plasticité [?] seront confrontés à des données comportementales existantes sur l'exploration de l'espace visuel [?]. En particulier, la stratégie d'exploration aléatoire uniforme devra être comparée à des stratégies d'exploration non aléatoire ainsi que des hypothèses "curiosity-based".
- 2) Dynamique du changement de décision et d'action. Les tâches visuo-motrices sont des bancs d'essai classiques pour étudier le choix catégoriel, mais peu d'aspects de la dynamique du changement dans les réponses motrices sont connus. Quelle est la rapidité avec laquelle les sujets modifient un schéma moteur lorsque les conditions environnementales évoluent ? Est-ce que les sujets détectent les changements ? Vont-ils continuer leur action ou l'adapter aux nouvelles conditions ? Afin de mieux prédire les séries caractéristiques du changement de la réponse, face à des situations ambiguës et/ou inconstantes, nous testerons le rôle prédictif des fluctuations aléatoires de la réponse en relation avec la dynamique d'adaptation [4]. Une importance particulière sera donnée : 2a) à l'analyse de la combinaison dynamique des mouvements de poursuites et de saccades dans la poursuite d'un objet mobile en cas d'incertitude sensorielle à propos de la position, de la direction et la vitesse de la cible ; 2b) aux dynamiques de l'adaptation saccadique dans des protocoles à double-étape.
- 3) Apprentissage et adaptation sur le long terme. Les erreurs de prédiction et les corrections en cours de mouvement sont des fonctionnalités essentielles des traitements liés à la décision. Les modèles error-based de décision en cas d'incertitude existent chez l'humain et l'animal et permettent une amélioration des temps de réaction et de la précision du mouvement.

Le cas de la capture visuelle peut, par exemple, suggérer un principe basé sur la récompense dans la foveation de cibles visuelles saillantes. Nous allons devoir comparer un simple apprentissage non supervisé d'invariances sensorimotrices, avec un apprentissage moteur basé sur la récompense, qui prescrit implicitement un choix optimal d'action à effectuer dans le contexte visuel. En étudiant simultanément différents types de mouvement oculaire (poursuite, poursuite lisse et saccade) sous contraintes de précision, nous cherchons à mettre au défi la capacité adaptative des modèles de décision visuomotrice. Un accent sera mis sur la conception et le test d'un modèle inférentiel prédisant des distributions de temps de réaction où interviennent des effets d'adaptation à long terme.

L'adaptation des décision motrices est un vaste champ de recherche et nous espérons y contribuer de façon originale. Ici, nous confronterons les avancées les plus récentes de la modélisation de l'apprentissage par renforcement, avec des observations quantitatives des opérations visuelles effectuées par les sujets humains et animaux, en utilisant des outils modernes d'observation. Nous nous attendons à ce que ce projet permette une meilleure compréhension des processus de décision chez les sujets sains et les patients atteints, par exemple, d'autisme ou de schizophrénie [?]. Cette recherche fondamentale pourrait donner lieu à des applications dans des domaines tels que la robotique ou la

recherche clinique.

Bibliographie

- [1] A. N. Burkitt. A review of the integrate-and-fire neuron model : I. homogeneous synaptic input. *Biol Cybern*, 95(1) :1–19, Apr 2006.
- [2] A. N. Burkitt. A review of the integrate-and-fire neuron model : II. inhomogeneous synaptic input and network properties. *Biol Cybern*, 95(2) :97–112, Jul 2006.
- [3] David Hansel, Germán Mato, and Claude Meunier. Synchrony in excitatory neural networks. *Neural computation*, 7(2) :307–337, 1995.
- [4] Paula Sanz-Leon, I. Vanzetta, G. S. Masson, and L. U. Perrinet. Motion clouds : model-based stimulus synthesis of natural-like random textures for the study of motion perception. *Journal of Neurophysiology*, 107(11) :3217–3226, March 2012.