UNIVERSITY OF CORDOBA
INSTITUTE OF POSTGRADUATE STUDIES

UNIVERSITY MASTER'S DEGREE

PHYSICAL TECHNOLOGY: RESEARCH AND APPLICATIONS

MASTER'S THESIS

Exploring the relationship between Hawkes processes and self-organized criticality in living systems

Antonio Rivas Blanco

Tutor(s): Jorge Hidalgo Aguilera and Serena di Santo

Line of research: Modelling of complex systems and their interdisciplinary applications

Córdoba, 07/2024

Autorización de defensa del Trabajo Fin de Máster

Tutor 1: Jorge Hidalgo Aguilera
Tutor 2: Serena di Santo

INFORMAN: Que el presente trabajo titulado *Exploring the relationship between Hawkes processes and self-organized criticality in living systems* que constituye la memoria del Trabajo Fin de Máster ha sido realizado por Antonio Rivas Blanco y AUTORIZA/N su presentación para que pueda ser defendido en la convocatoria 1ª con fecha 23/07/2024.

Firmado en Córdoba a X de julio de 2024

Tutor 1: Jorge Hidalgo Aguilera | Tutor 2: Serena di Santo

Declaración de autoría

Nombre y apellidos: Antonio Rivas Blanco
DNI: 49832223D

DECLARA

1. Que el trabajo que presenta es totalmente original y que se hace responsable de todo su contenido.
2. Que no ha sido publicado no total ni parcialmente.
3. Que todos los aportes de otros autores/as han sido debidamente referenciados.
4. Que no ha incurrido en fraude científico, plagio o vicios de autoría.
5. Que, en caso de no cumplir los requerimientos anteriores, aceptará las medidas disciplinarias sancionadoras que correspondan.

Firmado en Córdoba a, X de julio de 2024

Cambiar encabezado y pie de página en .sty

Contents

List of Figures

List of Tables

Abstract

Chapter 1

Introduction

Chapter 2

Objectives

Chapter 3

Methodology

Chapter 4

Results

This section provides the main results of the investigation. First, the results reproduced from the original paper [1] are presented. Then, the results of the analysis with $n = 2$ are shown. Finally, we have studied the behaviour of an inhibitory and excitatory neuron coupled.

4.1 Results from the original paper

The first result is the percolation phase diagram is shown in Figure 4.1. It displays the percolation strength P_∞ versus the resolution parameter Δ.

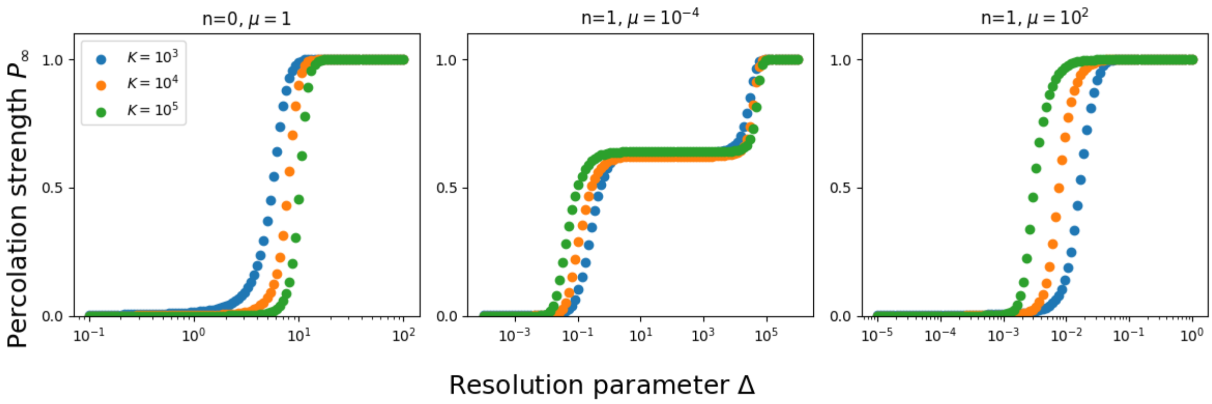

Figure 4.1. Percolation phase diagrams for different event number K taking average values of $R = 1000$ realizations.

The first plot configuration is a Markovian ($n = 0$) Poisson process with rate μ. This is the simplest case, where the inter-event time $x = t_i - t_{i-1}$ follows an exponential distribution $P(x_i) = \mu e^{\mu x_i}$. The other two plots are non-Markovian Hawkes processes for $\mu \ll 1$ and $\mu \gg 1$. In one hand, a double transition is observed when $\mu = 10^{-4}$, in the other hand, a single transition occurs when $\mu = 10^2$.

Once we have the phase diagram, we can study avalanche statistics. Given a resolution parameter Δ, we can spot clusters or avalanches of activity. A cluster starts when a neuron fires and ends if the neuron does not fire for a time greater than Δ. We define the size of a cluster as the number of spikes it contains and the duration as the time between the first and last spike. We have studied the avalanches for $K = 10^5$ events and $R = 1000$ realizations to obtain the average values since the process is highly not stationary. We will study the size and duration of the avalanches for the three different regions of the phase diagram for $\mu = 10^{-4}$ and the two regions of the phase diagram for $\mu = 10^2$. These regions are separated by two thresholds, a pseudocritical threshold Δ_1^* and the threshold of the second transition at Δ_2^*. We can compute these with the following formulas [1]:

$$\Delta_1^* \simeq \frac{log(K)}{\langle\lambda\rangle} = \frac{log(K)}{\mu + \sqrt{2\mu K}} \tag{4.1}$$

$$\Delta_2^* = \frac{log(K)}{\mu} \tag{4.2}$$

4.2 Results for n=2

In the article, the authors have studied a process which is critical itself because the parameter n is fixed to $n = 1$. We have studied the case $n = 2$ to see if the process is still critical. In the Figure 4.2 two time series for $n = 1$ and $n = 2$ are shown.

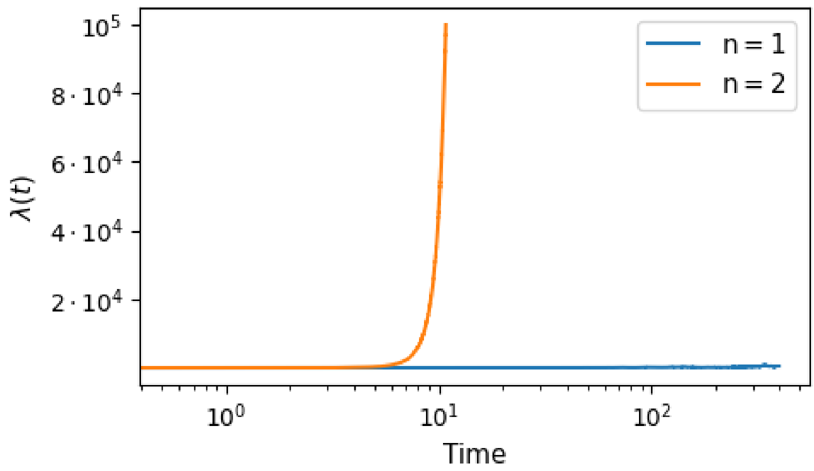

Figure 4.2. Time series for $n = 1$ and $n = 2$.

Similarly to the previous section, first we obtain the phase diagram in order to observe the transitions. In this case, Eqs 4.1-4.2 are not valid. Therefore, we will obtain this parameter graphically from the phase diagrams shown in Figure 4.3.

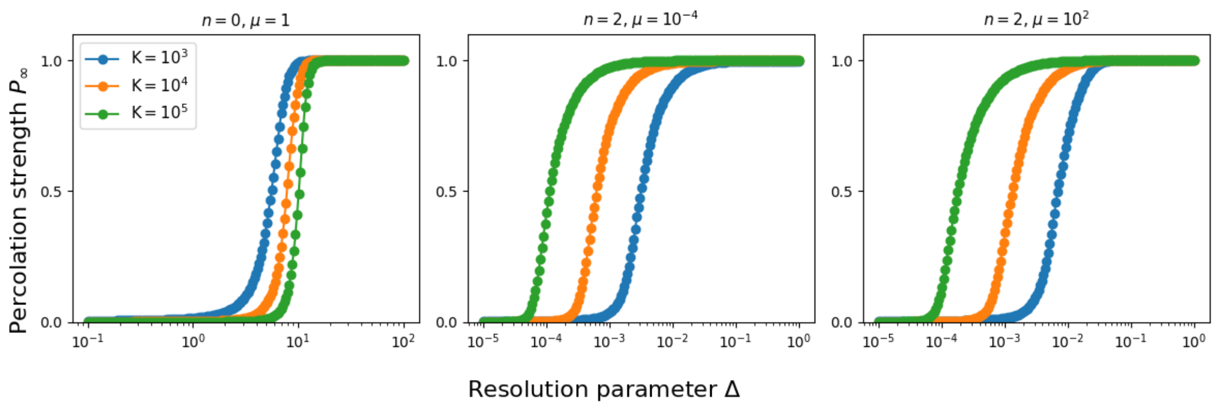

Figure 4.3. Percolation phase diagrams for a Hawkes process with $n = 2$.

As we can see, now we have a single transition for $\mu = 10^{-4}$ and $\mu = 10^{2}$ corresponding to 1D percolation, consequently, the exponents for the size and duration should be $\alpha = \tau = 2$. In a similar way, we have studied the avalanches for $K = 10^5$ events and $R = 1000$ realizations to obtain the average values. The statistics of the avalanches are shown in Figure 4.4.

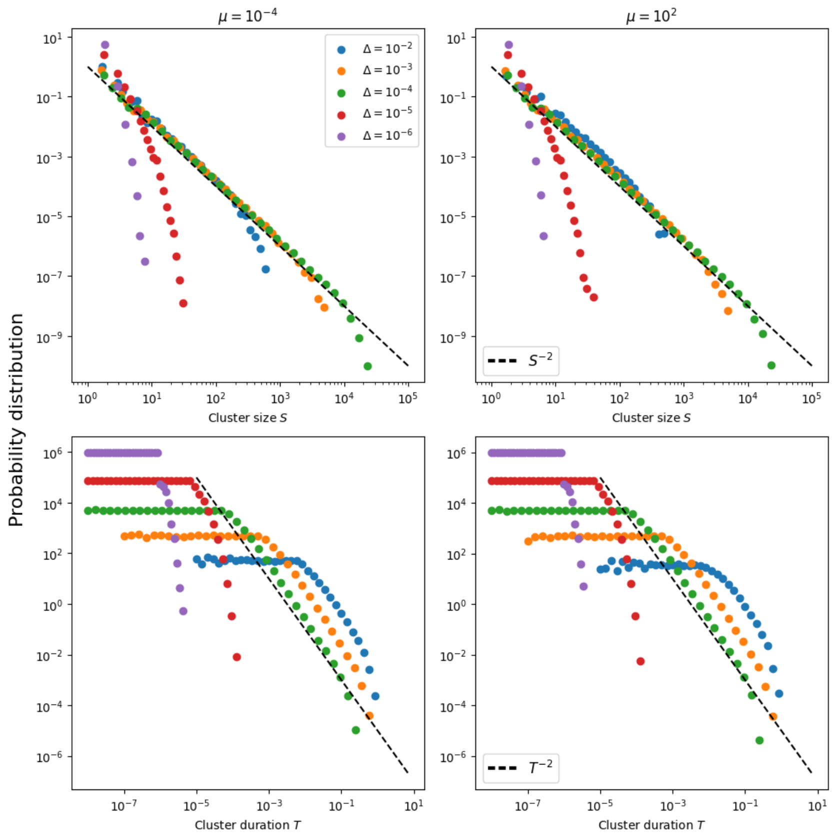

Figure 4.4. Avalanches' statistics for a Hawkes process with $n = 2$.

Chapter 5

Conclusions

Bibliography

[1] Daniele Notarmuzi et al. “Percolation theory of self-exciting temporal processes”. In: *Physical Review E* 103.2 (2021), p. L020302.

Anexo

REVISAR LOS CÓDIGOS PARA QUE ESTÉN ACTUALIZADOS

Script 5.1. *Script* Python con todas las funciones.

```
import numpy as np
import matplotlib.pyplot as plt

def algorithm(rate, mu, n):
    """
    Algorithm that computes interevent times and Hawkes intensity

    #Output: rate x_k, x_k
    """
    # Paso 1
    u1 = np.random.uniform()
    if mu == 0:
        F1 = np.inf
    else:
        F1 = -np.log(u1) / mu

    # Paso 2
    u2 = np.random.uniform()
    if (rate - mu) == 0:
        G2 = 0
    else:
        G2 = 1 + np.log(u2) / (rate - mu)


    # Paso 3
    if G2 <= 0:
        F2 = np.inf
    else:
        F2 = -np.log(G2)

    # Paso 4
    xk = min(F1, F2)

    # Paso 5
    rate_tk = (rate - mu) * np.exp(-xk) + n + mu
    return rate_tk, xk

def generate_series(K, n, mu):
    """
    Generates temporal series for K Hawkes processes

    ##Inputs:
```

```
    K: Number of events
    n: Strength of the Hawkes process
    mu: Background intensity

    ##Output:
    times: time series the events
    rate: time series for the intensity
    """
    times_between_events = [0]
    rate = [mu]
    for _ in range(K):
        rate_tk, xk = algorithm(rate[-1], mu, n)
        rate.append(rate_tk)
        times_between_events.append(xk)
    times = np.cumsum(times_between_events)
    return times, rate

def identify_clusters(times, delta):
    """
    Identifies clusters in a temporal series given a resolution parameter delta

    ## Inputs:
    times: temporal series
    delta: resolution parameter

    ## Output:
    clusters: list of clusters
    """
    clusters = []
    current_cluster = []
    for i in range(len(times) - 1):
        if times[i + 1] - times[i] <= delta:
            if not current_cluster:
                current_cluster.append(times[i])
            current_cluster.append(times[i + 1])
        else:
            if current_cluster:
                clusters.append(current_cluster)
                current_cluster = []
    return clusters

def generate_series_perc(K, n, mu):
    """
    Generates temporal series for K Hawkes processes

    ##Inputs:
    K: Number of events
    n: Strength of the Hawkes process
    mu: Background intensity

    ##Output:
    times_between_events: time series the interevent times
    times: time series the events
    rate: time series for the intensity
    """
    times_between_events = [0]
    rate = [mu]
    for _ in range(K):
        rate_tk, xk = algorithm(rate[-1], mu, n)
        rate.append(rate_tk)
        times_between_events.append(xk)
    times = np.cumsum(times_between_events)
```

```python
    return times_between_events, times, rate

def calculate_percolation_strength(times_between_events, deltas):
    percolation_strengths = []

    for delta in deltas:
        cluster_sizes = []
        # Initialize the size of the current cluster
        current_cluster_size = 1 # The first event is always a cluster

        for i in range(len(times_between_events)):
            if times_between_events[i] <= delta:
                current_cluster_size += 1
            else:
                if current_cluster_size > 1: # Only consider clusters with more than one event
                    cluster_sizes.append(current_cluster_size)
                # Reset the size of the current cluster
                current_cluster_size = 1 # The next event is always a cluster

        # Add the size of the last cluster
        if current_cluster_size > 1: # Only consider clusters with more than one event
            cluster_sizes.append(current_cluster_size)

        max_cluster_size = max(cluster_sizes)

        percolation_strengths.append(max_cluster_size / len(times_between_events))
    return percolation_strengths

def model(n_max, mu_E, mu_I, tau, n_EE, n_IE, n_EI, n_II, dt):
    """
    Solve the equations of the mena field model for a given number of iterations n_max

    Inputs:
    n_max: number of iterations
    mu_E: Poisson rate of excitatory neurons
    mu_I: Poisson rate of inhibitory neurons
    tau: characteristic time of the system
    n_EE: influence of excitatory neurons on excitatory neurons
    n_IE: influence of excitatory neurons on inhibitory neurons
    n_EI: influence of inhibitory neurons on excitatory neurons
    n_II: influence of inhibitory neurons on inhibitory neurons
    dt: time step

    Outputs:
    time: time series
    t_events_E: times of events of excitatory neurons
    t_events_I: times of events of inhibitory neurons
    rates_E: rates of excitatory neurons
    rates_I: rates of inhibitory neurons
    """
    n_E = n_I = n = 0
    t_events_E = [0]
    t_events_I = [0]
    rates_E = [mu_E]
    rates_I = [mu_I]
    time = [0]
    while n <= n_max:
        # Excitation neurons
        l_Enew = rates_E[-1]  + dt * (mu_E- rates_E[-1])/tau
        if np.random.uniform() < rates_E[-1]*dt:
            l_Enew += n_EE
```

```
            t_events_E.append(time[-1]+dt*np.random.uniform())
            n_E += 1
        if np.random.uniform() < rates_I[-1]*dt:
            l_Enew -= n_IE
            t_events_E.append(time[-1]+dt*np.random.uniform())
            n_E += 1

        # Inhibition neurons
        l_Inew = rates_I[-1] + dt * (mu_I- rates_I[-1])/tau
        if np.random.uniform() < rates_E[-1]*dt:
            l_Inew += n_EI
            t_events_I.append(time[-1]+dt*np.random.uniform())
            n_I += 1
        if np.random.uniform() < rates_I[-1]*dt:
            l_Inew -= n_II
            t_events_I.append(time[-1]+dt*np.random.uniform())
            n_I += 1
        rates_E.append(l_Enew)
        rates_I.append(l_Inew)
        time.append(time[-1]+dt)

        n = n_E + n_I
    return time, t_events_E, t_events_I, rates_E, rates_I
```