Estimation of a Linear-Nonlinear-Poisson neural encoding model for motor Brain-Machine Interfaces

DATA SCIENCE AND ENGINEERING

and

PHYSICS ENGINEERING

by

Mireia Cavallé Salvadó

under the supervision of

Prof. Yiwen Wang

tutored by

Xavier Giró Nieto

香港科技大學
THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de Formació Interdisciplinària Superior

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

telecos
BCN

Hong Kong, February 2022

ABSTRACT

In recent years, the demand of prosthetic devices that substitute dysfunctional limbs in human beings has been substantially increasing.
In order to control them with brain's neural activity, brain-machine interfaces related studies have developed a decoding model capable of inferring movement given the associated motor cortical neurons spike activity. However, to meet this model's full potential, one needs to provide an instantaneous neural encoding model that displays the direct relationship between the instantaneous kinematics (position, velocity and bias) and the neural spike activity.
The proposed encoder is a Linear-Nonlinear-Poisson model that predicts the firing probability of one neuron and feeds it to a Poisson spike generator. We test the encoder on the real neural activity collected from motor cortex of rats performing a lever pressing task. The aim of this research is to provide a procedure to estimate neural ending parameters and check the encoding suitability to predict neural activity.

En los últimos años, la demanda de prótesis que sustituyan extremidades disfuncionales de los seres humanos ha aumentado considerablemente.
Para controlarlas con la actividad neuronal del cerebro, algunos estudios sobre interfaces cerebro-computadora han desarrollado un modelo de descodificación capaz de inferir el movimiento de estas extremidades dada la actividad eléctrica de las neuronas motoras asociadas en el córtex cerebral. Sin embargo, para aprovechar todo el potencial de este modelo, es necesario proporcionar un modelo de codificación neuronal instantánea que exprese la relación directa entre la cinemática (posición, velocidad y sesgo) y la actividad neuronal.
El codificador propuesto es un modelo Lineal-NoLineal-Poisson que predice la probabilidad de activación de una neurona y se proporciona a un generador de pulsos de Poisson. Se prueba el codificador con datos reales obtenidos del córtex motor de ratas realizando una tarea de presión de palanca. El objetivo de esta investigación es proporcionar un procedimiento para estimar los parámetros neuronales finales y comprobar la adecuación de la codificación para predecir la actividad neuronal.

En els últims anys, la demanda de pròtesis que substitueixin extremitats no funcionals en els éssers humans ha augmentat considerablement.
Per controlar-les amb l'activitat neuronal del cervell, alguns estudis sobre interfícies cervell-computadora han desenvolupat un mode de descodificació capaç d'inferir el moviment d'aquestes extremitats donada l'activitat elèctrica de les neurones motores associades en el còrtex cerebral. Tot i això, per aprofitar tot el potencial d'aquest model, és necessari proporcionar un model de codificació neuronal instantani que expressi la relació directe entre la cinemàtica (posició, velocitat i biaix) i l'activitat neuronal.
El codificador proposat és un model Lineal-NoLineal-Poisson que prediu la probabilitat d'activació d'una neurona i se li proporciona a un generador de polsos de Poisson. Es prova el codificador amb dades reals obtingudes del còrtex motor de rates realitzant una tasca de pressió de palanca. L'objectiu d'aquesta investigació és proporcionar un procediment per estimar els paràmetres neuronals finals i comprovar l'adequació de la codificació per predir l'activitat neuronal.

KEYWORDS

Brain-machine interface, Computational neuroscience, Neuroprosthetic, Neuron, Movement intention, LNP Encoder

Interfaz cerebro-computadora, Neurociencia computacional, Neuroprótesis, Neurona, Intención del movimiento, Codificador LNP.

Interfície cervell-computadora, Neurociència computacional, Neuropròtesi, Neurona, Intenció del moviment, Codificador LNP.

ACKNOWLEDGEMENTS

I would like to express my thanks to all the people that supported me during the development of this thesis.

First and foremost to Prof. Yiwen Wang for guiding me in the whole process and instilling in me her passion for the topic.

Thanks to *CFIS*, *Fundació Cellex* and *Hong Kong University of Science and Technology (HKUST)* for giving me the opportunity to write this thesis with such a great research team in such an amazing location.

Also, thanks to Xavier Giró Nieto for tutoring me from Spain.

And finally, I would like to thank my family and friends for their motivating and relentless support. Without them I couldn't have succeeded in completing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, there are progressively more and more situations in which the society is trying to integrate computer based devices into our day to day lives. In many cases, the inclusion of these artificial devices is motivated by the improvement of accuracy and efficiency they can provide when executing certain functionalities. Many tasks can be easily automated by computers that perform them way easier, better and faster, and it is not difficult to realise many of them are already integrated in our routines: smart houses, interconnected cars, app bots...

However, there are some other cases in which including a computer based device into someone's life does not only imply improvement but also entails a huge upgrade in the standard of living of that person.
One of the most meaningful uses of artificial intelligent systems is the use of prosthetic devices that substitute dysfunctional or amputated limbs in human beings. Unfortunately, the number of persons suffering from either one of them has been substantially increasing during the past years. The most common causes for limb amputation are accidents, tumours, circulatory disorders and infections, and according to McDonald C. and Co., in 2017 there were 57.7 million people worldwide living with some limb amputation. Based on it, a total

amount of 75850 prosthesis were approximately estimated to be needed to treat people affected by this traumatic illness around the whole world.McDonald et al. (2021)

In the past decades, the inclusion of prosthesis has been fundamental for the life improvement of these people. Therefore, this kind of studies are key to tackle the global challenge of providing functional limbs to those who don't have a proper one.

Within this topic, the most significant challenge in these cases was to be able not only to find a proper replacement of the limb that could physically do the same movements but also that could be controlled by the brain, as how it is done with functional limbs.
The need of this solution was one of the reasons that pushed the scientific world to start investigating on it and as a direct consequence, a new concept emerged.

A Brain Machine Interface (BMI) - also referred as Brain Computer Interface (BCI) - is a computer-based system that acquires brain signals, analyses them and translates them into commands that are relayed to an output device to carry out a desired action. Shih et al. (2012).

Therefore, with the use of these devices one can control the movement of any prosthetic limb just by thinking about it just as if it was the real limb the one pretending to move.
In order to do so, one needs to model the relationship that exists between the neural activity in the motor cortical region of the brain that sends the order of movement to the limb and the actual movement of it. Basically, one needs to develop a **decoding model** that predicts the kinematic vector of the limb given the generating spike trains recorded from the associated neurons. Hence, whenever the subject thinks about moving that limb, his brain sends the order to the prosthetic equivalent so that it behaves the same way.

Most of the different approaches that have been proposed in order to define such model base their procedure in signal processing. Its basis is to analyse the neural activity occurring at the motor related region of the cortex while the subject is moving one of its limbs. For the moment, these kind of studies have only been developed with laboratory animals, properly trained for the moving task they are supposed to do, but the same procedures could be extrapolated to human beings, leading to its direct application in real prosthesis.

The approach that will be taken as the basis in this research is the use of a sequential Monte Carlo (SMC) estimation algorithm that directly works on the neural spike activity, as proposed by Wang et al. (2009) among others (Schwartz et al. (2001), Brockwell et al. (2004), Shoham et al. (2005), Wu et al. (2006), Yu et al. (2007)). The model directly predicts the movement that will be produced by the limb just by analysing the brain's electrical activity. This decoding algorithm itself relies on another **encoding model**, capable of describing the causal relationship that exists between the instantaneous limb movement to the generating spike train. In other words, it takes the instantaneous limb kinematic vector and predicts the spike train that generated it. Following the literature choices, as proposed by Wang and Principe (2010), the suggested encoder is a Linear-Nonlinear-Poisson model that predicts the firing probability of one neuron and feeds it to a Poisson spike generator that outputs the resulting spike train.

Having said so, this is a research study on motor cortical neuron encoding for brain-machine interfaces, hence only the already mentioned encoding model will be exhaustively analysed and not the whole recursive decoding algorithm. Henceforth, the main objective is to develop a full encoding observation model that is capable of predicting the firing pattern of a neuron from the kinematics of a physical activity developed by the subject of study. Two different approaches have been discussed: a linear one and a nonlinear one. Hence, a secondary objective of this thesis is to analyse whether the inclusion of nonlinearities enhances the suitability of the model to encode the relationship between the kinematic vector and the corresponding spike activity.

As it has been mentioned, the main motivation for developing such model is prompted by the direct applications it has in bionic limbs. If we are capable of modelling how the body naturally reacts from a neuron's spike train, we can make some neuroprosthetic devices to behave the same way when given the same input.

Chapter 2

Problem Statement and State Of The Art

Before facing the different modelling approaches that have been considered in several researches, one needs to better understand the basis of the signals we will be treating with.

First, a brief explanation of how the brain works and how their signals are mathematically represented must be given.
Once having so in mind, a few outlines of some of the different approaches that have been followed in the field will be given. This way, the choice of the proper method will be justified somehow.

2.1 Brain electrical signals

As one may know, the nervous system and hence the brain's fundamental unit of work is the **neuron**. Neurons are cells that are excited through electrical activations and that are capable of sending information to each other through connections called synapses. These connections are the base of the complexity of it, as they are the ones that define the different meanings of these activations. Despite so, one must get down to the basics and understand how a single unit works and how these activations are produced.

Neurons are constituted by a cell body also referred as soma, some dendrites, an axon and some presynaptic terminals Gazzaniga et al. (2014). While the dendrites are the ones that receive incoming signals from other neurons, the axon is responsible for originating the signals and send them to other neurons through the presynaptic terminals. These signals are referred as action potentials and are electrically generated and sent.
In order to have a basic knowledge of how the brain works, one must understand how the information is conducted from one neuron to the other as it is its main activity. Basically, one must see the signal as a voltage spike travelling along neurons. This spike represents a concentration gradient, which is the difference between the ion concentration inside and outside the neuron. Thanks to some channels that connect both the inner and outer part of it, these ions can move in and out and change this voltage difference. This is what lets the signal travel along them.
However, in this research we are going to avoid focusing on how the signal is conducted inside the neuron and instead, one will just describe whether one specific neuron got activated or not along time, that is, whether a spike was generated within it or not. One must see the spike as a signal that has an amplitude (let's say a voltage amplitude). Bearing this in mind, one can state that a neuron will be activated whenever the amplitude of the signal surpasses a specific threshold. Therefore, at each time step, one will have a binary number that will describe whether the neuron was activated or not. At the end, it will be the pattern of these generated spikes along time that will describe what the information that is being sent refers to. Further details on how the signal will be represented are described in section 2.1.1 Mathematical representation: Point Process.

In our case, only those neurons located in the motor-cortical region of the brain will be considered as they are the ones that are activated when moving a limb. These regions are subdivided into a primary motor area (Brodmann's area 4[1]) and a variety of premotor areas (Brodmann's area 6) Gazzaniga et al. (2014). Each of these areas is constituted by a great amount of neurons that are connected to the brain stem and spinal cord and they are the responsible of the intention of movement.

2.1.1 Mathematical representation: Point Process

In order to develop either the encoding or decoding models, one has to mathematically represent the time-dependent activation and deactivation of each neuron. As it has been said, this binary possible state will be represented by a train of deltas along time. Each delta, also called spike, will represent the activation of such neuron at that moment. According to Hegger (2000), in the cortex, the timing between the different spikes is quite irregular. This irregularity in the interspike time reflects a random process. The best way to represent it is through a point process.

*A **point process** is a set of discrete events that occur in continuous time. For a neural spike train this would be the set of individual spike times* (Truccolo et al. (2005)).

Hence, given the continuous time dimension, we will consider the spikes as the different discrete set of points spanning in it.
Given an observation time span of $(0, T]$, we'll define our spike train as one realisation of the point process:

$$spk(t) = \sum_{i=1}^{n} \delta(t - u_i) \tag{2.1}$$

where $\{0 \leq u_1 \leq ... \leq u_n \leq T\}$ is one of the realisations of the point process where u_i are the times at which a spike is found and $\delta(t)$ is the *Dirac delta*[2]. Thus, a total of n spikes are found.

[1]Brain cortex has been divided into 52 different Broadmann's areas, based on a combination of cytoarchitectonic and functional descriptions.

[2]Recall that the *Dirac delta* is for continuous data.

However, instead of treating the resulting brain activity with the resulting spikes, in the majority of cases one will use the corresponding instantaneous **firing probability** instead $p_{firing}(t)$. The main advantage of it is that, when one considers the formal limit of having infinite time steps, it is continuous. Hence and against the binary spike signal, it not only provides with more information about the brain's activity but also it makes it easier to compare it with the ground truth and establish an evaluation metric. Therefore, one will use this firing probability at each time step instead, making the whole process inhomogeneous[3] and also instantaneous.

Having said so, the most suitable approach that has been considered in this neural background is the **Poisson Point Process**. Basically, it defines a point process of finite size in which the number of discrete events (a.k.a. the spikes) is a random variable that follows a Poisson distribution. Hence, one can define the probability of having n spikes in a time step dt as:

$$p(n\ spikes\ in\ dt) = e^{-p_{firing}(t)} \frac{p_{firing}(t)^n}{n!} \tag{2.2}$$

If instead of considering the firing probability we wanted to consider the firing rate defined as $spikes/timebins$, one should only divide the firing probability by the time step: $r(t) = p_{firing}/dt$.

$$p(n\ spikes\ in\ dt) = e^{-r(t)\, dt} \frac{(r(t)\ dt)^n}{n!} \tag{2.3}$$

One must take into account that this approach assumes that the spikes are independent with each other. This assumption will be taken into account along the whole study although it might not be totally accurate with the reality as it is quite clear that the spikes in one neuron aren't totally independent between them. Some of the alternatives that do take into account this dependence are considering the refractory period or bursting which will be briefly commented in Chapter **7 Future Work**.

[3]It depends on time.

2.1.2 Previous work

During the past decades, many studies have been trying to accurately simulate the brain's functioning. The complexity of it receiving stimulus from the senses and reacting to them is the main issue that the researches have been trying to address.
Focusing on the motor field and having already defined how to mathematically express the spike activity, most of the studies tried to come up with the best approach to address the desired decoding model. Recall that its purpose is to directly translate the motor cortical brain's activity into the associated limb's movement.

The first approach that was considered was to use the population vector algorithm Georgopoulos et al. (1986) as the decoding model. The algorithm basically predicts the movement direction by taking the preferred direction deduced from each of the neurons related with the locomotive system and weight them according to their corresponding tuning curves.

Further investigations changed this method to a recursive Bayesian one that also predicted the kinematic vector describing the limb's movement but this time based on probabilities Schwartz et al. (2001), Brockwell et al. (2004), Shoham et al. (2005), Wu et al. (2006), Yu et al. (2007).

At each time step and based on each neuron's spike train, it computed the posterior probability density function for all the possible kinematic states. Hence, knowing the probability distribution of all the possible kinematic states, the final movement the limb could do could be easily guessed. At the same time, this posterior probability was computed based on the prior density of that state given the current one. This prior was in turn computed based on the posterior density from the previous time step along with the error between the real generated spikes and the ones predicted from observation.
Hence, it was straightforward that there must be an accurate **observation encoding model** capable of encoding the causal relationship from the limb movement to the spike train. In

other words, given the resulting kinematic vector, it must provide a good estimation of how the neural response responsible of this movement looks like. This encoding model will be the one that will let us compute the error between the prediction of the spikes and the actual real ones in each neuron and with it, the whole decoding process shall be complete.
This approach is the one that will be considered in this research study.

All in all, the encoding model indeed plays an important role within the whole process and will contribute to the inference of movement intention given the firing pattern of a neuron. This is the model that will be build in the next Chapter **3 Problem Resolution**.

Chapter 3

Problem Resolution

In this chapter, the whole simulated-experimental procedure followed along the study will be explained in detail.

One must recall that, although in reality it is the spike train that provokes the limb movement and hence the event of facts is in that order, the encoding goal is the opposite one. The aim is to provide a model that is capable of obtaining the generative spike train given the resulting limb movement. From this point on, we will refer it as stimulus kinematic vector.

The first problem to face is that when estimating the encoding model from real data, no ground truth model is available, hence there is no direct way to evaluate it. In order to solve this problem, provide a good estimation of this encoding model and check for the reliability of the estimation procedure, some tests have been done beforehand with generated triads of input-model-output cases. The idea behind it is that, given the input stimulus and the corresponding output spike train (generated by applying a well-known encoding model to the input), when predicting the encoding model one will have a ground true model with which to compare. Hence, it will be straightforward to determine whether the estimation

procedure followed is suitable for the problem or not when applying it to real data.
As it will be explained in the following sections, several encoding model cases will be considered and compared with simulation data in order to find the suitable one to use with real data.

Having said so, the whole procedure followed in the encoding estimation resolution will be divided in two different parts: on one hand the Chapter **4 Simulated Encoding**, which will include the generation of simulated data, the definition of a realistic encoding model and its estimation procedure; and on the other hand the Chapter **5 Experimental Encoding**, which will include the collection of real experimental data and the encoding model prediction, following the procedure already checked in simulation.

Chapter 4

Simulated Encoding

As it has been said, this section will gather the full generation of simulation data, the encoding model prediction and its verification, directly entailing the correctness of the estimation procedure followed before applying it to real data.
In this sense, the steps followed for it have been:

1. Generating the input kinematic vector signal from scratch, according to the assumptions made for it.
2. Applying a well known encoding model that generates the spike train. This model will be considered as the ground truth in this chapter.
3. Develop a complete pipeline capable of estimating the model given the initial kinematic vector and the resulting spike train.
4. Checking for the correctness of the model comparing the ground truth one used in step 2 and its estimation in step 3.

This way, one will ensure to find the proper model given any two related input-output signals when applying the same pipeline.

4.1 Input data generation

The input of our encoding model represents the resulting movement of the limbs that the subject of matter does. It must be represented by a kinematic vector that specifies the position of the limb at each moment. It could represent any kind of motion, from the simplest to the most complicated one. That's why the kinematic vector can potentially be of several dimensions, with different derivatives considering not only position but velocity or even acceleration. However, at least for the simulated data and for simplicity reasons we are going to consider only 1D signals, as if the limb just moved in one direction.

In order to represent the movement, it is straightforward that the signal must be time dependent. Therefore, only 1D spatio-temporal signals will be considered and will be referred as $x(t)$.
The time resolution we will consider to represent such movement has been chosen so that it was consistent with the one used in neuroscience: the millisecond. That is why the time step between samples has been set up to $dt = 0.001\ s$.
Regarding the length of the signal we should build, only a few seconds of these signals will be used as they will already have enough samples to provide an accurate result.
Moreover, one must keep in mind that the simulation data must be as reliable as possible. It should be similar to the corresponding real data we are going to use afterwards because at the end, the procedures we apply in this chapter will be the same ones applied to real data. Therefore, and having in mind the repetitive movement the task demands, it has been decided to consider a periodic signal.

After reckoning several possibilities, the final signal considered as the **simulated kinematic vector** has been a 2 second sinusoidal signal with some uniform white noise. We have defined it as

$$x(t) = sin\left(\frac{2\pi}{T}\ t\right) + A\ \epsilon + B \qquad 0 \leq t \leq 2 \tag{4.1}$$

where T is the period of the signal, A is the noise amplitude, ϵ the uniform white noise and

B a constant that adds amplitude to the signal.

Bearing in mind that the time step is of 1 millisecond and that it has been decided that a signal length of a couple of seconds will be enough to develop a reliable model, the final input signal is constituted of 2000 samples, spanning from time 0 s to time 2 s.
The choice of the sinus has been made for clearness conclusions because the different hills and valleys in it will provide very distinguishable characteristic results that will be useful for checking the correctness of the model.
The period of the sinus has been chosen taking into account the trade-off between having enough oscillations along the time span but also not having them cluttered, making them hard to interpret. The final chosen value has been of 0.3 seconds.
Moreover, the sinusoidal signal has been added an extra uniform white noise generated by incrementing each sample by a random number between 0 and 1 weighted by an amplitude value chosen to be $A = 0.4$.
Lastly and for the sake of simplicity, we have forced the signal to be positive by shifting the whole signal above the 0 level with the parameter B.

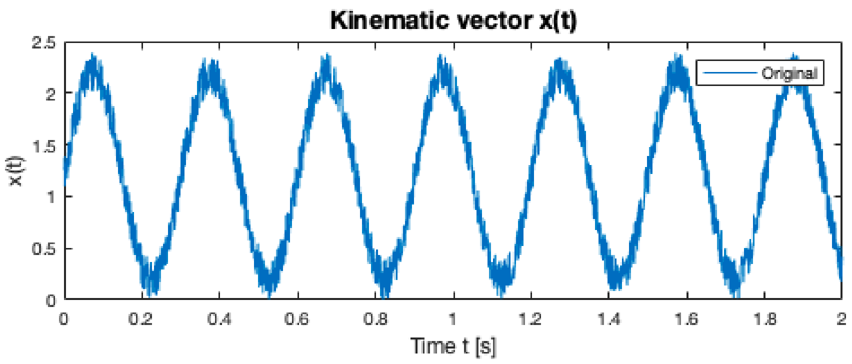


Figure 4.1: Simulation kinematic vector $x(t)$

4.2 Model definition: LNP and output data generation

Once having the input kinematic signal, one needs to find the proper encoding model that can build the corresponding generative spike train, the one that could potentially have sent the order of moving to the limb.

There have been many different approaches on how to represent the process by means of a mathematical model, starting with simple linear Wu et al. (2006) or exponential assumptions Brockwell et al. (2004) and ending with several parametric models.
However, the latest approach and the one followed in this project is the one proposed by Simoncelli *et al* Simoncelli et al. (2004): a **Linear-Nonlinear-Poisson (LNP) model**. This model directly relates the kinematic vector input signal with the resulting spike train generated in each neuron. As its name suggests, it is comprised of a linear filter, a nonlinear transformation function and an inhomogeneous Poisson spike generator. Each of these steps will be exhaustively detailed in the following subsections.

One must take into account that within the same LNP approach, a total of 4 different proposals have been considered by means of changing the tuneable parameters in the LNP structure. These 4 approaches will be referred as *Constant-Linear case, Constant-Exponential case, Window-Linear case* and *Window-Exponential case* and will be defined in the following subsections.

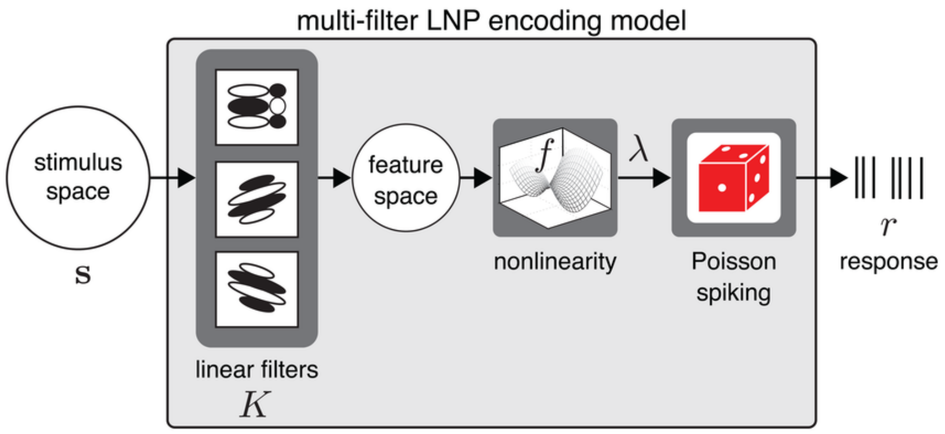


Figure 4.2: Linear-Nonlinear-Poisson encoding model pipeline. From Williamson et al. (2013)

4.2.1 Linear filter K

The linear transformation is the first one applied to the input kinematic vector signal. It consists of a filter K that projects the current signal into what is called the feature space. We define the feature space signal as

$$y(t) = K\ x(t) \tag{4.2}$$

The filter K can have many different shapes and can lead to different feature spaces. In this study, we have contemplated two different approaches with respect to it:

(A) $K = k$, a constant. In this case, the whole input kinematic vector is multiplied by this constant. From this point on, any case satisfying so will be referred as ***Constant case***.

(B) $\mathbf{K} = [\mathbf{k_1}, ..., \mathbf{k_m}]$ is a fixed length window, which is convoluted instead ($y(t) = K \circledast x(t)$), as proposed by Paninski et al.(2004a, 2004b) and Shoham et al. (2005) as well. The idea of this filter is to include some history to the feature space, that is, a sample of the resulting $y(t)$ will have information not only from that moment t but also from the past (as it considers some previous samples)[1]. For the sake of simplicity, we are going to build this window similar to a mean filter[2]: it is going to have the same value in each of its positions, *i.e.* $K = [k, ..., k]$, although they are not going to add up to 1. Hence, it doesn't compute the average value among them when convoluted but it has the same smoothing effect on the signal. The window length has been chosen to be $m = 5$. From now on, any case satisfying so will be referred as ***Window case***.

Depending on the construction of the filter and the original kinematic vector, the resulting feature space could be unidimensional or multidimensional and could have different interpretations.
Both of the approaches considered project the initial kinematic vector to the same dimensionality, that is to a 1D-spatio-temporal space, but because of the convolution performed in the second case, the length of the resulting $y(t)$ will be shorter.[3]

Notice that, as expected, in Figure 4.3(b), the $\mathbf{K} = \mathit{window}$ approach makes the feature space signal to be less noisy. This effect is thanks to the $\mathbf{K}$ applied which acts similar as a mean filter.

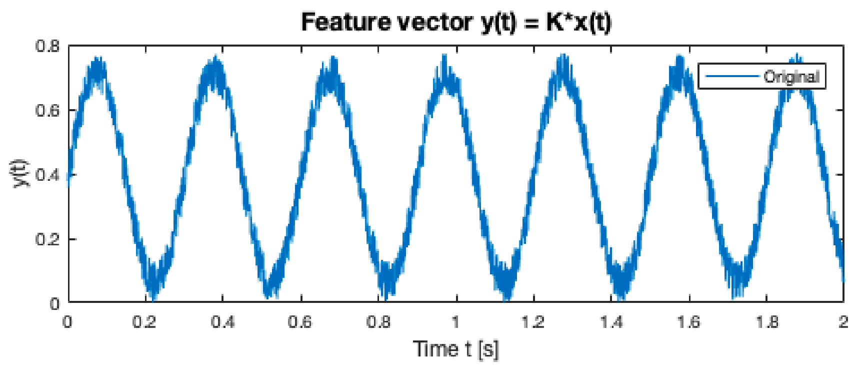
From this point on, all the pipelines and methodology applied to the initial kinematic vector $x(t)$ will be done twice, for each of the two K approaches, and their results will be separately analysed and compared.

The values that have been given to these filters can't be decided yet because they directly depend on the nonlinear function definition. They will be set in the following section so that the resulting spike train is as much realistic as possible.

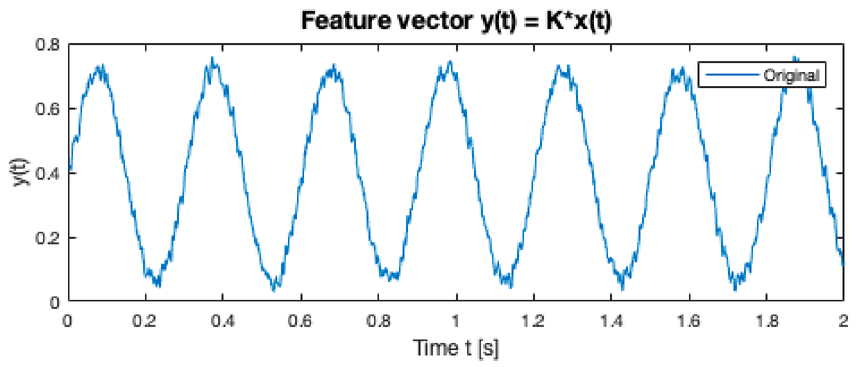
[1]Notice that, in order to consider the past and not both the past and the future, the vector $\mathbf{K}$ must be centred at its last position

[2]Mean filter requires that its terms add up to 1. It is not the case here

[3]Concretely of size $n - m + 1 = 2000 - 5 + 1 = 1996$ according to the properties of convolution.



(a) $K = ct$ approach



(b) $\mathbf{K} = window$ approach

Figure 4.3: Simulation feature space vector $y(t)$. *Note: the values of K chosen have been the ones specified in equation 4.5 and equation 4.6 (linear approach).*

4.2.2 Nonlinear function $f(y)$

The transformation that follows is the nonlinear one, represented by a function named $f(y)$. It links the feature space signal $y(t)$ with the resulting instantaneous firing probability $p_{firing}(t)$, that is, the probability of the neuron firing at each time step. It can also be seen as the firing rate fr multiplied by the time delta, that is $p_{firing}(t) = fr(t) * dt$. This resulting firing probability will be the one that will condition the number of spikes that will be generated at each time step, hence the one that will define how the resulting spike train will look like. Therefore, the instantaneous firing probability will be defined as

$$p_{firing}(t) = \lambda(t) = f(y(t)) = f(K\,x(t)) \tag{4.3}$$

Several approaches can be considered when defining the nonlinear function f. This study covers two different possibilities: a simpler linear one and a more complex one that tries to include some nonlinearities to the process.

(A) Linear $\mathbf{f(y(t)) = a * y(t) + b}$, as proposed by Wu et al. (2006). Any case considering this function will be labelled as ***Linear case***. For the sake of simplicity, we are going to set $a = 1$ and $b = 0$ for simulated data.

(B) Exponential $\mathbf{f(y(t)) = a * exp(b * y(t)) + c}$, as proposed by Brockwell et al. (2004). Similarly, any case considering this function instead will be labelled as ***Exponential case***. Similarly, we are also going to set $a = 1$, $b = 1$ and $c = 0$ in this case.

Thus, the four approaches considered in this project are the result of combining both the constant-window approach and the linear-exponential one. Hence, now one knows the differences between the already mentioned *Constant-Linear case, Constant-Exponential case, Window-Linear case* and *Window-Exponential case.*

One must make sure that this step leads to realistic firing probability values in line with how often a neuron in a human brain fires. Knowing that it should lie between 0.1 and 0.7[4], we can infer where the firing rate should lie and hence what should be the value of K.
Let's differentiate the four different proposals.

Linear $\mathbf{f(y(t)) = y(t)}$

$$p_{firing}(t) = f(y(t)) = y(t) = K\,x(t) \tag{4.4}$$

Following this expression, the mean value of $x(t)$ will be taken instead of the time varying

[4]Mean values proportioned by the laboratory, based on the experience when recording real spikes data

signal. Therefore, K should lie between

$$\frac{0.1}{mean(x(t))} \quad and \quad \frac{0.7}{mean(x(t))}$$

Considering the two K approaches we can distinguish:

- ***Constant-Linear case*** $\rightarrow$ The chosen value is the mean one in between the two limits
$$K = \frac{1}{2}\,\frac{0.1 + 0.7}{mean(x(t))} \tag{4.5}$$

- ***Window-Linear case*** $\rightarrow$ The chosen window will be constituted by several terms whose sum will add up to the $K = ct$ value. If a mean filter of length m is used, one has
$$\mathbf{K} = \frac{1}{m}\;K\;[1, ..., 1] = \frac{1}{m}\;\frac{1}{2}\frac{0.1 + 0.7}{mean(x(t))}[1, ..., 1] \tag{4.6}$$

Exponential $\mathbf{f(y(t)) = exp(y(t))}$

$$p_{firing}(t) = f(y(t)) = exp(y(t)) = exp(K\,x(t)) \tag{4.7}$$

Same way as before, $x(t)$ will be substituted by its mean value. Consequently, K should lie between

$$\frac{1}{mean(x(t))}\;log(0.1) \quad and \quad \frac{1}{mean(x(t))}\;log(0.7)$$

Again, distinguishing between both K cases one has:

- ***Constant-Exponential case*** $\rightarrow$ The chosen value is the one in between both limits
$$K = \frac{1}{2}\;\frac{1}{mean(x(t))}\;log\,(0.1 * 0.7) \tag{4.8}$$

- ***Window-Exponential case*** $\rightarrow$ The chosen window will be constituted by several terms whose sum will add up to the $K = ct$ value. If a mean filter of length m is used, one has

$$\mathbf{K} = \frac{1}{m}\ K\ [1, ..., 1] = \frac{1}{m}\ \frac{1}{2}\ \frac{1}{mean(x(t))}\ log\,(0.1 * 0.7)\,[1, ..., 1] \tag{4.9}$$

Only the *Window-Exponential* case resulting firing probability has been plotted in Figure 4.4 as it is the most general case and the equivalent graphics from the other cases are pretty similar. It can be stated that by applying the proper K values we obtain a firing probability laying approximately between the desired 0.1 and 0.7 values.

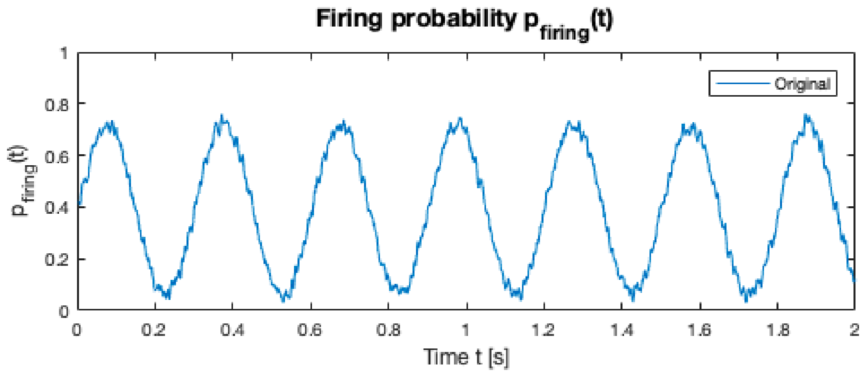


Figure 4.4: Firing probability $p_{firing}(t)$ for *Window-Exponential case*

4.2.3 Inhomogeneous Poisson Spike Generator

As a final step, the estimated firing probability must derive to the generation of the final spike train, which should represent how a single neuron fired to the given input kinematic vector. The assumption taken is that these spikes are generated following an inhomogeneous Poisson spike train generator whose rate parameter λ is the firing probability itself.

$$P\left(\frac{n\ spk}{dt}\right) = e^{-\lambda}\frac{\lambda^n}{n!} \tag{4.10}$$

The reason why the Poisson process is chosen is because the time between successive spikes is quite irregular and could be fitted to an exponential distribution (as happens with the

Poisson process). However, one must assume that the spikes are independent with each other which might not always be true in neuroscience. For the moment, this assumption will be made, but more realistic approaches could be used (see chapter **7 Future Work**). In addition, it is the fact that a time varying spike rate is considered that makes the process to be inhomogeneous.

Following the inhomogeneous Poisson spike generator premise, a spike will be generated whenever a random number generated under the hypothesis of a uniform distribution satisfies

$$rand(\ ) < p_{firing}(t) \tag{4.11}$$

The function $poissonSpikeGen\_prob(fr\_prob, nBins, nTrials)$ has been created in order to generate such spikes (see section 8.3 Matlab codes for the code).
The arising spike train will be mathematically expressed as a binary signal along time, with unit value when there is a spike and null value otherwise. It will have the same length as the pertinent kinematic vector signal (see Figure 4.5 for the *Window-Exponential* case). As expected, whenever the firing probability is high, the amount of spikes generated is considerably higher as well.

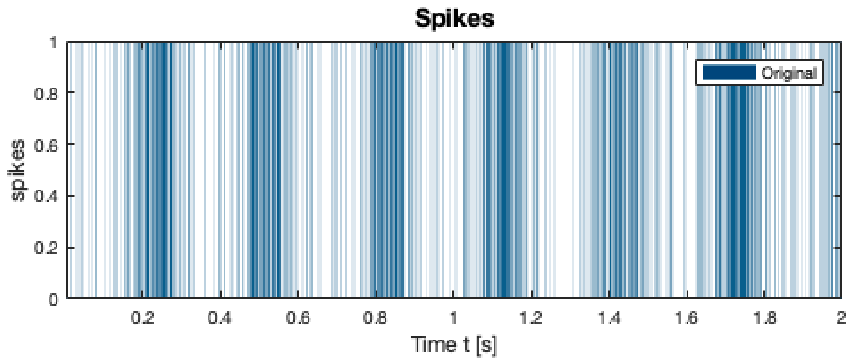


Figure 4.5: Resulting spike train for *Window-Exponential case*

4.2.4 Time delay Δt consideration

At this point, one should consider the fact that in real neural behaviour, the whole kinematic vector could be delayed in time.
It is true that, in the neural system, signals travel very fast along nerves, however, these fibres aren't optimal and as any other signal transfer, there exists a time delay between the input signal and the corresponding output one. In this field, these kind of delays are of the order of milliseconds but they must be taken into account as they are relevant given the time resolution we are dealing with.

Therefore, there should exist a little time delay between the spike train and the corresponding kinematic vector if they are meant to represent the reality as accurately as possible. In order to depict that in data, once having the input signal generated, the desired time delay will be directly applied on it. This shifted signal will be the one that will be fed to the LNP model in order to obtain the desired shifted spike train. By doing so, one will already have both the input and output simulated data that will be used in order to predict the model: the shifted resulting spike train $spk_{\Delta t}(t)$ (having propagated the shifted $x(t)$ through the theoretical LNP model) and the original kinematic vector (without the time delay).

For the simulation data, one has chosen the time shift to be of 100 time samples, which is of $\mathbf{\Delta t} = 100 * dt = \mathbf{0.1}s$.
If we plot the feature space vector $y(t)$ that gathers all the possible information from the original kinematic vector along with the shifted spikes for the *Window-Exponential* general case, we obtain the resulting Figure 4.6[5]. The time delay is clearly manifested by 0 spikes at the beginning. The idea when feeding the model with this data is that it also predicts the time delay between both signals.

All this being said, simulated data is ready to be used in order to estimate the model

[5]The fact that the $y(t)$ values are negative here in contrast with Figure 4.3(a) and Figure 4.3(b) is because of the $\mathbf{K}$ value chosen in the exponential case so that the firing probability range was between 0.1 and 0.7.

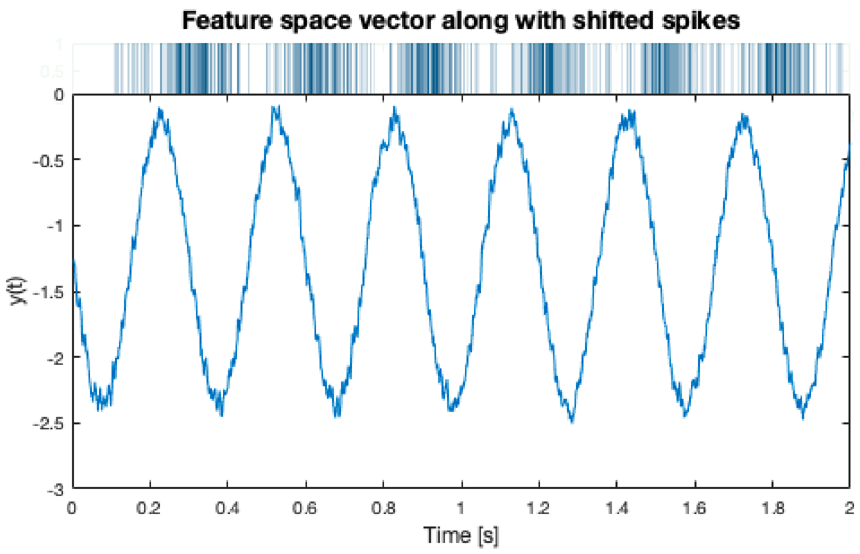


Figure 4.6: Resulting shifted spike train along with $y(t)$ for *Window-Exponential case*

it hides between the input and output results and that's what will be done in the following section.

4.3 Model parameter estimation

Recall that the aim of this previous analysis with simulated data was to ensure that the pipeline followed given the input kinematic vector and the resulting spike, provided a good estimation of the model. This is what is going to be proved in this section. The whole procedure will be again followed by each of the 4 approaches.
Presuming that the the LNP model is suitable for this purpose, the prediction becomes a fitting problem in which only the parameters of this LNP model have to be estimated. Hence, both the linear filter K and the nonlinear function $f(y)$ must be calculated, apart from the assumed time delay.

4.3.1 Linear filter K estimation: STA

First of all, the method to predict the linear filter K applied between the kinematic vector and the resulting feature space must be defined. We are going to provide the procedure given any signal $x(t)$ and the resulting spike train $spk(t)$.
We will see that this estimation procedure needs to be done along with the time delay estimation as they both depend on each other. The whole final procedure will be further explained in section 4.3.2 Time Delay Δt Estimation: Mutual information but for the moment, the individual K estimation will be defined here as if we already knew the time delay and both signals were already matched in time.

The Spike Triggered Average will be the tool that is going to be used for the linear filter estimation.

Spike Triggered Average (STA)

The Spike Triggered Average (STA) method is a tool for characterising the ideal input a neuron needs in order to fire based on both the spike train and the corresponding kinematic vector given as input.
The idea is to consider a fixed number of samples before the generation of every spike (that is a window in time), then take the kinematic vector signal in each of these windows and average it over them. The result of it will be the optimal filter that will lead to the generation of the spikes. It is named the receptive field of the current neuron.

Definition 4.3.1. Let $\mathbf{x}_i$ be a fixed length window of the 1 dimensional kinematic vector preceding time step i, and n_i the number of spikes occurred in that i'th time bin.[6] Then,

[6] Recall that, as how we have defined the problem, either one or zero spikes will occur in one time bin and thus n_i will be a binary variable indicating whether a spike was produced ($n_i = 1$) or not ($n_i = 0$)

the Spike Triggered Average is computed as:

$$STA = \frac{1}{n_{spk}} \sum_{i=1}^{T} n_i \mathbf{x_i} \tag{4.12}$$

in which n_{spk} is the total number of spikes along the whole signal spanning from time 0 to T.

* * *

To follow up, one must distinguish between the different approaches of encoding, starting from the differences between the *Window-Linear* and the *Window-Exponential* approach. The corresponding *Constant-Linear* and *Constant-Exponential* ones are straightforward. Let's first consider the linear one.

- ***Window-Linear case***: we have $\lambda(t) = p_{firing}(t) = y(t) = \mathbf{K}x(t)$. In matrix form and considering that the window size is T, one has that the instantaneous firing probability Λ is:

$$\Lambda = X\mathbf{K}$$

$$\begin{bmatrix} \lambda_T \\ \lambda_{T+1} \\ \vdots \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & \cdots & x_T \\ x_2 & x_3 & \cdots & x_{T+1} \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix} \begin{bmatrix} K_1 \\ K_2 \\ \vdots \\ K_T \end{bmatrix}$$

$$L(\text{Data}, \mathbf{K}) = \|X\mathbf{K} - \Lambda\|^2 = (X\mathbf{K} - \Lambda)^\top (X\mathbf{K} - \Lambda) =$$

$$= \Lambda^\top \Lambda - \Lambda^\top X\mathbf{K} - \mathbf{K}^\top X^\top \Lambda + \mathbf{K}^\top X^\top X\mathbf{K}$$

$$\frac{\partial L(\text{Data}, \mathbf{K})}{\partial \mathbf{K}} = -2X^\top \Lambda + 2X^\top X\mathbf{K} = 0$$

$$X^\top \Lambda = X^\top X\mathbf{K}$$

$$\mathbf{K} = \left(X^\top \, X\right)^{-1} \left(X^\top \, \Lambda\right) \tag{4.13}$$

One must notice that the expression we have just deducted essentially matches with the STA definition: the summation in equation 4.12 is translated to $(X^T\Lambda)$ in matrix form (one must match the number of spikes per bin n_i in the former expression with the firing probability λ_i in the latter one). The remaining $1/n$ term that averages the computation is translated into $(X^T\,X)^{-1}$, which is the autocorrelation matrix that has the same effect.

Hence, this is the method we'll use to predict the linear filter $\mathbf{K}$. The case of $K = ct$ is a specific case of this generalisation and therefore is solved the same way.

$$\begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix} K$$

Note that against the previous case, the resulting samples start at the first time position (hence the subindex 1 in the first λ sample), instead of at the T sample.

- ***Window-Exponential case***, the only difference is that the result of the linear filter and the nonlinear transformation isn't the firing probability itself but its logarithm. Since one has $\lambda(t) = p_{firing}(t) = exp(y(t)) = exp(\mathbf{K}\,x(t))$, we have to use $log(\lambda(t)) = \mathbf{K}\,x(t)$ to solve the problem the same way.
  Consequently, the solution in this case is

$$\mathbf{K} = \left(X^\top\ X\right)^{-1}\left(X^\top\ log(\Lambda)\right) \tag{4.14}$$

  Again, the constant approach can be solved the same way.

Using both the equation 4.13 and equation 4.14 one can predict the linear filters.

However, it must be noticed that they depend on both the kinematic vector X which we have and the firing probability Λ which we don't when using real data. One must always remember how the real data looks like, as the only purpose of this simulation experiment is to find the proper method to treat with real data. In this respect, one only possesses both the kinematic vector and the spike train. That's why one can't use the well known firing probability Λ when using the simulated data but we have to estimate it from the resulting spike train.

Firing probability estimation

One must remember that this spiking probability is time dependent and must lay between 0 and 1. So, somehow one should find the way to transform the spike train along time into a firing probability changing in time as well. Therefore, at each time step one can obtain the instantaneous firing probability that led to the generation or not of a spike at that moment.

With the aim of estimating the firing probability, the easiest way to do that is performing a **convolution** between the spike train and a Gaussian distribution function. Once having done so, one can obtain an estimation of the firing probability density by adding all of these Gaussian curves together. Hence, the time regions in which the neuron fired a lot will have high firing probabilities, while those in which only a few spikes were generated will have low firing probabilities.

One must realise that there are two parameters that must be fitted in this methodology: both the mean μ and the variance σ of the convoluted Gaussian distribution ($N(\mu, \sigma)$). While it is quite clear that the mean should be $\mu = 0$ in order to have the Gaussian distribution centred when convoluted with the spikes, it is quite difficult to find a proper automated method capable of deciding the optimal variance value σ. This parameter makes the Gaussian distribution to be wider or narrower. In order to choose its value, we are going to try several different values and compare the resulting firing probability with the theoretical one that will act as ground truth.

Moreover, one can notice that by performing such convolution, the resulting values will not have to be lower than one and hence one must normalise them. Therefore, the resulting

estimation of the firing probability must be divided by its maximum value.
Furthermore, and in order to make the estimation as accurate as possible, it has been decided that the mean value of the estimated firing probability should be equal to the number of spikes divided by the number of time steps in that realisation. Consequently, one will have to multiply the already normalised estimated firing probability by $\frac{\#spikes/\#timebins}{mean(estimated\, p_{firing})}$. Since this factor might be higher than one in some cases, one will only perform the multiplication whenever it is lower than 1.

So at the end, the criterion followed to choose the σ parameter has been that the root mean square error between the estimation and the ground truth was minimum. The final chosen parameter has been $\sigma = 1.9650$.

All in all, the linear filter K can be estimated for all of the cases considered, linear-nonlinear and $K = ct$ or $K = window$ ones.

4.3.2 Time delay Δt Estimation: Mutual information

Along with the K parameter, the first model parameter that must be estimated is the time delay. Once this is calculated, either the input or the output signal can be shifted so that both keep up with each other which makes the rest of the parameters estimation to be much easier.

According to Paninski et al. (2004b) and Wang et al. (2007), the causal time delay can be estimated following an information theory procedure. What we are looking for is the optimal time delay at which the input signal holds the maximum information about the subsequent spike train. In order to measure it, the concept of mutual information is used.

Mutual information

Mutual information is the metric proposed by Cover and Thomas (1991) which measures the mutual dependence between two random variables. More intuitively, it measures the amount of information, in terms of bits, that we get to know about one of the random variables by observing the other. It is a much more general metric than the well known correlation as it doesn't only consider linear relations but also any other kind.

Definition 4.3.2. Given two different random variables X, Y, its mutual information is defined as

$$I(X;Y) = \int_Y \int_X p_{X,Y}(x,y)\ log\left(\frac{p_{X,Y}(x,y)}{p_X(x)p_Y(y)}\right)\ dx\ dy \tag{4.15}$$

where $p_{X,Y}(x,y)$ is the joint probability density function of the random variables X and Y, and $p_X(x)$ and $p_Y(Y)$ are the respective marginal probability density functions. Therefore, the mutual information is bigger when the difference between $p_{X,Y}(x,y)$ and the product $p_X(x)p_Y(y)$ is bigger as well. If both variables were totally independent, both terms will be the same and hence the logarithm result will be $log(1) = 0$.

For the case we are dealing with, the mutual information is calculated for the resulting shifted spike train [7], say $X = spk_{\Delta t}(t)$, along with an hypothetical shifted input signal. We are going to use the feature space vector $Y = y_{lag}(t)$ because it gathers all the possible information hided in $x(t)$ (it lets us include history information for the $\mathbf{K} = window$ approach). The idea is that whenever we shift the signal the right amount of samples, it will match the shifted spikes and hence the mutual information will be maximum.

One must have noticed that one initially doesn't have the feature vector $y_{lag}(t)$ as the filter K is unknown. Hence, it must be estimated at the same time as the time lag and that's why their estimation depends one another. The corresponding procedure described within section 4.3.1 Linear filter K estimation: STA will be used here and will be referred as '*STA K estimation*'.

[7] Recall that this spike train is the result of feeding the LNP model with the shifted input signal. The goal is to find the time delay that was applied to this initial signal.

Following with the mutual information computation, the spike train is a discrete signal, one of the integrals from the previous formula should be changed to a summation. Thus, if we refer $spk_{\Delta t}(t)$ as spk and $y_{lag}(t)$ as y_{lag}, one has:

$$\begin{aligned} I(spk; y_{\text{lag}}) &= \int_y \sum_{spk=0,1} p(spk, y_{\text{lag}}) \log_2 \left(\frac{p(spk, y_{\text{lag}})}{p(spk)\ p(y_{\text{lag}})} \right) dy_{\text{lag}} = \\ &= \int_y \sum_{spk=0,1} p(y_{\text{lag}} \mid spk) p(spk) \log_2 \left(\frac{p(y_{\text{lag}} \mid spk)}{p(y_{\text{lag}})} \right) dy_{\text{lag}} \end{aligned} \tag{4.16}$$

where the integral is computed along the whole time span of the input signal $y_{\text{lag}(t)}$.[8]

* * *

Once we have the mathematical expression, the procedure of how to predict the optimal time lag applied (and also the optimal linear filter K) given both the input signal $x(t)$ and the shifted spike train can be described as follows:

1. Estimate the firing probability from the spike train, needed for the '*STA K estimation*'.

2. Compute the probability of having a spike. The probability of not having a spike will be the complementary.

$$p(spk = 1) = \frac{\#spikes}{\#time\ bins} \tag{4.17}$$

$$p(spk = 0) = 1 - p(spk = 1) \tag{4.18}$$

3. Several hypothetical time delay guesses lag are considered. For each of them:

3.1 One shifted input signal is created, that is, the kinematic vector signal $x(t)$ is shifted with the time delay lag leading to $x_{\text{lag}}(t)$. From this step on, $x_{\text{lag}}(t)$, $spk_{\Delta t}(t)$

[8]Bayes theorem has been used to pass from the first eqution to the second: $p(spk \mid y_{\text{lag}}) = \frac{p(y_{\text{lag}} | spk) p(spk)}{p(y_{\text{lag}})}$

and the estimated $p_{firing}(t)$ will be cropped so that only the samples after that time delay guess are considered. One has to do that because in real data, we would not know what's before that and one can't use information from the past we wouldn't have.

3.2 One performs the '*STA K estimation*' using both the $x_{\text{lag}}(t)$ and the shifted spike train leading to the corresponding $K_{optimal}$. With it, one calculates the resulting $y_{\text{lag}}(t) = K_{optimal} * x_{\text{lag}}(t)$.

3.3. Mutual information between the shifted feature space signal $y_{\text{lag}}(t)$ and the resulting spike train $spk_{\Delta t}(t)$ is computed following equation 4.16. To do so, we have to precompute the following probability density functions.

– $p(y_{\text{lag}})$, the probability density function of y_{lag}.

– $p(y_{\text{lag}} \mid spk)$, the probability density function of y_{lag} conditioned to both options $spk = 1$ and $spk = 0$.

4. The optimal time lag applied will be the one leading to the maximal mutual information value.

$$\text{lag}^* = \underset{\text{lag}}{\arg\max}\ \ I\left(\text{spk}; y_{\text{lag}}\right) \tag{4.19}$$

One will also store the $K_{optimal}$ obtained for such time delay which will be the final estimation for it.

One must notice that one doesn't know neither of the probability density functions needed for step 3.3. Hence, they must be directly estimated from data.
In order to do that, one should consider using what is called Kernel Density Estimator.

Kernel Density Estimator (KDE)

Definition 4.3.3. Let $(x_1, x_2, \dots x_n)$ be i.i.d.[9] samples from univariate distribution with unknown density function $g()$. Its Kernel Density Estimator (KDE) is

$$g_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h\left(x - x_i\right) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) \tag{4.20}$$

where K is the kernel (non-negative function) and h is a smoothing parameter that can be chosen. In our case a Gaussian kernel will be chosen: $K(x) = N(0,1)$.

Graphically, KDE consists in convolving the chosen kernel $K(x)$ with each sample position and add them all. Using a Gaussian kernel, the resulting density function estimation should be the 'continuous' version of the corresponding histogram.

* * *

To compute these probability densities, we are going to build a MATLAB function that will be referred as $[f_p, values] = kernel\_smoothing(var)$.

Definition 4.3.4. $[\mathbf{f_p}, \mathbf{values}] = \mathbf{kernel\_smoothing}(\mathbf{var})$ (see section 8.3 MATLAB codes for the code)
It will create a probability density function f_p for the input variable var. The $values$ term will be a grid of different equally spaced values spanning along the original range of the variable var.
In it, the MATLAB function $histfit(var, n_{bins}, 'kernel')$ will be first used. It builds an histogram of the variable var with n bins. Moreover, it tries to fit a distribution specified by the third variable. The option that is going to be used is the $'kernel'$ one, which uses a non-parametric kernel-smoothing distribution. With it, the density is evaluated at 100 equally

[9]Independent identically distributed

spaced points that cover the range of the variable var. The input n_{bins} has been chosen to be 100 as well.
However, after so, the fit we have isn't yet a distribution; that's why we have to divide it by the number of samples the histogram has been built with.
Then, one will just have to transform it into a function so that given a var value, it gives the probability of having it. To do so, we have used the MATLAB function $F = griddedInterpolant(grid, values\_grid, InterpolationMethod, ExtrapolationMethod)$ that creates an interpolating function F based on the $values\_grid$ we have at each point defined by the $grid$. Therefore, the $grid$ will be constituted by the different var values, and $values\_grid$ will be the corresponding computed probabilities. The $InterpolationMethod$ has been chosen to be 'linear'. According to MATLAB Documentation, '*The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension*'. Similarly, the $ExtrapolationMethod$ has been chosen to be 'nearerest'. According to the documentation, '*Nearest neighbor interpolation. The interpolated value at a query point is the value at the nearest sample grid point*'. It has been chosen this way as the extrapolated values at the edges may get negative if a linear extrapolation was chosen; hence this way, they are maintained the same as the last values from data. Consequently, one must keep in mind that the results one will obtain near to the edges won't be as reliable as they will come from extrapolation and not from real data.

Hence, this function will be used to compute both $p(y)$ and $p(y \mid spk)$.
Moreover, when dividing these two probability density functions they must be evaluated at the same points. In order to do that, both functions will be evaluated at the same equally spaced points, more concretely, to the grid resulting from the $kernel\_smoothing$ function when used with $p(y)$ as its range is for sure bigger or equal to the one obtained with its conditioned $p(y \mid spk)$.

Once having the estimation of the density functions, the mutual information can be directly computed and hence the optimal time lag is found.

One must take into account that, as the signal $y(t)$ we are using is periodic, the mutual information could also be maximum at shifts that are multiples of its period T plus the time lag (e.g. $T + lag$ or even $T/2 + lag$) as we can see in Figure 4.7. That is why the mutual information computation will only be carried out with time lag guesses smaller than half the period of the sinusoidal signal.

$$lag_{guess} < \frac{T}{2} \tag{4.21}$$

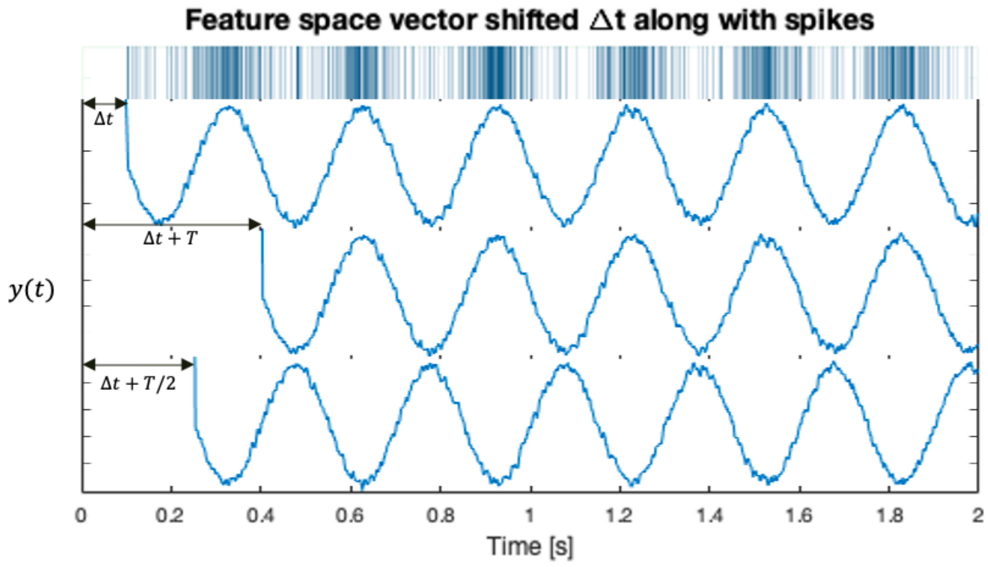


Figure 4.7: Shifted spike train along with possible $y_{lag}(t)$ for *Window-Exponential case*

Once we have the resulting plot of the mutual information against the different time lag guesses, one needs to find its maximum as it will indicate the optimal time delay to apply. However, the mentioned plot is kind of noisy, with high frequency oscillations that make it difficult to extract the right maximum. That's why one needs to process it in order to extract the general trend and hence the real maximum value.

Low-pass filtering mutual information

In order to extract the general tendency the mutual information signal has, one must apply a low-pass filter to it. Considering that the sampling interval is the time step $Ts = 1{*}10^{-3}s$, we

can compute the sampling frequency as $Fs = 1/Ts = 1/dt = 1000\,Hz$ and the corresponding Nyquist frequency as $Fn = Fs/2 = 500\,Hz$.
One must ensure that the chosen cutoff frequency Fc[10] of the filter we are designing should be lower than the Nyquist one in order to avoid aliasing $Fc < Fn$.

It is not direct to obtain the optimal cutting frequency to filter the mutual information signal but a good approach is to compute the **Fast Fourier Transform (FFT)** of the signal.
FFT is a tool to transform any signal from its time domain into its frequency domain. The idea behind it is that the same data can be decomposed in a series of sinus waves with different periods and hence frequencies. Therefore, the FFT provides us with the frequency spectrum of the signal, giving an insight of both the high and low frequencies that are present in it.
With this spectrum, one can see which low frequencies represent the general trend of the signal and which high frequencies are associated with noise. Knowing so, one is ready to choose the cutoff frequency Fc above which the signal will be attenuated: Fc will be chosen as the frequency at which the FFT has dropped for the first time.
In order to do so, the MATLAB *fft(x)* function was used and the first minimum found in the resulting spectrum was chosen as the cutoff frequency Fc. It must be taken into account that this cutoff frequency should still be lower than the Nyquist one Fn so that no aliasing is produced.

Once the parameters have been defined, the filter itself is computed with the MATLAB function $fir1(n, Wn)$ which designs a window-based FIR filter of order n. However, the function works with normalised frequencies instead that must lay between 0 and 1 so the normalised cutoff frequency should be provided instead. Since Fn is the maximum possible value the cutoff frequency could have, its normalisation procedure will be computed as follows: $Wn = Fc/Fn$ so that it led to 1 whenever $Fc = Fn$, its maximum value. Having

[10]The cutoff frequency is the one from which the low pass filter will attenuate the signal.

said so, the order n is yet to be decided. The impact this parameter has in the filter is just to specify how fast the trend of the magnitude in the frequency domain drops to 0. The highest the value, the faster it falls. Two different values were considered ($n = 10$, $n = 20$) and based on observation, the first case was the one giving more reliable results.

* * *

Having obtained the corresponding filter and having applied it to the original mutual information signal, the maximum value has been found and the corresponding time delay has been chosen as the optimal one. Once having it, one can shift the original kinematic vector $x(t)$ onwards so that it matches the originally shifted spike train in time. This way, the further estimations are much easier.
The corresponding linear filter $K_{optimal}$ is also stored as the optimal one for the model.

4.3.3 Nonlinear function $f(y)$ Estimation: Bayes Theorem

Finally, one must estimate the nonlinear function $f(y)$ to lastly have all of the puzzle pieces. One can see the nonlinear function as the one that represents the conditional density function $p(spk \mid y) = p(spk \mid Kx)$. So, following the Bayes Theorem[11], one can state that

$$f^*(\cdot) = p\left(spk \mid y\right) = \frac{p\left(spk, y\right)}{p\left(y\right)} = \frac{p\left(y \mid spk\right) p(spk)}{p\left(y\right)} \tag{4.22}$$

This can be seen as the well known expression of *posterior* $=$ *likelihood* $*$ *prior* by identifying $p\left(spk \mid y\right)$ as the posterior, $p(y \mid spk)$ as the likelihood, $p(spk)$ as the prior and $p(y)$ as a normalization constant.

Recall that each of the terms is a distribution except for $p(spk)$ which is a number; hence, they must be estimated. To do so, and similarly as before, we are going to use Kernel Density Estimators (KDE) (see Definition 4.3.3):

[11]Bayes Theorem says $p(A \mid B) = \frac{p(A,B)}{p(B)} = \frac{p(B|A)p(A)}{p(B)}$

- $p(y \mid spk)$ is the distribution of $y = Kx$ with both the restrictions of $spk = 0$ or $spk = 1$. It is calculated using the defined function $[f_p, values] = kernel\_smoothing(var)$ with $var = y \mid spk$ (see Definition 4.3.4).
- $p(spk)$ will be calculated adding up the amount of spikes there has been divided by the total number of time steps. Hence it is a scalar.
- $p(y)$ is simply the distribution of $y = Kx$ without any restrictions. It is also calculated with the function $[f_p, values] = kernel\_smoothing(var)$ but this time with $var = y$.

Again, the density functions must be transformed into functions and must be evaluated at the same equally spaced points so that they can be divided.

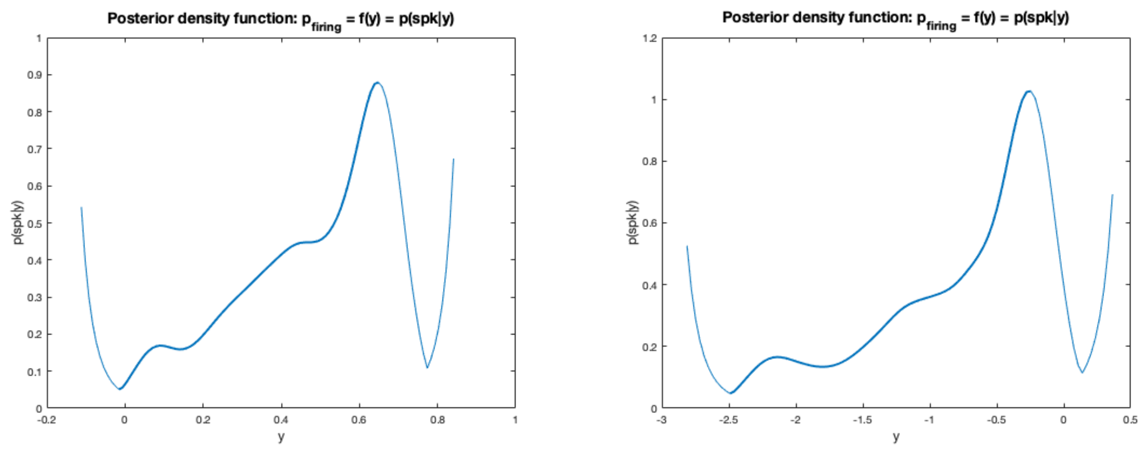
Until here, the procedure was the same for any of the cases considered. Of course, some differences must be found between the linear and the exponential proposals since one expects the results to resemble such functions.

f(y) fitting

In both cases, one can try to fit a function of the same form ($a * y + b$ for the linear and $a * exp(b * y) + c$ for the exponential one), predict the optimal parameters a and b (and c if it is the case) and compute the error with respect to the theoretical ones used.

In order to do that, one can use the MATLAB $g = fittype(expression)$ function along with $optimal\_fit = fit(x, y, g)$. They finally provide the $optimal\_fit$ of the y terms along the x values following the $expression$ parameter. This parameter will be set as $'a * x + b'$ for the linear case and $'a * exp(b * x) + c'$ for the exponential one.

However, and based on what has been said that the edges aren't as reliable, only the monotonically increasing regions of the nonlinear function will be considered for the fitting. In one hand, the left limit will be the minimum value that the feature space y achieved during the estimation of both the probability density $p(y)$ and $p(y|spk)$, both used in the $f(y)$ function estimation. On the other hand, the upper limit must be chosen as well.



(a) *Window-Linear* case (b) *Window-Exponential* case

Figure 4.8: $f(y)$ estimation for *Window* cases

If one observes the resulting estimation of the nonlinear function in Figure 4.8, one can notice that, although the trend initially looks like a linear (in Figure 4.8(a)) or an exponential function (Figure 4.8(b)), it drops to zero at the end. One must be aware that this result is totally expected because not only the leftest region of $f(y)$ but also the rightest region of it have a level of confidence way lower than the middle region. In other words and keeping in mind that this estimation has been made based on data, the amount of samples we have in these y values is not as much as the amount of samples we have in the middle region. Hence, the estimation of both the probability densities and consequently of the $f(y)$ computation at these regions is not accurate at all. Having so in mind, one has chosen the y position corresponding to the maximum value of $f(y)$ as the upper limit one will consider when fitting either the linear or the exponential form. One can see the final range of the nonlinear function that has been used for the fitting marked as a thicker line.

With it, one can estimate all the different parameters that define the desired LNP model and hence we have the pipeline well defined in order to further apply it to the experimental data.

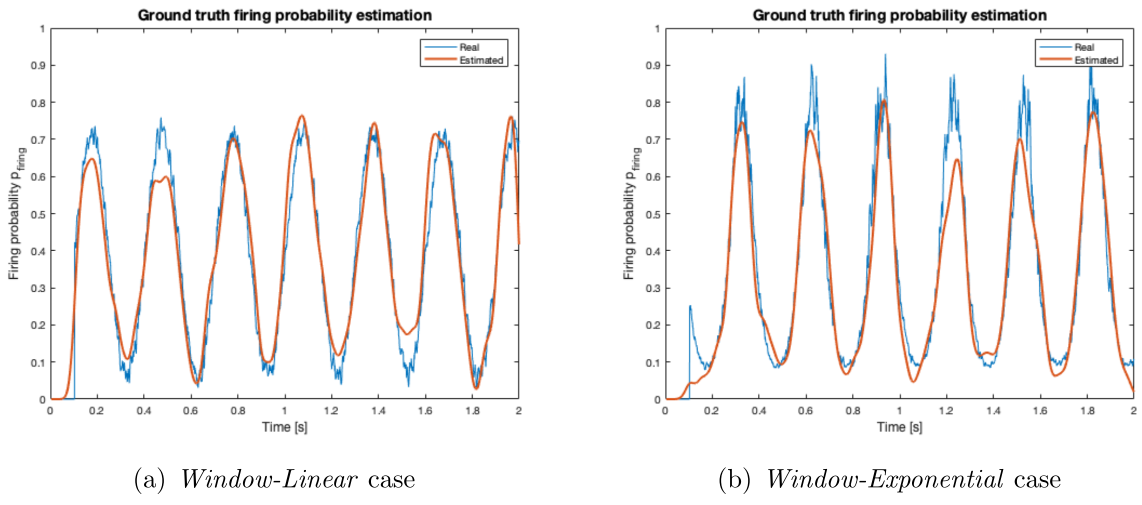
4.4 Model Results

Now that we know the whole procedure, we are going to present the results obtained for each of the 4 different approaches.
Several partial results have been found for each of them and will be gathered in tables below. However, the corresponding plots will only be displayed for the *Window-Linear* and *Window-Exponential* cases as they are the most general ones and will be the ones that will be assumed in the experimental encoding procedure. The rest of the plots corresponding to both the *Constant-Linear* and *Constant-Exponential* will be displayed in section 8.2.1 Simulated Encoding additional Figures and Tables in the Appendix.

4.4.1 Linear filter K and Time delay Δt results

So, starting from the beginning, the first parameters estimated are both the time delta and the linear filter K.
Its estimation included as first step the computation of the **firing probability** $\mathbf{p_{firing}(t)}$ from the shifted spike train. One can see in Figure 4.9 that it makes sense as whenever the probability is high, more spikes are generated. Also, it lies between 0 and 1. One must remember that it has been estimated so that its mean value matched the quotient between the number of spikes and the number of time samples.

In order to see how accurate the results are, the root mean square error (rmse) between the approximation and the real firing probability will be computed. The resulting values are gathered in Table 4.3 and will be commented in that section.



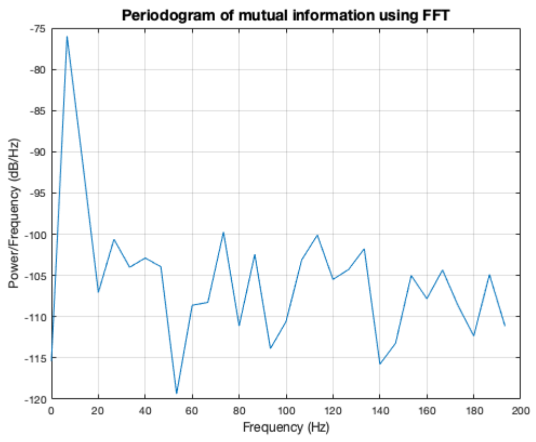
(a) *Window-Linear* case (b) *Window-Exponential* case

Figure 4.9: p_{firing} estimation for *Window* cases

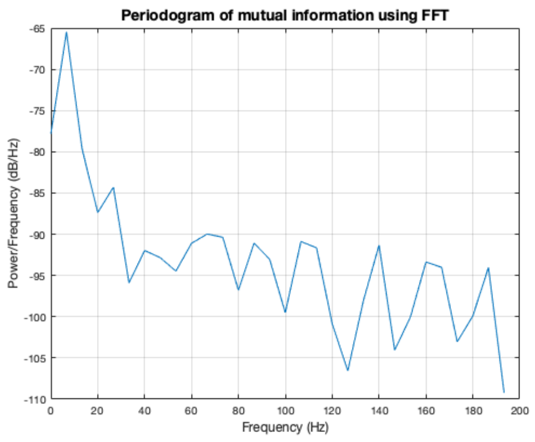
Following with the pipeline, the next computation was to find the time lag guess that led to the maximum mutual information. However, the resulting mutual information is quite noisy and had to be filtered.

In order to decide which filter to apply, remember that the Fast Fourier Transform (FFT) of the mutual information plot was computed. The resulting plots for both the *Window* cases are represented in Figure 4.10. From them one wants to know which are the frequencies that are present in the given signal. With it, if we identify the low frequencies present in the signal we can build a filter that cancels all frequencies above that level and hence obtain a low-filtered signal with the trend of the original one but without the noise.

With both cases, we can see that the main low frequencies lay below 20. Knowing so, several filters were considered but the chosen one has been a 10th order low pass filter with a Nyquist maximum frequency of $Fn = Fs/2$ where $Fs = 1/dt$ is the sampling frequency, a cutting frequency of $Fc = 1Hz$ and a normalized frequency of $wn = Fc/Fn$. Its Bode diagram is represented in Figure 4.11.



(a) *Window-Linear* case



(b) *Window-Exponential* case

Figure 4.10: Fast Fourier Transform (FFT) of mutual information for *Window* cases

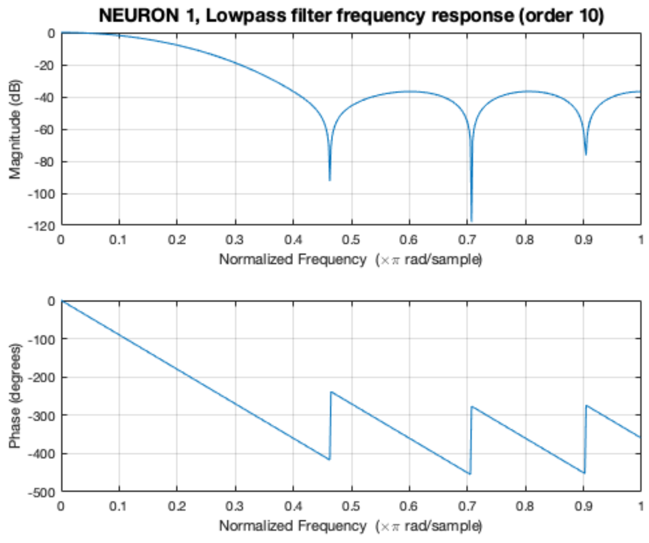
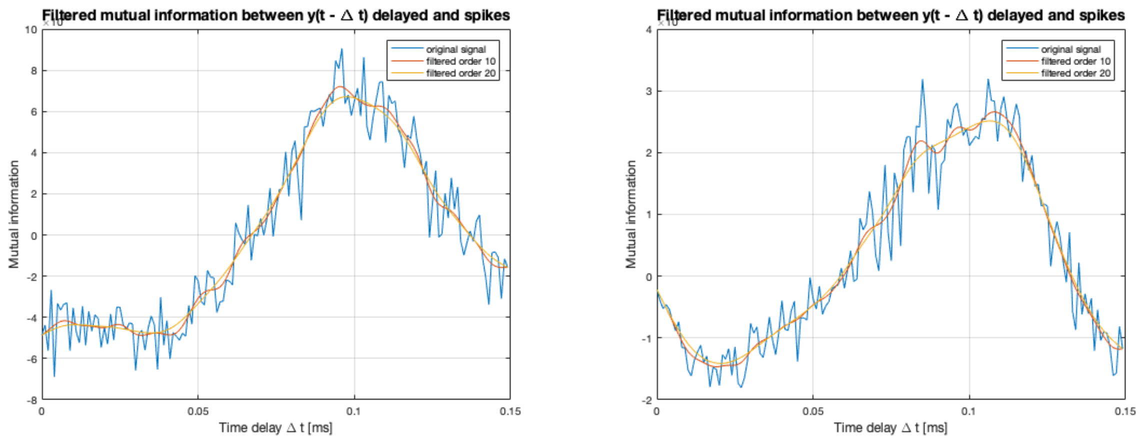


Figure 4.11: Bode diagram of the filter applied to Mutual Information for *Window* cases

As we can see, the filter leaves the low frequencies but cancels the high ones, hence it is definitely a low-pass one.

If we plot the resulting filtered mutual information with the filter that has just been described we obtain Figure 4.12. While the blue line is the original mutual information, the red one is the one corresponding to the chosen filter and it is the one that has been considered in order to find the position of the peak. The yellow line corresponds to another possible filter considered with the same cutting frequency but of higher order (order 20).



(a) *Window-Linear* case (b) *Window-Exponential* case

Figure 4.12: Filtered Mutual Information estimation for *Window* cases

After this step, both the time delay Δt and the optimal linear filter K are already estimated. We can see the obtained values along with the ground truth ones (marked with a *) in Table 4.1.

METRIC	CL	CE	WL	WE
Δt^*	100	100	100	100
Δt	104	114	105	91
K^*	0.3277	-1.0800	$\begin{bmatrix} 0.0649 \\ 0.0649 \\ 0.0649 \\ 0.0649 \\ 0.0649 \end{bmatrix}$	$\begin{bmatrix} -0.02155 \\ -0.02155 \\ -0.02155 \\ -0.02155 \\ -0.02155 \end{bmatrix}$
K	0.3179	-1.2024	$\begin{bmatrix} 0.0294 \\ 0.0462 \\ 0.0611 \\ 0.0788 \\ 0.0859 \end{bmatrix}$	$\begin{bmatrix} -0.4302 \\ -0.3010 \\ -0.2094 \\ -0.0942 \\ -0.0245 \end{bmatrix}$

Table 4.1: Δt and $\mathbf{K}$ estimations for simulation encoding approaches

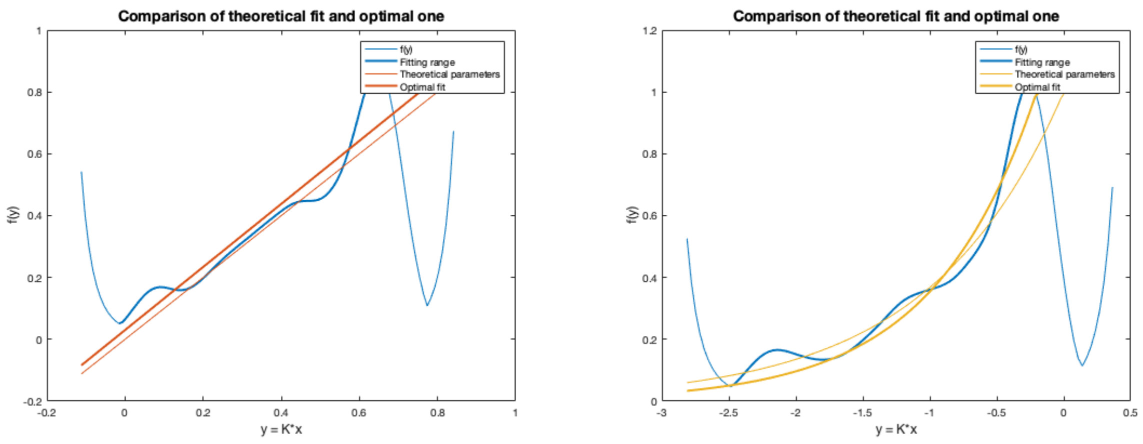
As we can see, the time delays are pretty much well estimated, specially in the linear cases. On the other hand, the K values were also well computed in the constant cases. If we consider the window cases, we see that the estimation doesn't provide with the same value in each of the terms but different ones and pretty far from the theoretical one.

The corresponding relative errors have been computed with respect to the ground truth. The results are again gathered in Table 4.3 within the evaluation section.

4.4.2 Nonlinear function $f(y)$ results

The final result is the resulting nonlinear function $f(y)$ along with its fittings according to either the linear or exponential approach.
The red colour has been associated to the linear approach and the yellow one to the exponential one. While the thinner line represents the theoretical nonlinear function that was used during the data building and hence it is considered the ground truth, the thicker one is the fitting that has been obtained as the optimal one, based on the obtained blue $f(y)$ function. One must remember that the fittings have been done considering only the monotonically increasing part of the resulting function as the edges confidence level is way much lower due to the kernel tails. This fitting range region is marked as a thicker blue line along the estimated function $f(y)$.

One can see in both pictures in Figure 5.8 that the fitting results obtained are quite good. Starting with the obtained function $f(y)$ (in blue), one can see that it is far away from the theoretical ground truth (coloured thin line). If one doesn't consider both the edges, then we can clearly see that it follows the right trend: linear in the left image and exponential in the right one. However, if we used it as part of our model, the error of the estimation would be huge.
Luckily, if we consider the optimal fittings in both cases, they aren't far away from the theoretical ground truth function, specially at the central part of the range. Therefore and based on what we see graphically, if we take this fitting as part of our predicted model one expects the results to be pretty good. That's why we are going to consider these fittings



(a) *Window-Linear* case (b) *Window-Exponential* case

Figure 4.13: Nonlinear function f(y) along with linear and exponential fittings for *Window* cases

as the resulting nonlinear function $f(y)$ estimation as they resemble way better the ground truth ones.

The obtained coefficient values when fitting $f(y)$ are displayed in Table 4.2.

METRIC	**CL**	**CE**	**WL**	**WE**
a*	1	1	1	1
a	1.06	1.40	1.02	1.30
b*	0	1	0	1
b	0.00	1.20	0.03	1.30
c*	-	0	-	0
c	-	0.00	-	0.00

Table 4.2: Fitting coefficients for simulation encoding approaches

Once again, some evaluation metrics between these variables have been computed and are gathered in the same Table 4.3. In this case, the relative error has been computed between variables.

With it, we have obtained the resulting time delay Δt, the linear filter K and the nonlinear function $f(y)$ estimation that constitute the prediction of our LNP encoding model. Now, one only has to assess how good are these predictions for all of the 4 cases considered and compare them, in order to find a clue on which approach to follow for real data.

4.5 Model Evaluation

In order to see how accurate the resulting LNP model is and hence how suitable the chosen pipeline is for applying it to the experimental data, one needs to compare how similar the estimated parameters are compared to the theoretical ones.

In order to do that, we are going to distinguish between the estimation of the different model parameters 1. the time delay 2. the linear filter K and 3. the nonlinear function $f(y)$. Furthermore, we are also going to check for the accuracy of the resulting output firing probability output with respect to the simulated one using the Time Rescaling Theorem.

4.5.1 Parameters evaluation

Time delta Δt

Starting with the time lag between the neural activity and the resulting kinematic vector, we are going to compare how similar are them.
In order to do that, we are just going to compute the **relative error** between the ground truth value Δt^* and the estimation Δt.

Definition 4.5.1. Relative error of variable var:

$$\epsilon_{var} = \frac{|var^* - var|}{var^*} \tag{4.23}$$

where var^* is the ground truth value and var its estimation.

Hence the error has been computed as:

$$\epsilon_{\Delta t} = \frac{|\Delta t^* - \Delta t|}{\Delta t^*} \tag{4.24}$$

Linear filter K

Following with the linear filter K, the same evaluation metric has been used: the relative error. However, one has to distinguish both the case in which $K = constant$ and the one in which $K = window$.

- On one hand, for the $K = constant$ approach, the error computation is straightforward:

$$\epsilon_K = \frac{|K^* - K|}{|K^*|} \tag{4.25}$$

- On the other hand, for the $K = window$ approach, one has to compute the relative error as well but using the vector's 2-norm.

  Definition 4.5.2. Let $\mathbf{x} = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$, then the 2-norm of $\mathbf{x}$ is:
  $||\mathbf{x}||_2 = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$

  Hence, the relative error is computed as:

$$\epsilon_{\mathbf{K}} = \frac{||\mathbf{K}^* - \mathbf{K}||_2}{||\mathbf{K}^*||_2} \tag{4.26}$$

Nonlinear function $f(y)$

In this case, one wants to measure how similar one function is to another one. It is quite difficult to obtain an evaluation metric that directly computed that and that's why another whole method was used.

Depending on which case we are dealing with, we have fit either a linear or an exponential function to the one estimated. Therefore we can compare the found parameters a and b (and c for the exponential case) with the theoretical ones a^* and b^* (and c^*), to see if they match[12].

Having said so, both values a and b for the linear case will be chosen as:

$$a, b = argmin_{a,b} \left(\frac{1}{m} \sum_{i=1}^{m} |f(y_i) - (a * y_i + b)| \right) \tag{4.27}$$

Similarly, for the exponential case we will choose those parameters that satisfy:

$$a, b, c = argmin_{a,b,c} \left(\frac{1}{m} \sum_{i=1}^{m} |f(y_i) - (a * exp(b * y_i) + c)| \right) \tag{4.28}$$

In order to do that, we have used the MATLAB function $g = fittype(expression)$ and then $fit = fit(y_{values}, f_y, g)$. With g we define the kind of function we want to fit (expression $= a * x + b$ for the linear case and $a * exp(b * x) + c$ for the exponential one), and with fit, we obtain the optimal parameters specified in *expression* that fit the function f_y along the specified y_{values}.

In order to consider the correctness of the parameters, the relative errors have been calculated as:

$$\epsilon_a = \frac{|a^* - a|}{|a^* + \epsilon|} \qquad \epsilon_b = \frac{|b^* - b|}{|b^* + \epsilon|} \qquad \epsilon_c = \frac{|c^* - c|}{|c^* + \epsilon|} \tag{4.29}$$

where a^*, b^* and c^* are the theoretical parameters used when building the data and the model. Notice we have add a $\epsilon = 1e - 16$ value to the denominator to avoid dividing by 0.

However, the coefficient errors aren't the ones we really want to analyse but the fitting one. That's why a fitting error has also been calculated based on the mean square error between the optimal fitting function and the theoretical one. These results are gathered in Table 4.3.

[12]Remember the linear form was $f(y) = a * y + b$ and the exponential one like $f(y) = a * exp(b * y) + c$

Metric	**CL**	**CE**	**WL**	**WE**
p_{firing} **rmse**	0.0771	0.0759	0.0733	0.0572
Δt **relative error**	0.0400	0.0300	0.0500	0.0900
K relative error	0.0297	0.0826	0.0423	0.1586
a relative error	0.0600	0.4000	0.0200	0.3000
b relative error	0.0000	0.2000	3e14	0.3000
c relative error	-	0.0000	-	0.0000
Fitting rmse	0.0291	0.0410	0.0490	0.0552

Table 4.3: Evaluation parameter metrics estimation for simulation encoding approaches

Observing the results, we can see that in general all four approaches lead to pretty good results.

On one hand, the firing probability rmse in all cases is lower than 10%. The window cases here lead to better results and it is quite obvious because as we know, the window act as a smoothing filter for the signal and hence the resulting firing probability is less noisy and hence easier to estimate.

On the other hand, the time delay and the linear filter K relative error are in general below 15% as well which is quite impressive.

Then, if we analyse the fitting parameters, we can see that the relative errors of the coefficients are quite elevated. We see in the window linear case that the coefficient b's error is huge; in part because the ground truth was 0 and a value of 0.03 was predicted. However, one must be aware that the coefficient errors aren't as significant as the fitting one because one can have two exponential functions that are very similar in a region but that their coefficients are far away from being near to each other. If we look again at the resulting fitting plots in Figure 5.8, we can see that even though the coefficient errors are huge, both the theoretical fitting and the estimated one are kind of close together in the y variable span we are considering.

Having said so, the fitting rmse obtained in all cases is lower than a 6% which is an impressive result.

4.5.2 Output evaluation

One can take the parameter errors and gain an insight on how good the model is. However, there is another method to see how accurate the estimation of the LNP model is and that in fact makes more sense to rely on: the idea is to compare the output of the model, that is, comparing the ground truth theoretical firing probability with its estimation, obtained after feeding the estimated model with the same input data as the one used for obtaining the ground truth. In order to do that, firstly one has to estimate the ground truth firing probability from the resulting spike train.

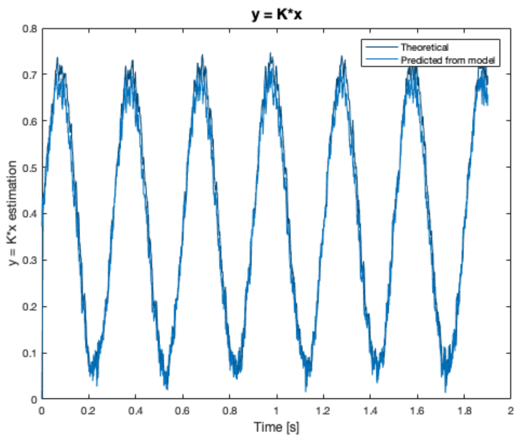
Ground truth firing probability

First of all and in order to evaluate the correctness of the output of the model, one must find the ground truth of the firing probability, that is the real firing probability that led to the generation of the spike train obtained as real data. This estimation has already been done when estimating the linear parameter K through STA (see section 4.3.1 Firing probability estimation) so the procedure followed is the same. Basically, a Gaussian Kernel was convoluted with each of the different resulting spikes and the results were added and scaled so that they represented a distribution.

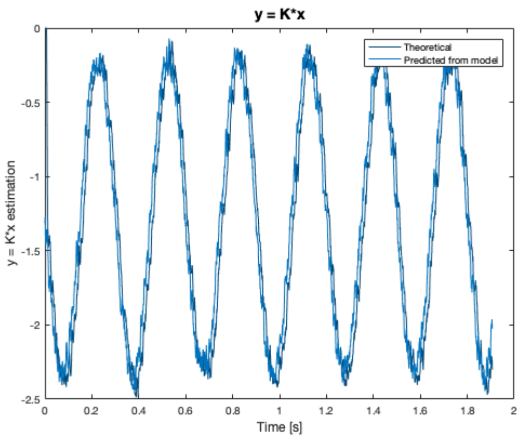
Output generation

Besides having the ground truth firing probability, one needs the resulting firing probability predicted from the model. It should be obtained by feeding the estimated model with the original kinematic vector $x(t)$. With it, we will obtain a p_{firing} estimation that will be compared to the ground truth one.

The first partial output of the LNP model built is the resulting feature space signal $y(t)$. It is represented in Figure 4.14. Remember that the two missing cases figures are gathered in 8.2.1 Additional Figures and Tables.



(a) *Window-Linear* case

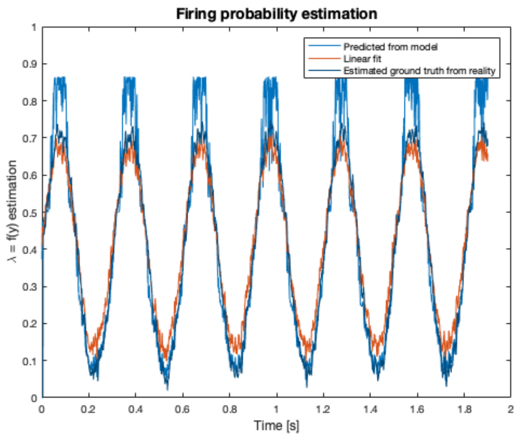


(b) *Window-Exponential* case

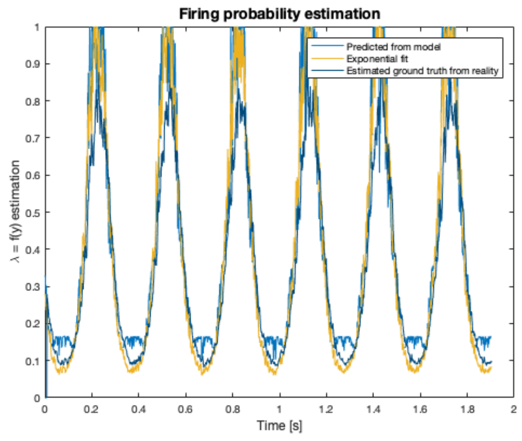
Figure 4.14: Estimated feature space vector $y(t)$ for *Window* cases

With the estimation, we have also plotted the real $y(t)$ values obtained from projecting the original kinematic vector $x(t)$ with the real $\mathbf{K}$. As it can be seen, the estimations are almost perfect, which is a good sign.

The next step is to apply the nonlinear function $f(y)$ to $y(t)$. We have plotted the results of $f(y)$ along with the ones obtained with the corresponding fitting. The results can be see in Figure 4.15.



(a) *Window-Linear* case



(b) *Window-Exponential* case

Figure 4.15: Estimated p_{firing} for *Window* cases

As one can see, for the *Window-Linear* case, the estimated function brought to worse results than the linear fitting ones. As we can see in the plot, the red line approaches more accurately the ground truth firing probability.
On the other hand, for the *Window-Exponential* case, both the direct estimatino in blue and the exponential fitting lead to similar results.

Once we have the ground truth firing probability directly obtained from the real data spike train, we need an appropriate goodness-of-fit measure in order to compare it with the one predicted feeding our LNP model with the kinematic vector.

For this purpose, one must review the Time Rescaling Theorem.

Time Rescaling Theorem

The time rescaling theorem is useful to assess goodness-of-fit between the output of the statistical model that has just been fitted and the spike train from real data. Let's first take a look to the formalities in order to understand how the theorem will be used in order to evaluate our model.

According to Barbieri et al. (2002) it states that '*Any inhomogeneous Poisson process may be rescaled or transformed into an homogeneous Poisson process with a unit rate*'[13]. More generally, it states that '*Any point process with an integrable conditional intensity function may be transformed into a Poisson process with unit rate*'.
The mathematical definition of the theorem for a point process spanning from time 0 to time T is as follows (find proof at Appendix 8.1).

Theorem 4.5.3. *Let* $0 < u_1 < u_2 <, ..., < u_n \leq T$ *be a realisation from a point process with a conditional intensity function* $\lambda(t|H_t)$ *satisfying* $0 < \lambda(t|H_t)$ *for all* $t \in (0, T]$*. Define the*

[13]Remember that a Poisson process being inhomogeneous refers to having a time dependent rate $\lambda(t)$; whereas homogeneous refers to having it constant λ

transformation:

$$\Lambda(u_k) = \int_0^{u_k} \lambda(u|H_u)du \tag{4.30}$$

for $k = 1, ..., n$*, and assume* $\Lambda(t) < \infty$ *with probability one for all* $t \in (0, T]$*.*
Then the $\Lambda(u_k)$*'s are a Poisson process with unit rate.*

For our case, one has to identify the realisations u_k as the different times at which a spike was generated, being n the total amount of spikes along the $(0, T]$ time span. Also, the rate $\lambda(t|H_t)$ corresponds to the firing probability rate at each time t given the previous history of the process $H_t = \{0 < u_1 < ... < u_{N(t)} < t\}$ where $N(t)$ is the amount of occurred spikes in the interval (0, t]. In our case, this rate can be identified as the Poisson rate $\lambda(t)$.

Hence, now one has a well-known transformation that can convert any point process realisation into a Poisson process with unit rate.

* * *

In order to assess the goodness-of-fit measure, one needs to build the following derived variables:

$$\begin{aligned} \tau_k &= \Lambda(u_k) - \Lambda(u_{k-1}) \quad for\ k = 2,\ ...,\ n \\ \tau_T &= \int_{u_n}^{T} \lambda(u|H_u)du \end{aligned} \tag{4.31}$$

As proved in Appendix 8.1, if the model is correct and hence meets the specifications required for Theorem 4.5.3, these new variables τ_k's are independent identically distributed exponential random variables with mean equal to 1.
Furthermore, some new variables must be built:

$$z_k = 1 - \exp(-\tau_k) \quad for\ k = 2,\ ...,\ n,\ T \tag{4.32}$$

It is direct to see that, if the τ_k fulfils the specifications given (i.i.d. exponential random variable with mean equal to 1) and thus the model is correct, z_k's are independent uniform variables between (0, 1].

This statement is the one that will be used in order to evaluate the correctness of the LNP predicted model.
Therefore, in order to check whether these z_k's are independent uniform variables, either the Kolmogorov-Smirnov test or the Quantile-Quantile-plot can be used.

Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov Test is a nonparametric test that compares one data sample with a reference probability distribution. It basically answers the probability that the tested sample was drawn from the hypothesis distribution. It is gonna be used to see whether the resulting z_k's belong to a uniform distribution, as stated by the correctness hypothesis.
The steps one should follow are:

1. Order z_k's in ascending order.

2. Compute the cumulative distribution function of the uniform density as:

$$b_k = \frac{k - 1/2}{n} \quad for k = 1,\ ...,\ n \tag{4.33}$$

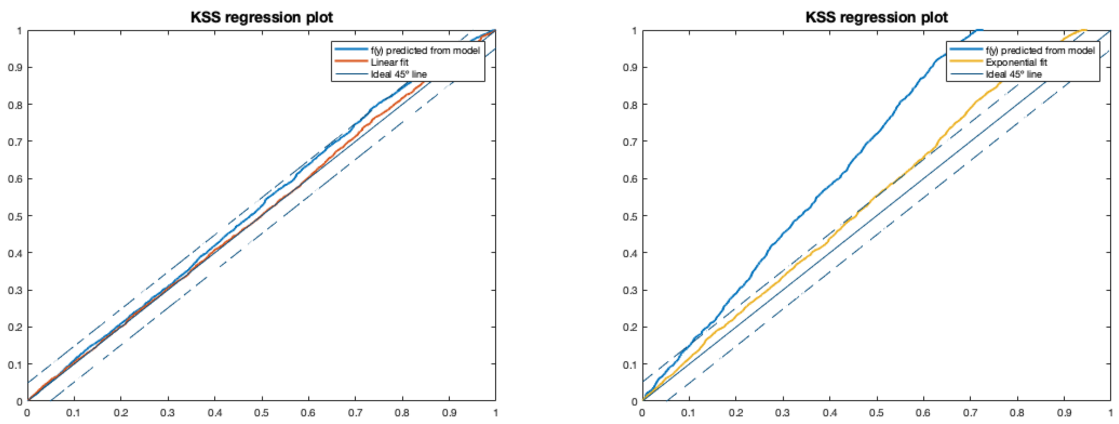
3. Plot b_k's against z_k's. Since we are plotting the cumulative distribution function, if z_k's belong to a uniform distribution and hence the model is correct, the plot should lie on a 45-degree line.

All these steps have been gathered in the function $computeKSStats(spikeTrain, lambda, opt)$ (see section 8.3 Matlab codes for the code).

Metric	**CL**	**CE**	**WL**	**WE**
KSS f(y) prediction	1.6157	1.4074	1.1472	3.9874
KSS linear fit	1.2820	-	0.6145	-
KSS exponential fit	-	2.6136	-	2.2751

Table 4.4: Output evaluation KSS metrics for simulation encoding approaches

If we plot the results obtained for both the $Window-Linear$ and $Window-Exponential$ cases, we obtain the results in Figure 4.16. As we can see, in both cases the fitting leads to better results than the obtained function $f(y)$. Moreover, it seems that the linear case led to better results than the exponential one as the corresponding KSS line approaches more to the ideal 45º line.



(a) *Window-Linear* case

(b) *Window-Exponential* case

Figure 4.16: KSS plot for *Window* cases

In addition, not only the KSS plot has been built but also its metric. One has to know that the lowest the KSS metric the best. The obtained results for each of the four cases are gathered in Table 4.4.

For the KSStats, the lower the better. Therefore, in most of the cases (except the *Constant-Exponential* one), the corresponding fitting enhances the performance of the direct $f(y)$ prediction, hence when it comes to predict the nonlinear function in real data, the same procedure will be followed and it will be these fittings the ones that will be considered and

compared.
Moreover, the linear approaches give better results than the exponential ones when comparing within the constant cases on one hand and the window cases on the other. The explanation for it may be that the linear cases are easier to estimate so its results are better. Similarly, if we now focus within the linear cases on one hand and within the exponential cases on the other, we see that the window approaches lead to better results than the constant ones, with metrics lower than the corresponding constant cases ones. This result could be expected as the window approaches consider history and hence they have more information when it comes to correctly predict the resulting firing rate.

Having said so, one can state that the models that will be considered in real data are in first place the *Window-Linear* one, as it is the one that led to the lowest KS Statistic and also the *Window-Exponential* one as within the exponential cases it is the best one and being it more similar to the window-linear one makes it easier to compare between them.
In addition, considering these two cases, one is going to see the ***Window-Linear*** case as the **baseline model** and the ***Window-Exponential*** as one **nonlinear model proposal**. With it, one expects to determine whether including the non linearity of the function $f(y)$ gives to better results or not in the building of an LNP neural encoding model.

Quantile-Quantile Plot

Another option that could be used is to use the quantile-quantile plot. It is a similar graphical method suitable to compare whether two probability distributions are comparable or not. The idea is to plot the quantiles of both distributions one against the other.
In our case, we would need to plot the quantiles of the uniform distribution samples b_k against the ones from our estimations z_k's. If the uniform distribution assumption for z_k is fulfilled, the plot should also lie in a 45-degree line.

This plot has not been performed as the results obtained from the same KS are already significant.

All in all, and after having analysed the performance of the four different approaches, it has been decided that the both the *Window-Linear* and *Window-Exponential* cases are the ones that suit better for describing the encoding procedure that exists between the kinematic vector and its associated neuron activity. They will definitely be the approaches followed for real data in the next chapter.

Chapter 5

Experimental Encoding

Once having ensured that the estimation procedure or pipeline is the appropriate one for the defined problem, it will be applied to real data.

This data was experimentally collected in the laboratory by studying the behaviour of some rats when given a stimulus. Their brain responses were simultaneously recorded along with the stimulus that they were given. The steps that constituted the pipeline in this section have been:

1. Simultaneously collecting the stimulus signal and the corresponding brain response from rats.
2. Given these two signals and following the pipeline already checked with simulated data, predicting a reliable encoding model.

A more detailed explanation of how the data was collected and what were the steps followed constituting the whole pipeline will follow, as well as an exhaustive exposition of the approach considered.

5.1 Data collection

Both the motor input and output data collection have been done in the biological laboratories of The Hong Kong University of Science and Technology (HKUST). The idea behind the collection of data is to train some rats to do some kind of task and simultaneously record their neural activity, that is, the spiking pattern in several targeted neurons. The inclusion of animals within the data collection procedure was approved by the Animal Ethics Committee at the same University HKUST.

In this research case, the task selected for the rats to learn is a two lever discrimination task. It consists in pressing either the high lever or the low one for at least 500 ms whenever a high (10 kHz) or low frequency (1.5 KHz) sound was emitted respectively. The fact of requiring to hold the lever for at least 500 ms is set so that the pressing of any of them isn't an accident. Whenever the task is correctly achieved, a container is filled with water and thus the rats can drink from it. An audio feedback with the same feedback is produced to denote success. The fact of drinking water has nothing to do with the stimulus signal but it is just an incentive for the rats to learn the task properly. Whenever there is an early releasing, wrong pressing or no pressing, it doesn't lead to the water reward.

This being said, the data that we are going to use has been provided by the same laboratory and has been recorded for rat number 11 during the 12 of December of 2018. Several trials were performed with the rat each of which lasted 6 s. Also, the inter-trial period lasted for a random amount of time in between 3 s and 6 s.

Considering the task, two different groups of data are recorded.
The first one is referred as the manual data. It contains a very simple binary stimulus representing whether any of the levers is pressed or not and the resulting neural spike activity. Consequently, the input stimulus is a non continuous binary signal with zeros and ones.
On the other hand, the second type of data is referred as the **brain data**. In this case, the stimulus signal represents the whole movement of the rat lifting up his hand in order

to press the high lever and also the movement of lowering it when pressing the bottom lever. Therefore, the resulting stimulus signal is a continuous representation of his hand's movement. This is the data that will be used for the encoding model estimation as it is the most similar to the one considered during the whole simulation process: not only it is continuous but it also contains a periodicity according to the upward and downward movement of the rat's hand.

However, some data characteristics differ a little bit from the simulated data ones.
On one hand, regarding the stimulus data, instead of a one dimensional variable along time, we have a **5 dimensional kinematics vector** that also evolves in time. That means, at each time step we have a vector of five elements that correspond to position x and y of rat's hand, its velocity x and y and a bias. This last term is needed because in reality neurons still fire randomly without being activated by any stimulus, hence, there is a background firing that must be taken into account. This fifth dimension has this responsibility. The complexity added by the different dimensions makes the encoding model a more difficult one to estimate but also a more accurate one when built, as it considers not only the position variables but also their first order derivatives. This fact implies that some of the steps followed with the simulation data will have to be slightly modified. Further details on how to deal with it are going to be explained in this section.
On the other hand, for this unique stimulus, 20 resulting train spikes have been recorded corresponding to 20 different neurons that want to be studied. Hence, the idea is to somehow identify which of these **20 neurons** are related with the stimulus shown and to develop a unique encoding model for each of them. The criterion for deciding which of the neurons are related with the stimulus and hence which neurons are going to be fully analysed is going to be detailed in the task-related neuron selection within the 5.2.2 Time delay Δt estimation: Mutual Information subsection.

5.1.1 Preprocessing of data

Furthermore, some preprocessing has to be done to this data.
One has to be aware that the process of the rat pressing the buttons is not ideal. Sometimes the rat performs the task as needed, waiting for the alarm to sound, pressing or releasing according to the sound and staying in that position for the right amount of time until achievement, but sometimes it doesn't. That's why one should only consider those occurrences in which the whole task was correctly done. Every attempt of achieving the task will be referred as event.

To do so, another related variable was given: the so called ***labelled behaviours***. As its name says, this variable contains the label of the subactivity the rat was doing at each time within each event. The possible labels are '*HighStart*' for when a high frequency sound is played, '*PressHigh*' when the rat starts pressing the top button, '*HighReward*' when the rat presses the top button for the right amount of time and achieves the task and '*HighFail*' whenever the rat doesn't complete the task properly. The same variables with the word 'Low' instead of 'High' represent the equivalent when a low frequency sound is played and the rat presses the bottom button instead. With it, only the triads 'HighStart' - 'PressHigh' - 'HighReward' and 'LowStart' - 'PressLow' - 'LowReward' are considered as success events. So, in order to properly encode an encoding model that represents the stimulus neural activity, only these cases should be considered. Having said so, we are going to match the behaviour labels with the stimulus input signal in time and we are going to identify the success triads we have just mentioned.
A whole new signal will be built from them and will be the one used as input stimulus to our encoding model. In order to do so, we are going to consider a window that expands $\pm 500\ ms$ before and after each label belonging to a success event, and we are going to concatenate them all together in time so that at the end the resulting stimulus signal will be built from all the success blocks joined together. Doing it this way, it will properly represent the repeatedly achievement of the task object of study. The corresponding spike train signals

will be cropped and joined accordingly. Recall that, if two labels are located at a distance in time smaller than two times the time span of $500ms$, when performing the window approach, there will be a segment of the signal that will be repeated twice. This would make no sense as it won't represent the real signal properly. That's why, if two windows overlap we are going to consider the repeated region only once.

The stimulus data we begin with is a 5-kinematic vector with a total of 140000 samples (each corresponding to 1 ms so, 140 s in total). After the preprocessing we have just said, the length of it will still be way longer than the stimulus signal we used in the simulation data (of only a few seconds). That's why one will still have to cut the already built signal so that it is shorter. The criterion we have followed for cutting it has been to include at least 10 success trials along time, so that we can have enough information to proper analyse the signals. Considering that, in case no overlapping occurred, each of the success trials is made of 3 $\pm$500 ms windows, the resulting signal should be of length $3 * (2 * 500) * 10 = 30000$ ms. If overlapping indeed occurred, we would still include at least 10 success trials if not more, so it still works.

To sum up, the resulting data we'll have for the modelling of the encoding model is a 30000 sample 5-dimensional kinematic input signal evolving in time containing only the time spans corresponding to the success trials and properly cut so that at least 10 of them are present. The identified task-related spike trains out of the possible 20 given are similarly cropped and joined together. The Figure 5.1 shows the resulting 2nd term of the 5-dimensional success $x(t)$ for neuron 1 for the first 2 seconds. It is kind of periodical so it resembles the built signal in the simulation section.

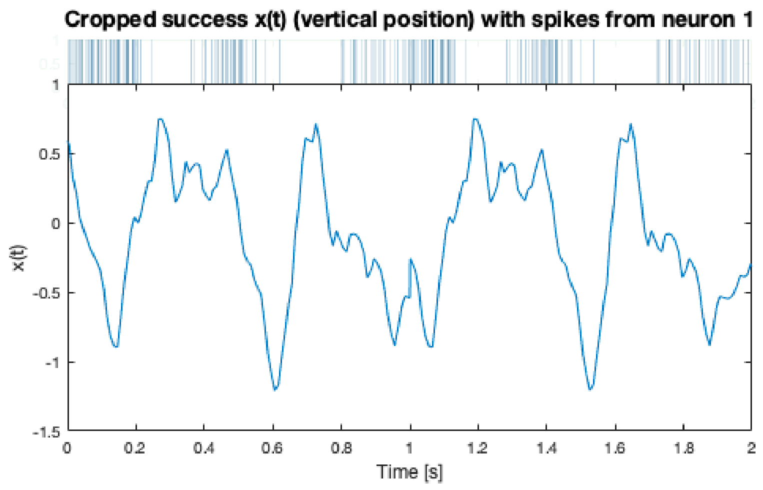


Figure 5.1: Built success $x(t)$ along with the spikes. *Note: only 2 s have been plotted for clarifying purposes. The whole signal expands for 30 s instead.*

5.2 Model parameter estimation

When it comes to estimate the proper encoding model, it has already been said that both the window approaches will be considered. The *Window-Exponential* one will be the approach that will give us the final estimation of the model we are looking for, while the *Window-Linear* approach will be used as baseline model. However, the dimension to which we apply the window will not be the same as the one used in simulation data and this deserves to be explained in detail.

When the window approach has been considered in the simulation data, the aim was to provide the model with some extra information rather than just one sample. This extra information was represented by some history, that is, not only the present sample was taken into account but also some past ones. Hence, the window was applied in the time dimension so it is of size $(1\,x\,m)$ where the first term is the kinematic dimension and the second one is the length of the window along time.
Nevertheless, with real data one already has 5 different dimensions for each time step and that's why the window will expand along these 5 dimensions instead. It is another way to

include some extra information to the encoding model and hence the history approach is no longer needed. This time, the window will be of size ($m\,x\,1$) with $m = 5$ (see Figure 5.2).

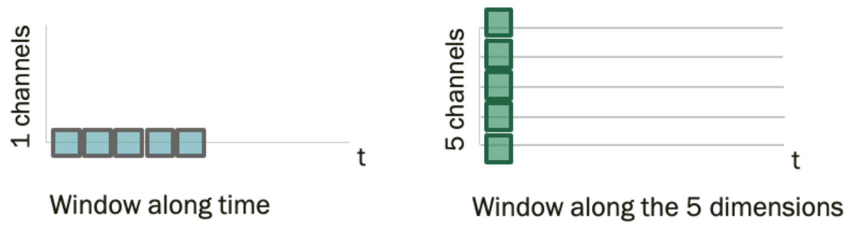


Figure 5.2: Window approach differences between Simulated and Experimental data

5.2.1 Linear filter K estimation: STA

Similarly as what has been done with the simulation data, the first parameter to estimate is the linear parameter K. Recall that this estimation will be done along with the time delay one; however and following the same as before, the procedure in here will be explained as if we already knew this delay and all the signals were matched in time.

This step presents some differences with the corresponding one in simulation data. As we have already mentioned, the real data is 5 dimensional instead and hence we need to expand the calculations to this new multidimensional space. To do so, it is quite obvious to see that what we need is a 5-dimensional K parameter so that when multiplied by the original kinematic vector, it projects it to a one dimensional feature space.

Recall that the method to properly find the optimal K parameter was the Spike Triggered Average (STA). Some adjustments must be made in order to consider the multidimensional characteristics.
The formula that was used in simulation data was:

$$\mathbf{K} = \left(X^\top\, X\right)^{-1} \left(X^\top\, \Lambda\right) \tag{5.1}$$

However, in order to use that, one would need the firing probability Λ which is unknown for the moment when given the real data. Here we would have two options, the first one would be estimating it and using the equation 5.1 or computing the STA in a whole different way. We are going to choose the second option as the estimation of Λ would bring some error to the estimation that doesn't want to be propagated along the computations.

In order to explain this second option, one must remember the equation of the definition of the STA.

$$STA = \frac{1}{n_{spk}} \sum_{i=1}^{T} n_i \mathbf{x_i} \tag{5.2}$$

Remember that the n_i were the number of spikes in each time bin. In our case we either have 0 or 1 spike in each bin, hence n_i would be either 0 or 1 as well. Consequently, the summation itself would only contain those cases in which we had a spike ($n_i = 1$). Hence, the summation can be translated into the summation of each one of the terms of the kinematic vector samples in which a spike was generated. If we translate this into matrix form we have that the STA $\mathbf{K}$ estimation can be computed as:

$$\mathbf{K} = \left(X^\top X\right)^{-1} sum\left(X_{spk}\right) \tag{5.3}$$

Here, the first term $\left(X^\top X\right)^{-1}$ again represents the normalisation term $1/n$ and the second term $sum\left(X_{spk}\right)$ represents the summation of only those samples that had a spike.

In order to use this method, first, we need the kinematic vector $\mathbf{X}$. Let it be the five dimensional stimulus signal expanding along time from 0 to T, being T the number of time samples.

$$\mathbf{X} = \begin{bmatrix} x_1(t) \\ \vdots \\ x_5(t) \end{bmatrix} = \begin{bmatrix} x_1^0 & x_1^1 & \cdots & x_1^T \\ \vdots & \vdots & \ddots & \vdots \\ x_5^0 & x_5^1 & \cdots & x_5^T \end{bmatrix} \tag{5.4}$$

Besides, one also needs the conditioned matrix $\mathbf{X_{spk}}$ with those samples containing the spikes. If $\{0 \leq u_1 \leq ... \leq u_n \leq T\}$ are the different times at which a spike was generated (with n spikes in total), we can define $\mathbf{X_{spk}}$ as:

$$\mathbf{X_{spk}} = \begin{bmatrix} x_1^{u_1} & x_1^{u_2} & \cdots & x_1^{u_n} \\ \vdots & \vdots & \ddots & \vdots \\ x_5^{u_1} & x_5^{u_2} & \cdots & x_5^{u_n} \end{bmatrix} \tag{5.5}$$

Therefore, the summation will be:

$$sum(\mathbf{X_{spk}}) = \begin{bmatrix} \sum_{i=1}^{n} x_1^{u_i} \\ \vdots \\ \sum_{i=1}^{n} x_5^{u_i} \end{bmatrix}$$

Then, one only has to use the STA following equation 5.3 to compute the optimal $\mathbf{K}$.

Using this estimation, we obtain the desired and optimal 5-dimensional $\mathbf{K}$ that, when multiplied by $\mathbf{X}$, leads to a 1-dimensional feature space that evolves along time.

$$\mathbf{y} = \mathbf{K}^T\,\mathbf{X} \tag{5.6}$$

5.2.2 Time delay Δt estimation: Mutual information

The next step we need to follow is to compute the time delay existing between the feature space vector $y(t)$ and the spike train.

For each of the 20 neurons, the time delay has been estimated following the same steps as the ones defined with the simulation data: computing the mutual information between the spikes and different shifted stimulus guesses, low-pass filtering it and finding its maximum value.

However, one must take into account that in order to do that the same way, we need to select one of the terms among the 5 different channels the input stimulus has so that we have a one dimensional signal instead. It is true that all of them five will be linked and their oscillations will be related but the final taken decision is to take the y position. The reason why we considered it instead of the x term, the velocities or the bias is because the vertical direction is the one that is more sensible to the movement of the rat's hand pressing both levers as it moves vertically. Hence the y term will be the reference one from this point on.

Period estimation

Recall that in order to properly estimate the time delay, the period of the stimulus signal must be known[1]. In contrast with the simulation data case, the period of the real data is unknown and hence must be estimated. In order to do so and according to what has been said, one will consider the vertical position of the original success kinematic vector to properly characterise the oscillation period.

So as to proper calculate it, one should try to remove the noise that is implicit in the original signal. The easiest way to do so is low filtering the signal to eliminate the characteristic high frequency oscillations of noise and just leaving the low frequency oscillations that characterise the vertical movement.

Considering that the sampling interval is the time step $Ts = 1 * 10^{-3}\ s$, we can compute the sampling frequency as $Fs = 1/Ts = 1/dt = 1000\ Hz$ and the corresponding Nyquist frequency as $Fn = Fs/2 = 500\ Hz$. One must ensure that the chosen cutoff frequency Fc[2] of the filter we are designing should be lower than the Nyquist one in order to avoid aliasing $Fc < Fn$.

By observation it is easy to notice that the signal is full of different high frequency oscillations that are inconvenient when it comes to calculate the period and hence we only want

[1]When finding the possible time delays, we only have to make guesses that are lower than 0.5 times the period divided by the time step, as specified by equation 4.21.

[2]The cutoff frequency is the one from which the low pass filter will attenuate the signal.

to keep the general trend that is behind them. That's why the cutoff frequency has been chosen to be of $1\,Hz$, which is indeed way lower than the Nyquist frequency as needed. Once the parameters have been defined, the filter itself is computed with the MATLAB function $fir1(n, Wn)$ which designs a window-based FIR filter of order n. However, the function works with normalised frequencies instead that must lay between 0 and 1 and hence, the normalised cutoff frequency should be provided instead. Since Fn is the maximum possible value the cutoff frequency could have, its normalisation procedure will be computed as follows: $Wn = Fc/Fn$ so that it led to 1 whenever $Fc = Fn$, its maximum value.
Having said so, the order n is yet to be decided. The impact this parameter has in the filter is just to specify how fast the trend of the magnitude in the frequency domain drops to 0. The highest the value, the faster it falls to 0. Three different values were considered ($n = 100$, $n = 300$, $n = 500$) and based on observation, the latest case was the one giving more reliable results (see Figure 5.3).

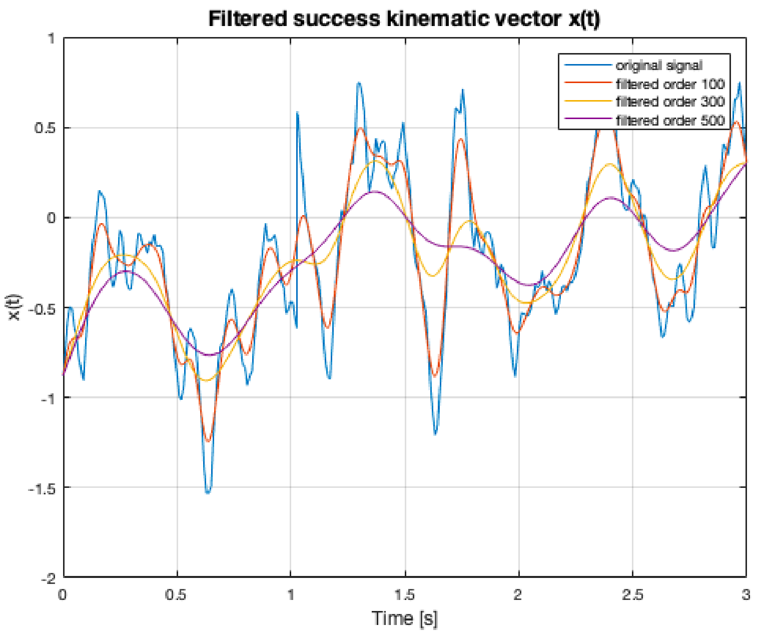


Figure 5.3: Filtered success $x(t)$ vertical position term with several low-pass filters

Having obtained the corresponding filter and having applied it to the original vertical position term of the success kinematic vector $x(t)$, the inter-peak times of it were found and averaged in order to find the period of it. A final period of $\mathbf{T} = \mathbf{0.5333\,s}$ was found.

* * *

Once having obtained the period of the signal, the mutual information plot is obtained for all the possible time lag guesses and the optimal parameters K and Δt are found at its maximum.

One must bear in mind that from this point on, the nonlinear function estimation won't be done for all the 20 neurons. Instead, and based on their optimal mutual information value, we are going to select those that are related to the lever pressing task we are dealing with.

Task-related neuron selection:

Once we have found the optimal time delay applied to each of the neurons, one must select which of these neurons are proper candidates for being related with the task the rat is performing.

To do so, what has been done is plotting the different neurons in descending order according to their maximum mutual information. With it, only those first neurons that have a considerable high mutual information value will be assumed to be related with the task and will be the ones that will be used in further computations. Hence, those that lay in the leftest part of the plot, and more specifically, those whose mutual information lays before the trend drops significantly, will be the ones considered.
This discrimination is done because it makes no sense to estimate an encoding model for a neuron that has nothing to do with the task object of study.

All in all, the following steps will be done for each of the task-related neurons that will have been selected previously so one different model will be obtained for each.

5.2.3 Nonlinear function $f(y)$ Estimation: Bayes Theorem

Following with the pipeline, one must perform the estimation of the nonlinear function $f(y)$. In this case, the procedure is the same as the one followed with the simulation data because both the starting signal and the resulting one are of the same form as before: the former is a one dimensional feature space signal evolving along time and the latter corresponds to the firing rate.

Following the same procedure as the one for simulation data, kernel density estimators are used to estimate two of the probability density function terms present in the Bayes rule equation 4.22, which are $p(Kx \mid spk)$ and $p(Kx)$. They will be calculated again with the defined function 4.3.4 $kernel\_smoothing(var)$.
The remaining $p(spk)$ term is equally calculated as the number of spikes divided by the number of time samples.

Again, one will try to fit both a linear and an exponential function to the obtained $f(y)$ estimation. As we have seen in simulation, one expects them to bring better results than the actual direct $f(y)$ estimation. Similarly as before, not the whole y range will be used for the estimation. In this case, the lower limit will be the position of the minimum value in the first third of the nonlinear function estimation and the upper limit will be the position of the maximum in the second half.

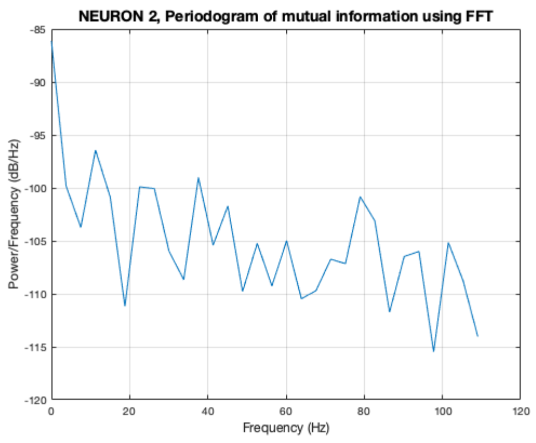
5.3 Model Results

Once the whole procedure has been applied to the different 20 neurons we have, we are going to present the results obtained in each intermediate step.
The corresponding plots will only be displayed for neurons number 2 and 13. The rest of the plots corresponding to the remaining neurons will be displayed in Chapter 8.2.2 Experi-

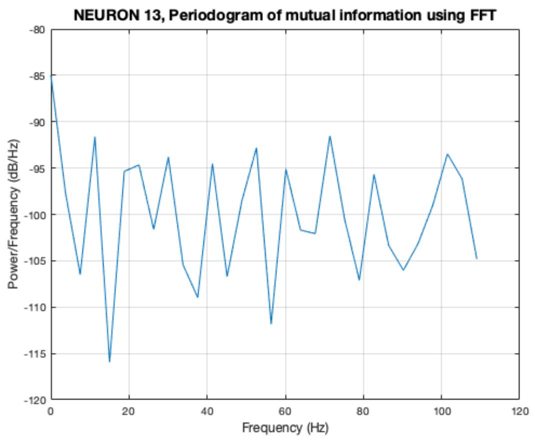
mental Encoding additional Figures and Tables. Also, several tables containing some of the estimations will be gathered in this section as well.

5.3.1 Linear filter K and time delay Δt results

One has computed the estimation of the linear filter K for each of the neurons along with the time delay Δt. The resulting mutual information was again noisy and had to be filtered. The filter used has been the same as the one used in simulation data for all the neurons (see Figure 4.11) as the resulting Fast Fourier Transform of the signal was quite similar for all the cases (see Figure 5.4 for neurons 2 and 13).



(a) Neuron 2

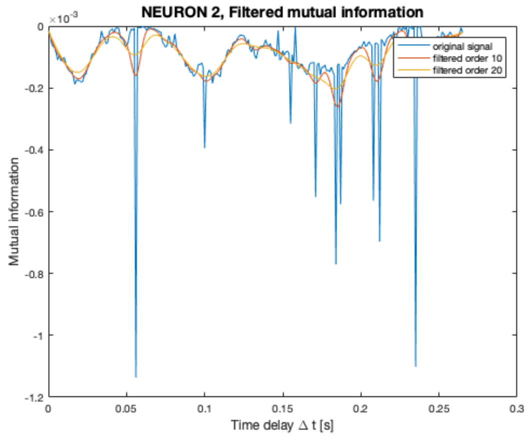


(b) Neuron 13

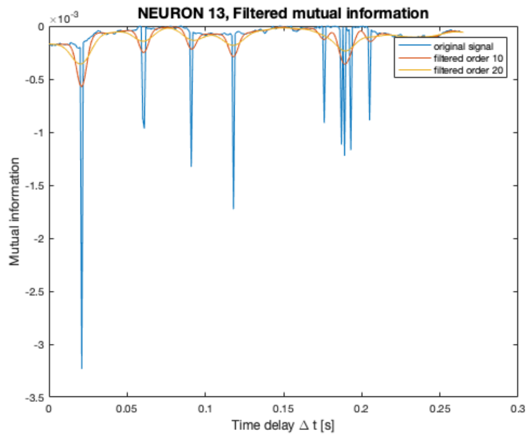
Figure 5.4: FFT plot for neuron 2 and 13

The resulting filtered Mutual Information graphics for both neurons 2 and 13 are represented in Figure 5.5. As we can see, the general trend of it is obtained and it is from this filtered plot that one will find its maximum value.

The resulting estimations of both the time delay Δt and the linear filter K are gathered in Table 8.1. If we only represent the ones from neuron 2 and 13 we obtain the following table.



(a) Neuron 2



(b) Neuron 13

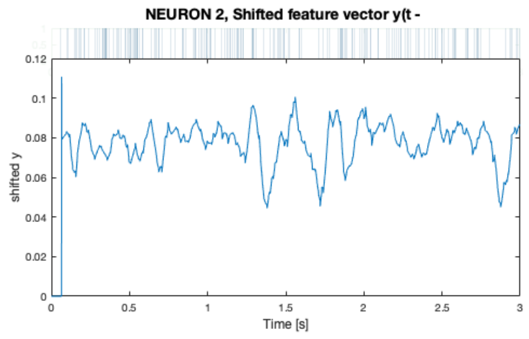
Figure 5.5: Filtered mutual information for neuron 2 and 13

METRIC	**n2**	**n13**
Δt	66	151
K	$\begin{bmatrix} -0.0469 \\ -0.0046 \\ 0.1750 \\ -0.0115 \\ 0.0814 \end{bmatrix}$	$\begin{bmatrix} 0.0004 \\ 0.0059 \\ 0.2844 \\ 0.0061 \\ 0.0731 \end{bmatrix}$

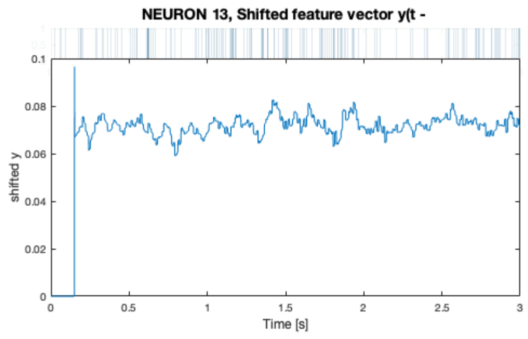
Table 5.1: Δt and **K** estimation for neuron 2 and neuron 13

In this case, we can't compute any error as we don't have any ground truth regarding any of the two parameters that we are considering.

Knowing the optimal time delay and the linear filter one can now shift the signals so that they match in time. If we plot the spikes along with the shifted feature space signal resulting from $y(t) = K * x(t)$, then we obtain the plots represented in Figure 5.6.



(a) Neuron 2



(b) Neuron 13

Figure 5.6: Feature space signal $y(t)$ along with spikes, matched in time for neuron 2 and 13

Task-related neuron selection

At this point and after having estimated both parameters for all the 20 neurons, they have been placed in optimal mutual information descending order so that we could choose the neurons that were task related.

Hence, according to the Figure 5.7, neurons from 15 to 6 (15, 2, 5, 16, 13, 14, 11, 19, 8, 6) will be considered as related with the lever task and hence will be the only ones for which an encoding model will be estimated. Hence, we have ten different related neurons that will be analysed.

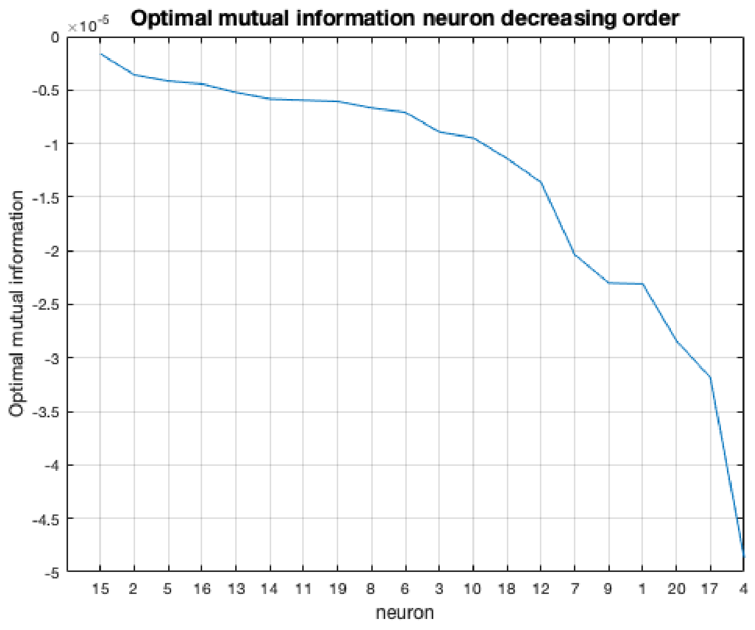
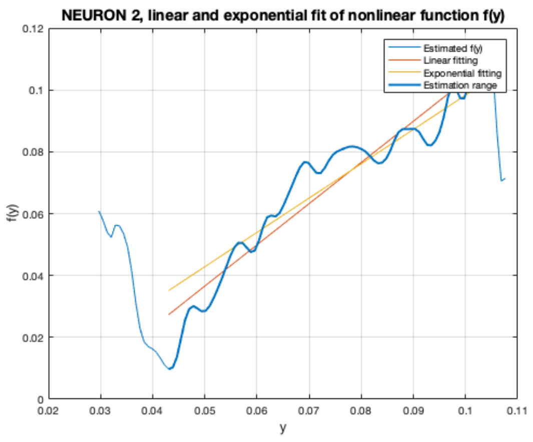


Figure 5.7: Decreasingly ordered neurons according to optimal mutual information value

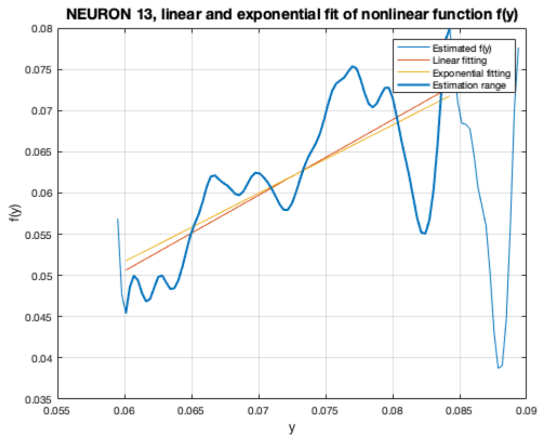
5.3.2 Nonlinear function $f(y)$ result

Finally, one wants to obtain the nonlinear function $f(y)$ along with both the linear and the exponential fittings.
Matching the simulation encoding section, the red colour has been associated to the linear approach and the yellow one to the exponential one. Again, the thinner blue line represents the direct estimated nonlinear function $f(y)$ while the thicker region of it marks the range of values that have been considered for the fitting.

In both cases, the linear and exponential approaches don't differ that much and hence we expect them to provide similar results in terms of evaluation metrics. This is what is going to be discussed in the following section.



(a) Neuron 2



(b) Neuron 13

Figure 5.8: Nonlinear function f(y) along with linear and exponential fittings for neuron 2 and 13

5.4 Model Evaluation

Now that we already have developed a model for each of the neurons that were related with the task of pressing the high and low levers, one needs to evaluate how accurate the encoding models are for each of them. Recall that the following steps will be done for each of the task-related neurons but the procedure will be described as if only we had only one individual case.

The idea behind the evaluation method would be: given the original kinematic vector, if the neuron predicted LNP model was applied leading to an estimated firing probability, how similar would it be to the one coming from the real data spike train.

As one may have already realised, in this case we have no ground truth in terms of the parameters of the model and hence no parameter evaluation can be done. However, one can still measure the accuracy of the output of the model when fed with the data compared to a ground truth firing probability. We will see that this ground truth p_{firing} isn't directly obtained from data but it must be estimated.

5.4.1 Output evaluation

Ground truth firing probability

First of all and in order to evaluate the model, one must find the ground truth of the firing probability, that is the real firing probability that led to the generation of the spike train obtained as real data. One must remember that this spiking probability is time dependent and must lay between 0 and 1. So, somehow one should find the way to transform the spike train along time into a firing probability changing in time as well. Therefore, at each time step one can obtain the instantaneous firing probability that led to the generation or not of a spike at that moment.

The procedure followed is the same one that was followed in section 4.3.1 Firing Probability Estimation within simulation data.

It consisted in performing a **convolution** between the spike train and a Gaussian distribution function. Once having done so, one can obtain an estimation of the firing probability density by adding all of these Gaussian curves together. Hence, the time regions in which the neuron fired a lot will have high firing probabilities, while those in which only a few spikes were generated will have low firing probabilities.

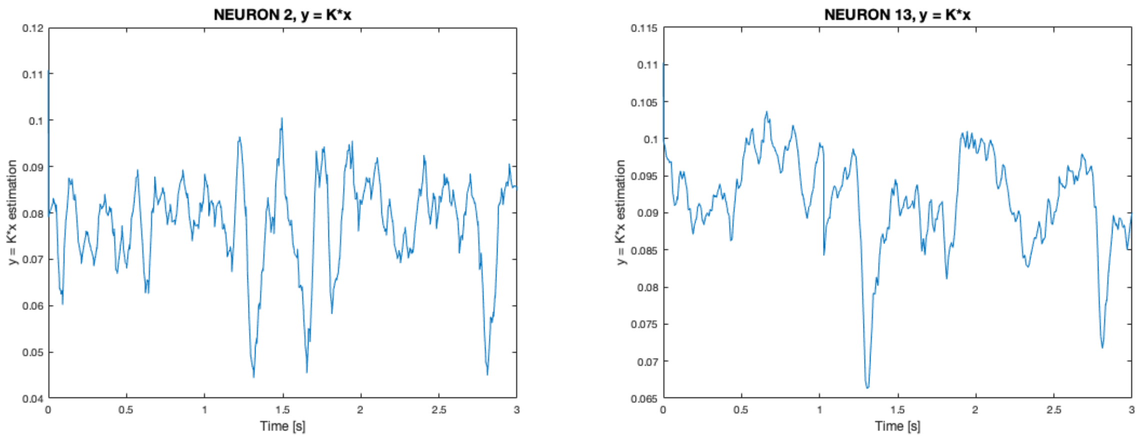
The two parameters that must be fitted in this methodology are both the mean μ and the variance σ of the convoluted Gaussian distribution ($N(\mu, \sigma)$). The values used in here have been the same ones used in simulation data as we expect both firing probabilities to be similar to each other and hence to need similar parameters when estimated. Therefore, we cans set $\mu = 0$ and $\sigma = 1.9650$.

Besides from that, when adding the different Gaussian distributions, the resulting signal doesn't lie between 0 and 1 and hence we had to normalise it by dividing the whole resulting signal by its maximum value. Moreover, it has been imposed that the mean of the estimated firing probability along time should match the quotient $\frac{\#spikes}{\#samples}$. Hence, we have multiplied the whole signal by $\frac{\#spikes/\#samples}{mean(p_{firing})}$. One must notice that this quotient can sometimes be bigger than one. In that case, we didn't apply the rescaling because it would lead to firing probabilities greater than one.

Output generation

Once we have the ground truth firing probability with which to compare we need to take the kinematic vector as input and feed it to the model we have just predicted. With it, we will obtain a resulting firing probability that will be compared to the ground truth one.

The first partial output of the LNP model built is the resulting feature space signal $y(t)$. It is represented in Figure 5.9.

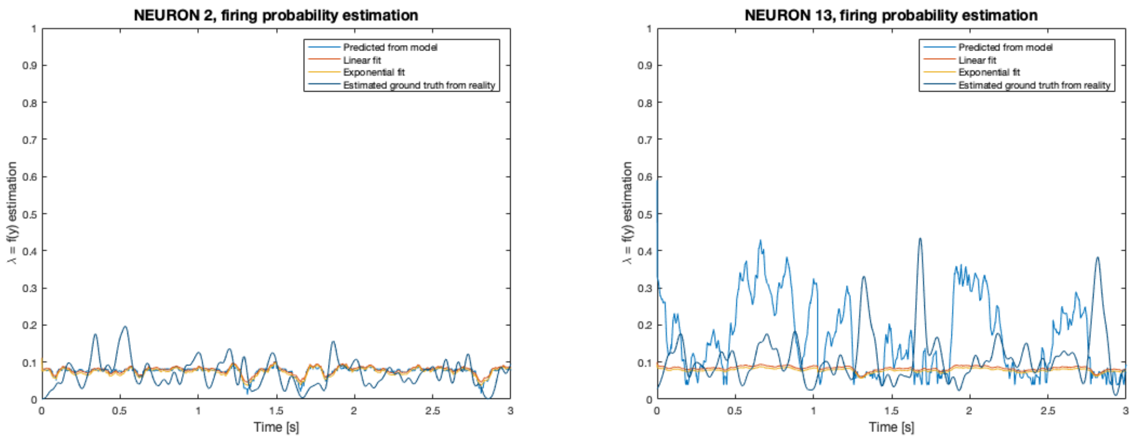


(a) Neuron 2 (b) Neuron 13

Figure 5.9: Estimated feature space vector $y(t)$ for neuron 2 and 13

Not many things can be extracted from these plots as they belong to an unknown feature space that tries to weight the importance of each of the 5 dimensions in the input kinematic vector.

The next step is to apply the nonlinear function $f(y)$ to $y(t)$. We have plotted the results using first the actual estimation of $f(y)$, second the linear fitting and third the exponential one. The results can be seen in Figure 5.10.



(a) Neuron 2 (b) Neuron 13

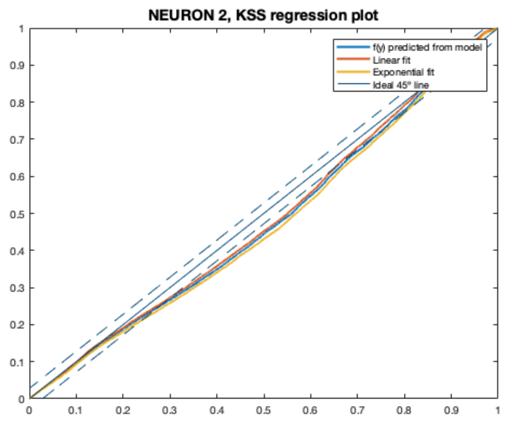
Figure 5.10: Estimated p_{firing} for neuron 2 and 13

As one can see, for neuron 2, the actual original estimation of the nonlinear function (in blue) is very similar to the ground truth one, following its hills and valleys along time. In this case both the linear and exponential cases also follow the trends but with way much less amplitude. Hence, they kind of like homogenise the resulting p_{firing}.
On the other hand, for neuron 13, something similar happens. In this case, it is clear that the fittings help the estimation to be more constant and less fluctuating.

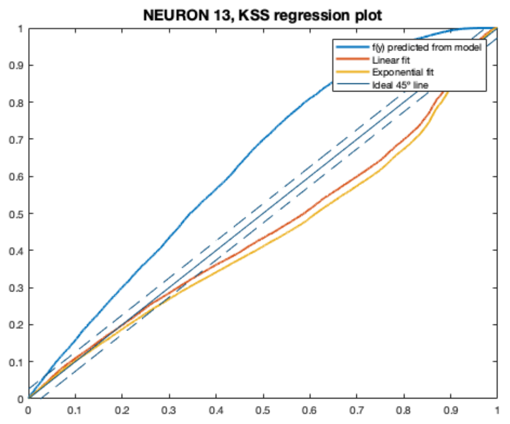
Kolmogorov-Smirnov Test

Once we have the ground truth firing probability directly obtained from the real data spike train and the resulting firing probability obtained after feeding our estimated model with data, we need an appropriate goodness-of-fit measure. As it has been used in simulation, the proper measure is the Kolmogorov-Smirnov Test, which bases its realibility in the Time Rescaling Theorem (see section 4.5.2 Time Rescaling Theorem for more details on it).

Hence, after following the proper steps described in section 4.5.2 Kolmogorov-Smirnov Test, one has obtained the following KSS plots for neurons 2 and 13.



(a) Neuron 2



(b) Neuron 13

Figure 5.11: KSS plot for neuron 2 and 13

As we can see in Figure 5.11(a), all three approaches lead to quite good results while in Figure 5.11(b), we can see a small improvement of the fittings with respect to the original nonlinear function estimation. The latter behaviour is the one we found in simulation data and the one we were expecting. Still, one must look at the resulting metrics.

METRIC	**n15**	**n2**	**n5**	**n16**	**n13**
KSS	16.2726	2.1850	3.1916	44.6294	8.1764
KSS linear	6.7685	1.8842	5.9441	3.5623	4.1671
KSS exponential	9.2153	2.6621	8.9506	2.3470	5.1386
	n14	**n11**	**n19**	**n8**	**n6**
KSS	18.0825	14.3594	4.5381	10.7801	29.1903
KSS linear	2.0217	2.3678	5.2771	5.8199	7.2290
KSS exponential	1.9103	4.3620	5.3195	3.4403	13.6398

Table 5.2: Output evaluation KSS metrics comparison for the ten task-related neurons

If we gather the obtained metrics for the ten different considered related neurons we can see that in general, both the fittings enhance the results compared to the direct $f(y)$ estimation, getting lower KSS metrics.

Moreover, if we compare the linear with the exponential approaches we can see that in the majority of the cases the linear approach gets better results than the exponential one (with

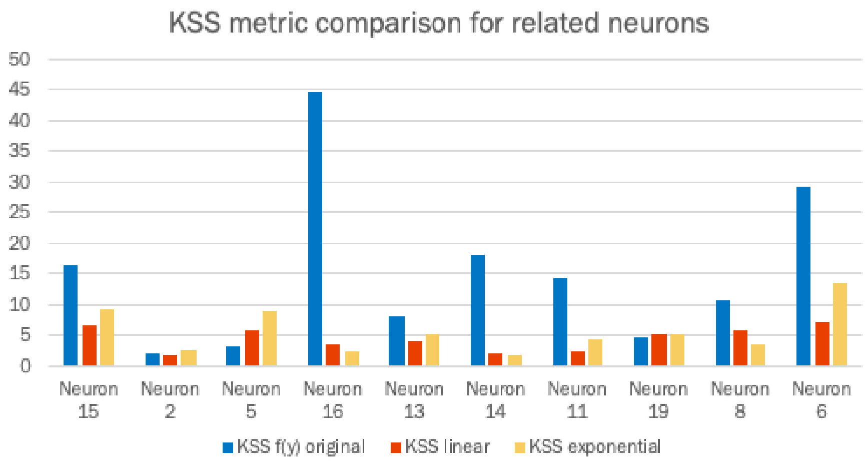


Figure 5.12: Output evaluation KSS metrics comparison for the ten task-related neurons

exception of neuron 16, 14 and 8). This behaviour is the same as the one we obtained in simulated data and it shows us that the nonlinearities introduced by the exponential function doesn't provide with useful extra information that helps the model to better predict the firing probability.

In general term though, both the linear and the exponential fittings provide pretty good fittings of the nonlinear function with KSS values below 10 in most of the cases.

This results lead to the conclusion that in general, the linear fit already represents data quite well. It is only in some punctual neurons, more concretely in neurons 16, 14 and 8, that the exponential approach brings extra information that makes the model predict a better estimation of the whole procedure.

Chapter 6

Conclusions

Finally, to conclude with this thesis, we can state that the LNP approach is a good model to encode the relationship between the given kinematic vector $x(t)$ and the related spike train; or more specifically, the resulting firing probability $p_{firing}(t)$.

Regarding the simulation results, we found that the 4 different approaches considered are suitable for our purpose, leading to pretty good KS metrics.
In all cases, the corresponding parameter errors were quite low, achieving estimations quite near the theoretical ground truth values. Moreover, when considering the nonlinear function $f(y)$, the corresponding fittings (either linear or exponential) enhanced the performance when compared to that when using the direct estimated function.
We found out that in general, the linear approaches gave better results than the exponential ones (maybe because of them being easier to estimate) and that in turn, the window approaches overperformed the constant ones. This last fact highlights the fact that the window approaches use more information as they not only consider the current sample but the history before it. Hence, the best approach expected from simulation is the *Window-Linear* case. This one has been used as the baseline model and it has been compared with the *Window-Exponential* case.

Once knowing so, the procedure has been applied to real experimental data as well. The model proportions a good methodology to predict firing probability in terms of mean value and trend, but it isn't capable of accurately predicting the little oscillations in it. The results obtained are quite similar to those in simulation: in most of the cases the linear approach fitted even better than the exponential one and hence the nonlinearities included by the latter were not remarkable when predicting the resulting firing probability.

All in all, we have obtained a proper model that establishes the relationship between the kinematic vector and the resulting firing probability. The hypothesis that including some nonlinearities could help the encoding model to be a better one has been disproved with most of the neurons as the linear approach fits better. Hence, we can state that for most of the neurons, a linear encoding model will be good enough for the prediction to lead to reliable and good results.

Chapter 7

Future work

The fact that we encountered that for most of the task-related neurons the exponential nonlinearity didn't give extra useful information made us think that there might still exist other nonlinear behaviours within the process that couldn't be represented by such nonlinear function $f(y)$. This statement arises some new challenges that could be considered in order to enhance the model estimation and its performance after all.

There are several issues that haven't been taken into account when modelling the encoding procedure through a LNP model.
First, the spikes were assumed to be independent with each other when considering the inhomogeneous Poisson spike generator. However, and as one may expect, they have a **dependency** between them so adding some relations between them could enhance the model performance

Also, one could consider the **refractory period**. The refractory period is the time after a generated spike in which that neuron is unable to fire again. In real life, the neurons need this time in order to 'recover' from previous firings so that they can throw another action potential.

In the LNP model approach we considered, no refractory period is taken into account: at each time step, one decides whether a spike is generated or not, and no 'recovery' period is considered after each spike.

Alongside, one could also consider **bursting**. The term bursting refers to the state of a neuron when it repeatedly fires creating bursts of spikes. These moments in time aren't treated distinctively in the proposed encoding model. These moments could be predicted so that the resulting encoding model could adapt to them an provide with a better estimation.

Finally, the next step would be to consider also the **relationship between the different neurons**. At the end, with our LNP model we are treating each neuron as an independent isolated unit that doesn't interact with the others. However, this is not true, all neurons are connected and they depend on each other and it might be helpful to try to include such relationships in the model so that at the end it better encodes the reality of what is happening in our brains

Hence, I encourage the scientific world to try to keep working on that as these kind of studies are key to tackle the global challenge of providing functional limbs to those who don't have a proper one.

Chapter 8

Appendix

8.1 Time Rescaling Theorem proof

Theorem 8.1.1. ***Time Rescaling Theorem*** *Let* $0 < u_1 < u_2 < ... < u_n \leq T$ *be a realisation from a point process with a conditional intensity function* $\lambda(t \mid H_t)$ *satisfying* $0 < \lambda(t \mid H_t)$ *for all* $t \in (0, T]$*. Define the transformation:*

$$\Lambda(u_k) = \int_0^{u_k} \lambda(u|H_u)du \tag{8.1}$$

for $k = 1, ..., n$*, and assume* $\Lambda(t) < \infty$ *with probability one for all* $t \in (0, T]$*.*
Then the $\Lambda(u_k)$*'s are a Poisson process with unit rate.*

Proof. Lets first define some equation that will be needed for the proof itself.

Let $N(t)$ be the number of spikes in the interval $(0, t]$.
Then we can define the conditional intensity function as:

$$\lambda(t \mid H_t) = lim_{\Delta t \to 0} \left(\frac{Pr(N(t + \Delta t) - N(t) = 1 \mid H_t)}{\Delta t} \right) \tag{8.2}$$

where $H_t = 0 < u_1 < ... < u_{N(t)} < t$ is the history of the process at time t. One must notice that if the point process is an inhomogeneous Poisson process, then $\lambda(t \mid H_t) = \lambda(t)$, the Poisson rate. Hence, one can say that the definition of the conditional intensity function is a generalisation of the Poisson rate.

Also, one may notice that the conditional intensity function can be expressed in terms of the spike time probability density $f(t \mid H_t)$.
In order to do so one should consider the Hazard function

$$\begin{aligned} Pr(N(t+\Delta t) - N(t) = 1 \mid H_t) = Pr(u_k \in [t, t+\Delta t) \mid u_k > t, H_t) &=_{eq.8.4} \\ = \frac{Pr(u_k \in [t, t+\Delta t) \mid H_t)}{Pr(u_k > t \mid H_t)} = \frac{\int_t^{t+\Delta t} f(u \mid H_t)du}{1 - \int_{u_{N(t)}}^t f(u \mid H_t)du} &\approx \\ \approx \frac{f(t \mid H_t)\Delta t}{1 - \int_{u_{N(t)}}^t f(u \mid H_t)du} =_{eq.8.2} \lambda(t \mid H_t)\Delta t & \end{aligned} \tag{8.3}$$

where we have used that

$$P(A \mid B, C) = \frac{P(B \mid A, C)P(A \mid C)}{P(B \mid C)} \tag{8.4}$$

with $A = u_k \in [t, t+\Delta t)$, $B = u_k > t$ and $C = H_t$. Hence $P(B \mid A, C) = Pr(u_k > t \mid u_k \in [t, t+\Delta t), H_t) = 1$.

So, taking the last two expressions, we can express the conditional intensity function as:

$$\lambda(t \mid H_t) = \frac{f(t \mid H_t)}{1 - \int_{u_{N(t)}}^t f(u \mid H_t)d} \tag{8.5}$$

We want to use this result with a spike train, so one needs to compute the joint probability density of exactly n spikes in $(0, T]$. Therefore we have:

$$\begin{aligned} f(u_1, ..., u_n \cap N(T) = n) &= f(u_1, ..., u_n \cap u_{n+1} > T) = \\ &= f(u_1, ..., u_n \cap N(u_n) = n) Pr(u_{n+1} > T \mid u_1, ..., u_n) = \\ &= \prod_{k=1}^{n} \lambda(u_k \mid H_{u_k})\, exp\left(-\int_{u_{k-1}}^{u_k} \lambda(u \mid H_u) du\right) exp\left(-\int_{u_n}^{T} \lambda(u \mid H_u) du\right) \end{aligned} \tag{8.6}$$

where we have assumed the Poisson spike generator approach with probability function $f(x) = \frac{\lambda^x}{x!} exp(.\lambda)$. This implies that $f(x = 1) = \lambda exp(-\lambda)$ and therefore the joint probability density of n events in $(0, u_n]$ is $f(u_1, ..., u_n \cap N(u_n) = n) = \prod_{k=1}^{n} \lambda(u_k \mid H_{u_k}) exp\left(-\int_{u_{k-1}}^{u_k} \lambda(u \mid H_u) du\right)$

Now that we have this, if we take the following transformations

$$\begin{aligned} \tau_k &= \Lambda(u_k) - \Lambda(u_{k-1}) \quad for\ k = 2, ..., n \\ \tau_T &= \int_{u_n}^{T} \lambda(u \mid H_u) du \end{aligned} \tag{8.7}$$

then one has to prove that the τ_k's are independent identically distributed exponential random variables with mean 1.

Since the τ_k transformation is one-to-one, then $\tau_{n+1} > \tau_T\ iff\ u_{n+1} > T$ and then the joint probability density of τ_k's is:

$$f(\tau_1, ..., \tau_n \cap \tau_{n+1} > \tau_T) = f(\tau_1, ..., \tau_n) Pr(\tau_{n+1} > \tau_T \mid \tau_1, ..., \tau_n) \tag{8.8}$$

Let's take the first term and apply a multivariate change of variable from τ_k's to u_n's. One knows that if we have $y = g(x)$, then $p_Y(y) = |J_{y \to x}| p_X(x)$ where $|J_{y \to x}|$ is the determinant of the Jacobian of the transformation between x and y. In our case $x = u_j$ and $y = \tau_k$.

The determinant is computed as $|J| = |\prod_{k=1}^{n} J_{kk}|$ with $J_{kk} = \frac{\delta u_k}{\delta \tau_k} = \lambda(u_k \mid H_{u_k})^{-1}$. Hence:

$$\begin{aligned} f(\tau_1, ..., \tau_n) = |J| f(u_1, ..., u_n \cap N(u_n) = n) &=_{eq.8.6} \\ &= \prod_{k=1}^{n} \lambda(u_k \mid H_{u_k})^{-1} \prod_{k=1}^{n} \lambda(u_k \mid H_{u_k})\, exp\left(-\int_{u_{k-1}}^{u_k} \lambda(u \mid H_u) du\right) = \\ &= \prod_{k=1}^{n} exp\left(-[\Lambda(u_k) - \Lambda(u_{k-1})]\right) = \prod_{k=1}^{n} exp(-\tau_k) \end{aligned} \tag{8.9}$$

If we take the second term we have:

$$\begin{aligned} Pr(\tau_{n+1} > \tau_T \mid \tau_1, ..., \tau_n) &= Pr(u_{n+1} > T \mid u_1, ..., u_n) =_{eq.8.6} \\ &= exp\left(-\int_{u_n}^{T} \lambda(u \mid H_u) du\right) =_{eq.8.7} \\ &= exp(-\tau_T) \end{aligned} \tag{8.10}$$

Therefore, if we join the two results, equation 8.8 becomes:

$$f(\tau_1, ..., \tau_n \cap \tau_{n+1} > \tau_T) = \prod_{k=1}^{n} \left(exp(-\tau_k)\right)\ exp(-\tau_T) \tag{8.11}$$

where the first term is a history-dependent rescaling term and the second one is the probability density function of a Poisson distribution with unit rate.
Therefore, the Time Rescaling Theorem is proved. □

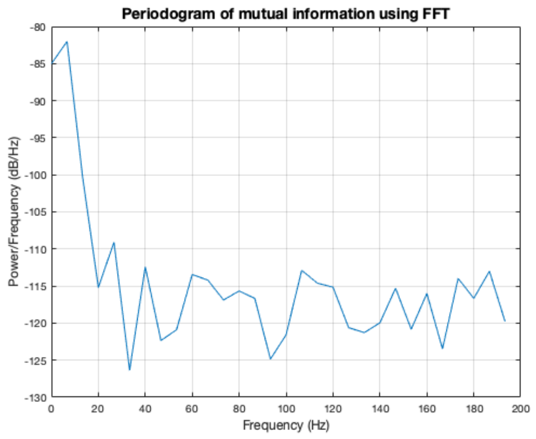
8.2 Additional Figures and Tables

In this section, one gathers the figures that were not displayed in the body of the thesis.

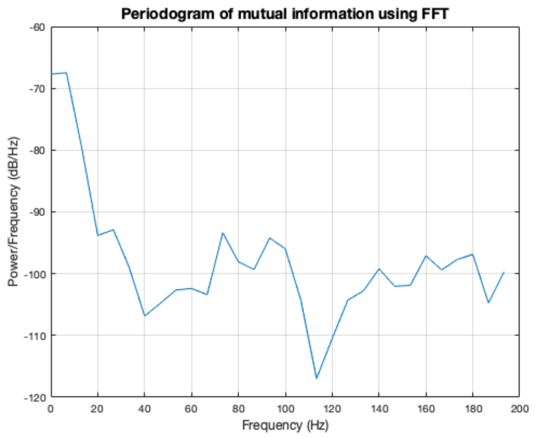
8.2.1 Simulated Encoding additional Figure and Tables

Rearding the simulated encoding, the associated pictures for both the *Constant-Linear* and *Constant-Exponential* approach are displayed one next to the other.

Model Results

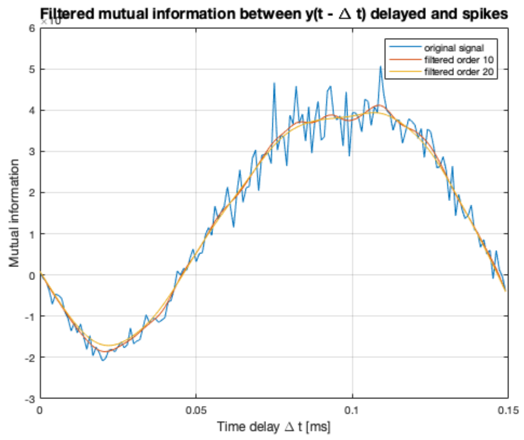


(a) *Constant-Linear* case

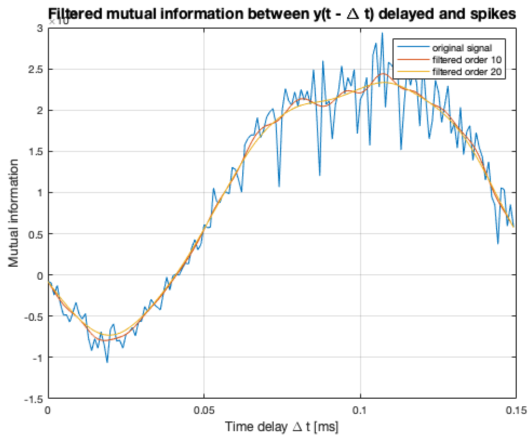


(b) *Constant-Exponential* case

Figure 8.1: FFT of the mutual information plot for *Constant* cases

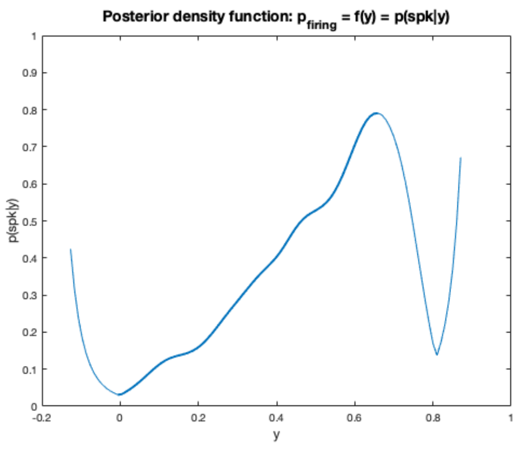


(a) *Constant-Linear* case

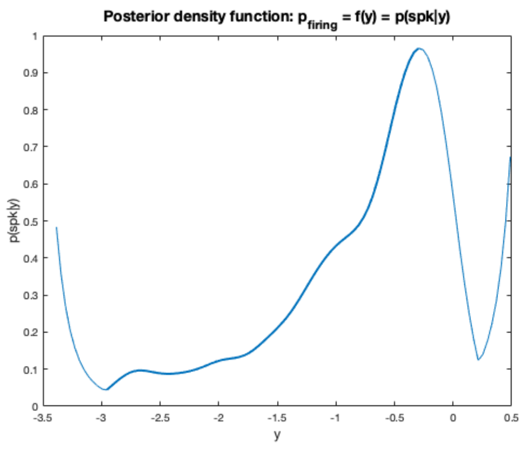


(b) *Constant-Exponential* case

Figure 8.2: Filtered Mutual Information for *Constant* cases

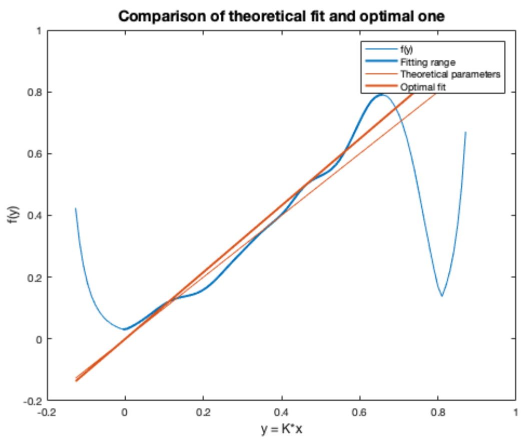


(a) *Constant-Linear* case

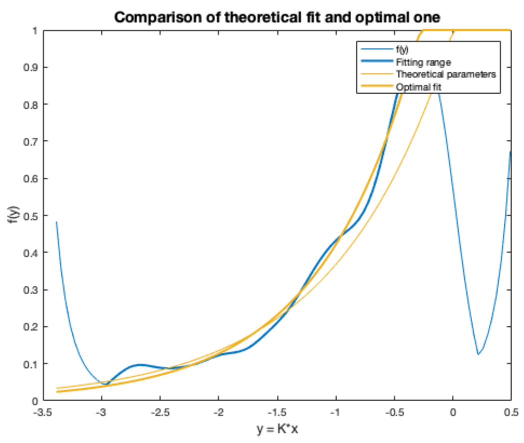


(b) *Constant-Exponential* case

Figure 8.3: $f(y)$ estimation for *Constant* cases

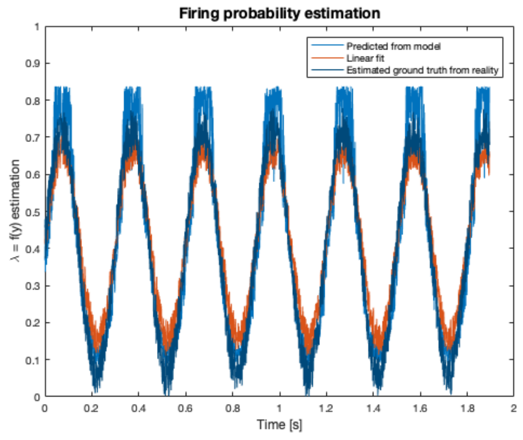


(a) *Constant-Linear* case

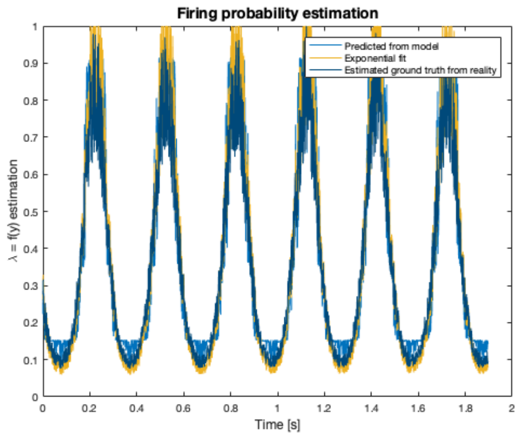


(b) *Constant-Exponential* case

Figure 8.4: Nonlinear function $f(y)$ along with linear and exponential fittings for *Constant* cases

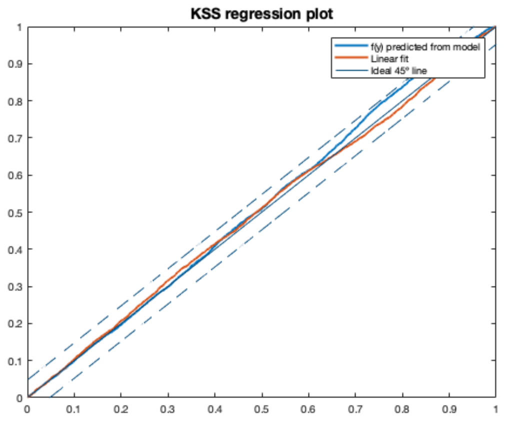


(a) *Constant-Linear* case

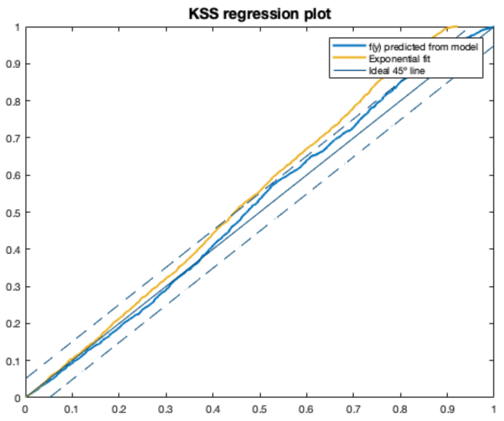


(b) *Constant-Exponential* case

Figure 8.5: p_{firing} estimation for *Constant* cases



(a) *Constant-Linear* case



(b) *Constant-Exponential* case

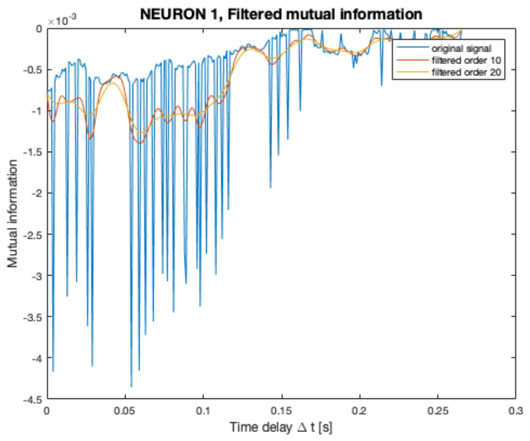
Figure 8.6: KSS plot for *Constant* cases

8.2.2 Experimental Encoding additional Figures and Tables

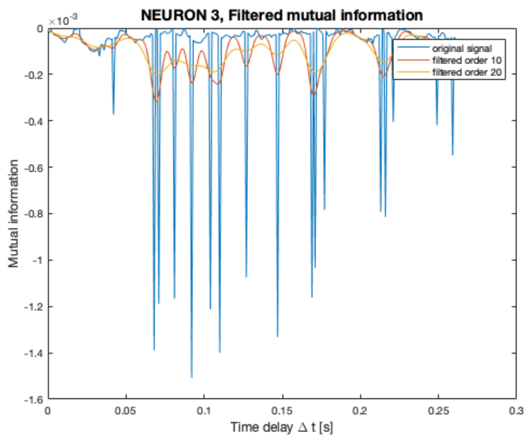
Model Results

Here we gather the results of the rest neurons that weren't displayed in the body of the thesis (except neuron 2 and 13).

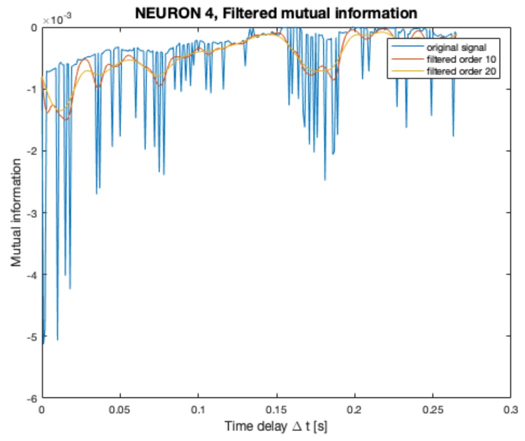
From the K computation and Δt estimation we only plot the resulting filtered mutual information plot.



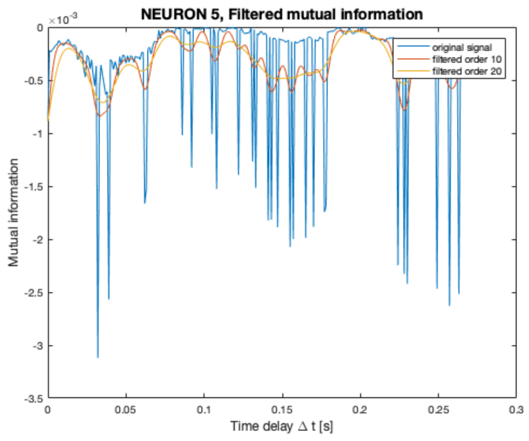
(a) Neuron 1



(b) Neuron 3

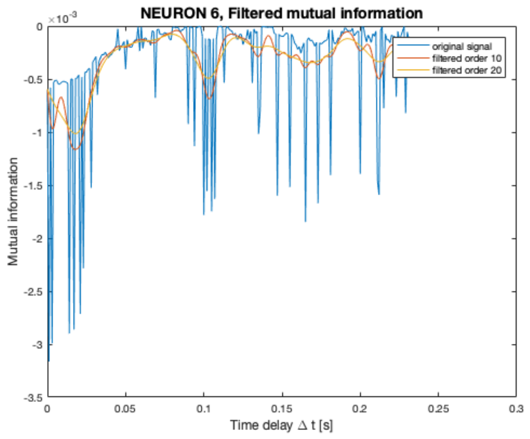


(c) Neuron 4

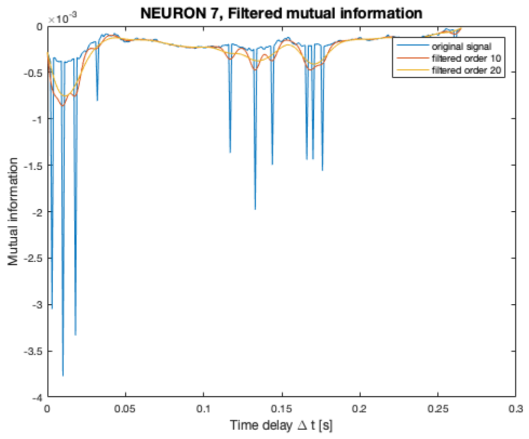


(d) Neuron 5

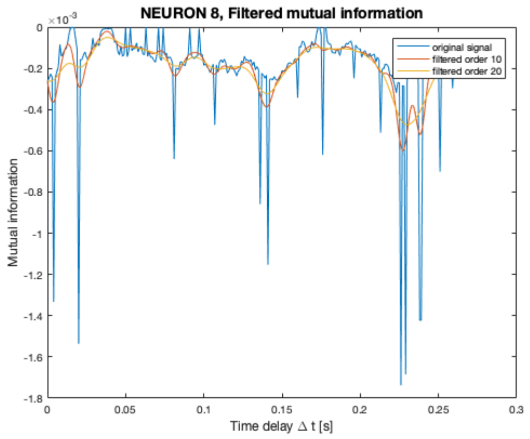
Figure 8.7: Filtered Mutual Information for neuron 1, 3, 4 and 5



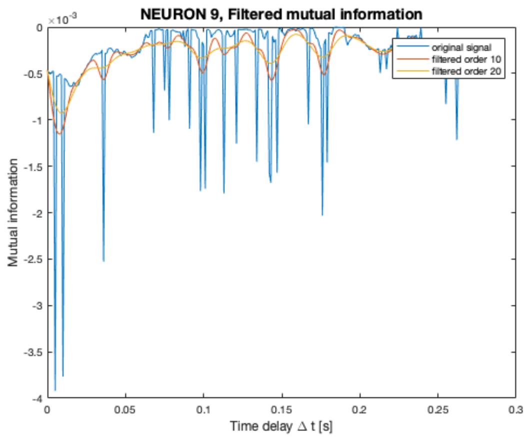
(a) Neuron 6



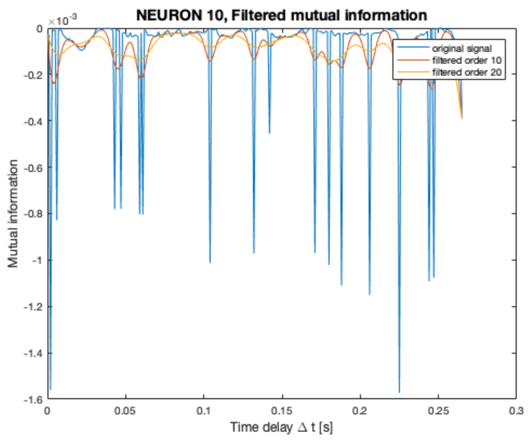
(b) Neuron 7



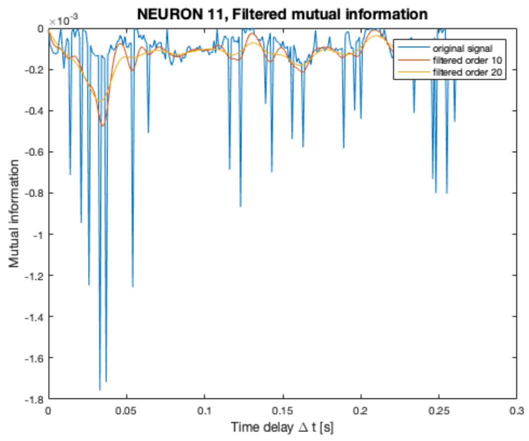
(c) Neuron 8



(d) Neuron 9

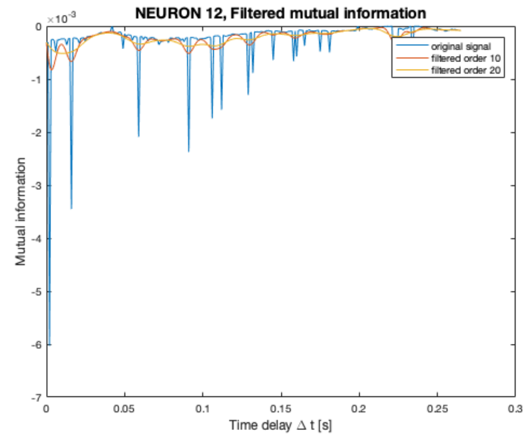


(e) Neuron 10

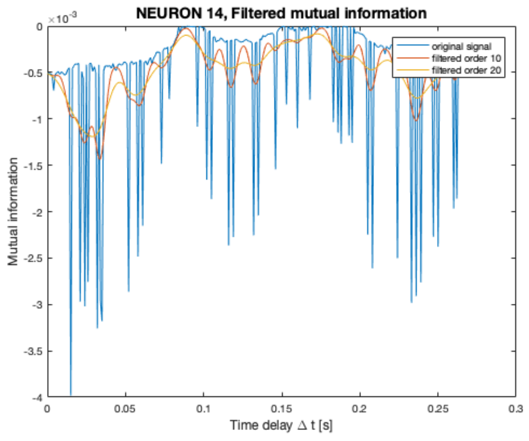


(f) Neuron 11

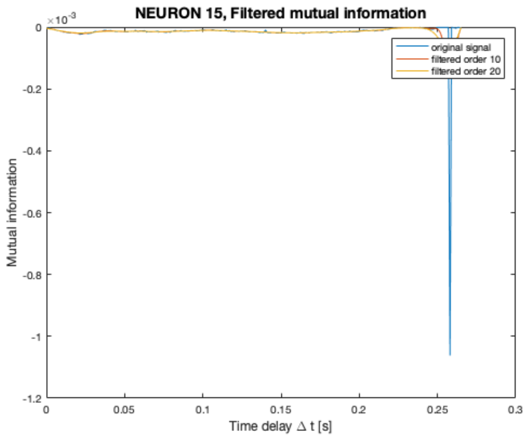
Figure 8.8: Filtered Mutual Information for neuron 6, 7, 8, 9, 10 and 11



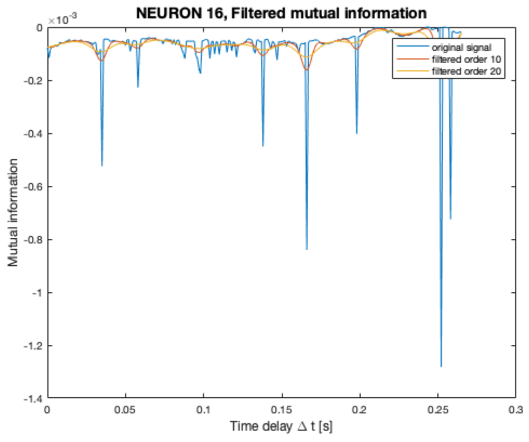
(a) Neuron 12



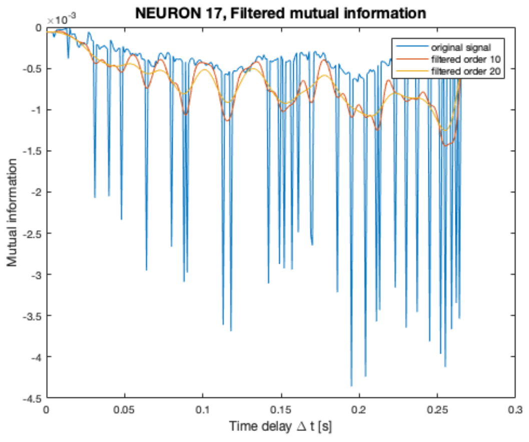
(b) Neuron 14



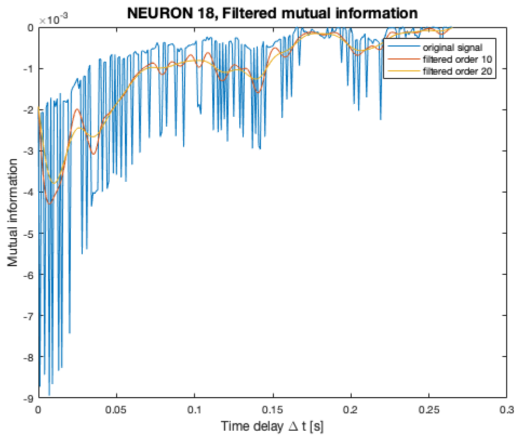
(c) Neuron 15



(d) Neuron 16

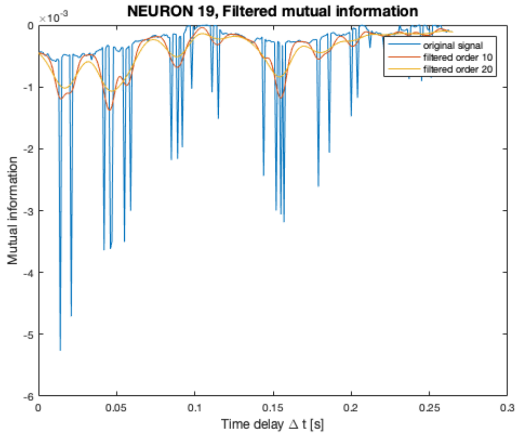


(e) Neuron 17

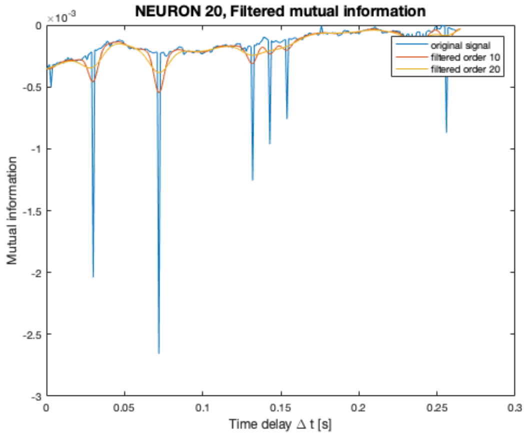


(f) Neuron 18

Figure 8.9: Filtered Mutual Information for neuron 12, 14, 15, 16, 17 and 18



(a) Neuron 19



(b) Neuron 20

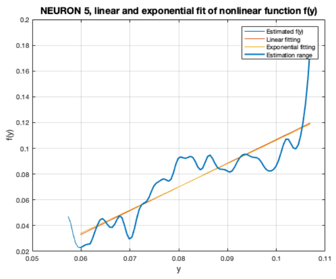
Figure 8.10: Filtered Mutual Information for neuron 19 and 20

The resulting time delay Δt and linear filter K estimations for each of the 20 neurons are gathered in Table 8.1.

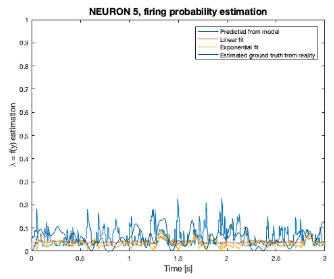
METRIC	**n1**	**n2**	**n3**	**n4**	**n5**
Δt	266	66	1	220	201
K	$\begin{bmatrix}0.0437\\-0.0005\\0.2181\\0.0587\\0.1509\end{bmatrix}$	$\begin{bmatrix}-0.0469\\-0.0046\\0.1750\\-0.0115\\0.0814\end{bmatrix}$	$\begin{bmatrix}0.0285\\0.0070\\-0.6614\\-0.0981\\0.0579\end{bmatrix}$	$\begin{bmatrix}-0.0364\\-0.0126\\-0.0013\\0.0934\\0.1239\end{bmatrix}$	$\begin{bmatrix}-0.0230\\-0.0130\\0.0252\\-0.0083\\0.0912\end{bmatrix}$
METRIC	**n6**	**n7**	**n8**	**n9**	**n10**
Δt	84	266	39	159	36
K	$\begin{bmatrix}-0.0443\\0.0056\\-0.3438\\-0.0348\\0.1481\end{bmatrix}$	$\begin{bmatrix}-0.0097\\-0.0059\\-0.0018\\0.0462\\0.0847\end{bmatrix}$	$\begin{bmatrix}-0.0001\\0.0194\\0.2767\\0.0192\\0.1110\end{bmatrix}$	$\begin{bmatrix}-0.0307\\0.0093\\-0.6582\\-0.1333\\0.1508\end{bmatrix}$	$\begin{bmatrix}-0.0235\\-0.0012\\0.1394\\0.0623\\0.0540\end{bmatrix}$
METRIC	**n11**	**n12**	**n13**	**n14**	**n15**
Δt	210	215	151	176	266
K	$\begin{bmatrix}0.0158\\0.0085\\-0.1415\\-0.0314\\0.1045\end{bmatrix}$	$\begin{bmatrix}-0.0236\\-0.0057\\0.3652\\0.1039\\0.0713\end{bmatrix}$	$\begin{bmatrix}0.0004\\0.0059\\0.2844\\0.0061\\0.0731\end{bmatrix}$	$\begin{bmatrix}0.0275\\-0.0112\\-0.0743\\0.0215\\0.0984\end{bmatrix}$	$\begin{bmatrix}-0.0035\\-0.0087\\-0.0234\\0.0143\\0.0148\end{bmatrix}$
METRIC	**n16**	**n17**	**n18**	**n19**	**n20**
Δt	213	9	266	106	266
K	$\begin{bmatrix}-0.0110\\0.0020\\0.1857\\0.0018\\0.0365\end{bmatrix}$	$\begin{bmatrix}0.0325\\-0.0400\\2.6536\\0.4703\\0.1405\end{bmatrix}$	$\begin{bmatrix}0.0556\\0.0164\\0.1596\\-0.0797\\0.1898\end{bmatrix}$	$\begin{bmatrix}0.0407\\0.0041\\-0.6122\\-0.0919\\0.1129\end{bmatrix}$	$\begin{bmatrix}-0.0011\\-0.0029\\0.0343\\-0.0583\\0.0736\end{bmatrix}$

Table 8.1: Δt and **K** estimation for each of the neurons

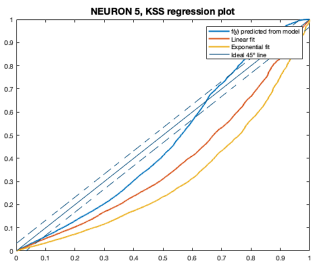
From this point on, the plots displayed will only be from those neurons belonging to the task-related list (again avoiding those from neuron 2 and 13 already displayed in the main body). The resulting estimated $f(y)$ is plotted along with the estimation of the firing probability and the KSS plot.



(a) $f(y) fittings$

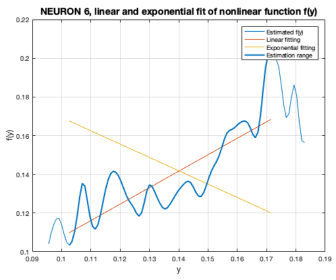


(b) Estimation of p_{firing}

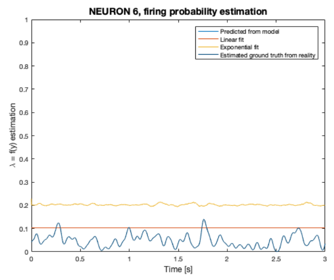


(c) KSS plot

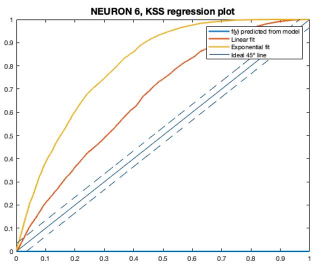
Figure 8.11: Neuron 5 results



(a) $f(y) fittings$

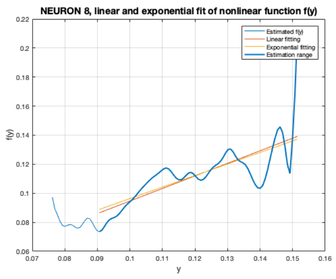


(b) Estimation of p_{firing}

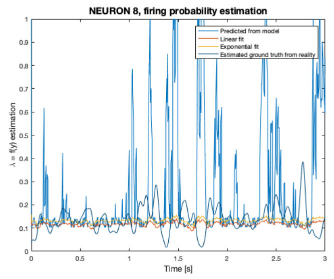


(c) KSS plot

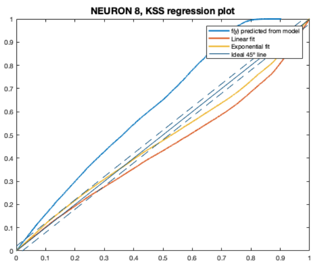
Figure 8.12: Neuron 6 results



(a) $f(y) fittings$

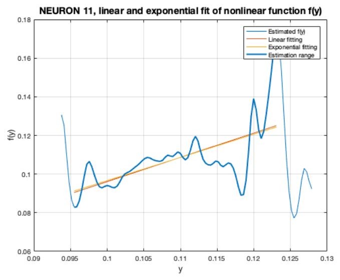


(b) Estimation of p_{firing}

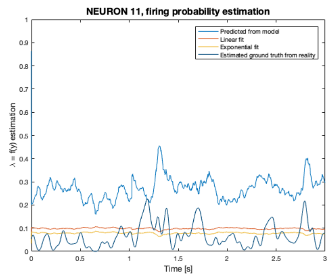


(c) KSS plot

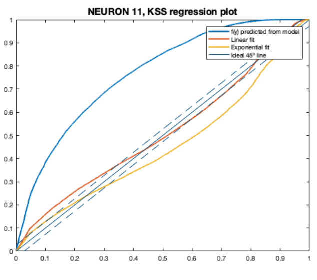
Figure 8.13: Neuron 8 results



(a) $f(y) fittings$

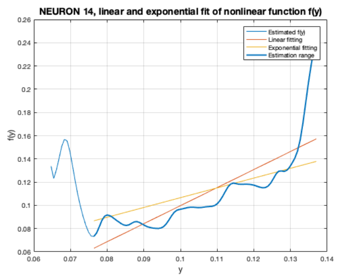


(b) Estimation of p_{firing}

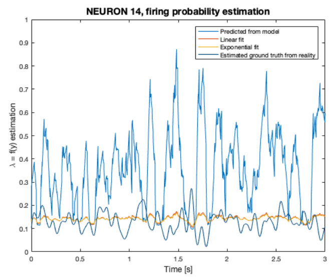


(c) KSS plot

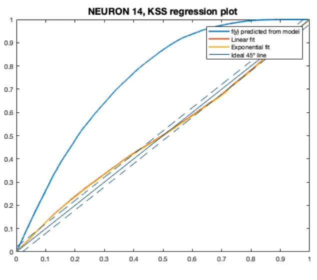
Figure 8.14: Neuron 11 results



(a) $f(y) fittings$

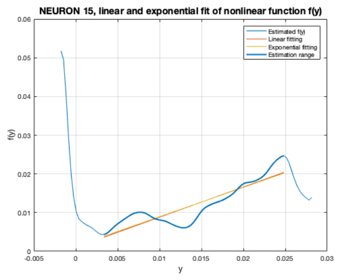


(b) Estimation of p_{firing}

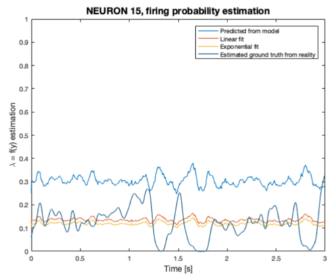


(c) KSS plot

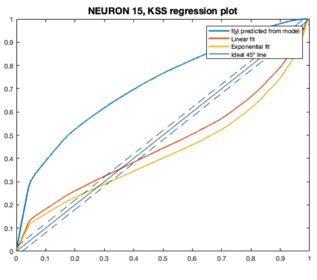
Figure 8.15: Neuron 14 results



(a) $f(y) fittings$

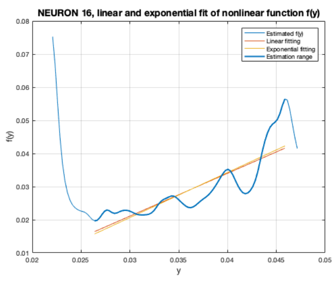


(b) Estimation of p_{firing}

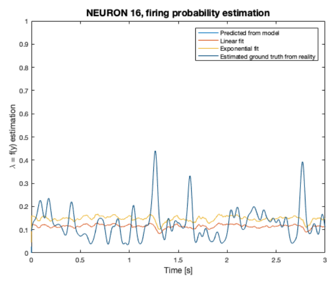


(c) KSS plot

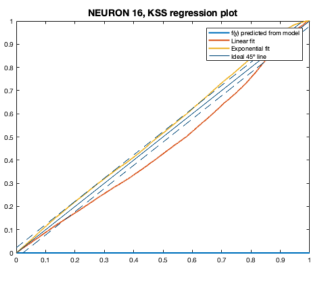
Figure 8.16: Neuron 15 results



(a) $f(y) fittings$

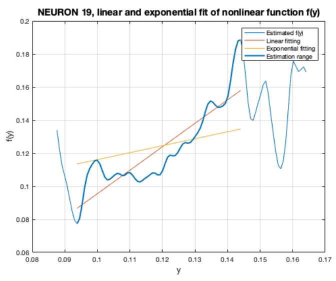


(b) Estimation of p_{firing}

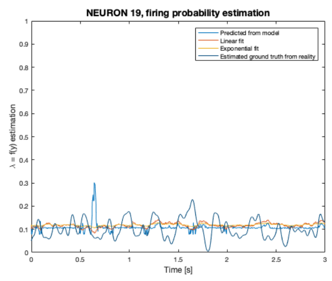


(c) KSS plot

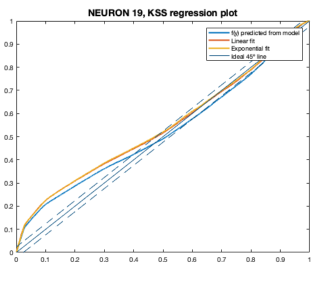
Figure 8.17: Neuron 16 results



(a) $f(y) fittings$



(b) Estimation of p_{firing}



(c) KSS plot

Figure 8.18: Neuron 19 results

8.3 Matlab codes

This section gathers the MATLAB codes used to perform all the computations mentioned in the thesis.

Only the experimental codes are provided as they are a generalisation of how the simulation ones are.

8.3.1 Main code

The main MATLAB code is displayed below.

```
%% EXPERIMENTAL DATA: KINEMATIC_WINDOW - LINEAR & EXPONENTIAL APPROACH
% Linear-nonlinear-Poisson cascade model
 clear; clc;
 load('Rat11_M1_BC_20181212.mat')
 load('EventLabel.mat')

 % Epsilon value, added to avoid 0 denominators
 eps = 1e-16;

 % Define color for graphics
 blue = [0, 0.4470, 0.7410];
 dark_blue = [0 0.286274522542953 0.47843137383461];
 green = [0.4660 0.6740 0.1880];
 dark_green = [0.356862753629684 0.513725519180298 0.141176477074623];

 red = [0.8500, 0.3250, 0.0980];
 yellow = [0.9290, 0.6940, 0.1250];

%% Experimental data
% To run this code, one must import the variables X and Spk from a matlab
% .mat object.

% Signal
% We consider the y position movement as it is the one more relatable to
% the reality of the experiment (the movement to push the button is ...
   vertical)
dt = 1e-3;
tVec_initial = 0:dt:(length(X(:,2))-1)*dt;
plot_signal_spikes(tVec_initial, Spk(1,:), X(:,2), 'Time [s]', 'x(t)', ...
    'Kinematic vector x(t) (vertical position) with spikes from neuron 1')

%{
figure(1)
plot(X(:,2))
xlabel('Time [s]', 'FontSize', 12); ylabel('x(t)', 'FontSize', 12)
title('Stimulus x(t) (only vertical position movement)', 'FontSize', 14)

% Spikes
figure(2)
bar(Spk(1,:), 'FaceColor', dark_blue)
xlabel('Time [s]', 'FontSize', 12); ylabel('Spikes', 'FontSize', 12);
title('Spikes of first neuron', 'FontSize', 14)
%}

%% LABEL Construction
% The time unit is second while the index of data is 10ms.
% Therefore, you can multiply the label time by 100 to transform the ...
    time into data index.
% We'll order the events in time, labelling each of the events as:
% 1 - HighFail
% 2 - HighReward
% 3 - HighStart
% 4 - LowFail
% 5 - LowReward
% 6 - LowStart
% 7 - PressHigh
% 8 - PressLow

% Set labels as they happened in time
time_labels = [100*HighFail, ones(length(HighFail),1); 100*HighReward, ...
    2*ones(length(HighReward),1); 100*HighStart, ...
    3*ones(length(HighStart),1);100*LowFail, ...
    4*ones(length(LowFail),1);100*LowReward, ...
    5*ones(length(LowReward),1);100*LowStart, ...
    6*ones(length(LowStart),1);100*PressHigh, ...
    7*ones(length(PressHigh),1); 100*PressLow, 8*ones(length(PressLow),1)];
time_labels = sortrows(time_labels);

% The proper order should be HighStart, PressHigh, HighReward or LowStart,
% PressLow, LowReward. Hence: 3 - 7 - 2 or 6 - 8 - 5
% Let's create a matrix with the triads of times of the success ...
    occurrences in
% every row. The first row contains whether the ocurrence is high (1) ...
    or low
% case (0)

success = [];
first_start = find(time_labels(:,2) == 3 | time_labels(:,2) == 6);
for idx = 1:length(first_start)-1
    i = first_start(idx);
    triad = time_labels(i:i+2,2);

    % Make sure there are is more than 1 s between events
    %{
    Δ_t_1 = time_labels(i+1,1) - time_labels(i,1);
    Δ_t_2 = time_labels(i+2,1) - time_labels(i+1,1);
    %}

    if triad == [3; 7; 2] %& Δ_t_1 ≥ 1000 & Δ_t_2 ≥ 1000
        success = [success; 1, time_labels(i:i+2,1)'];
    elseif triad == [6; 8; 5] %& Δ_t_1 ≥ 1000 & Δ_t_2 ≥ 1000
        success = [success; 0, time_labels(i:i+2,1)'];
    end
end

% We will consider the signal only at these successes: we'll consider 500
% ms pre and post of each behaviour and we are going to concatenate ...
    them in
% time

pre_post = 500;
data_limit = length(X(:,2));
```

```

x_success = [];
spikes_success = [];
last_post = 0;

% For each row (success)
for i = 1:length(success(:,1))
    valid = 0;
    signal = [];
    spk_signal = [];

    post = 0;
    % For each column (event)
    for j = 2:4
        event = success(i, j);

        % Pre
        pre = event - pre_post;
        if pre ≤ 0 pre = 1; % pre out of data
        elseif pre ≤ post pre = post + 1; % overlapping within success
        end
        if j == 2 && pre ≤ last_post pre = last_post + 1; end % ...
            overlapping between success

        % Post
        post = event + pre_post;

        % Check all three events belong to data
        if pre > 0 && post < data_limit
            valid = valid + 1;
            signal = [signal; X(pre:post,:)];
            spk_signal = [spk_signal, Spk(:,pre:post)];
        end
    end

    if valid == 3
        x_success = [x_success; signal];
        spikes_success = [spikes_success, spk_signal];
        last_post = post;  % Keep last position of success
    end
end

tVec_success = 0:dt:(length(x_success(:,2))-1)*dt;
plot_signal_spikes(tVec_success, spikes_success(1,:), x_success(:,2), ...
    'Time [s]', 'x(t)', 'Success x(t) (vertical position) with spikes ...
    from neuron 1')

%{
figure(3)
plot(x_success(:,2))
xlabel('Time [s]', 'FontSize', 12); ylabel('x(t)', 'FontSize', 12)
title('Stimulus x(t) (only vertical position movement)', 'FontSize', 14)

figure(4)
bar(spikes_success(1,:))
xlabel('Time [s]', 'FontSize', 12); ylabel('Spikes', 'FontSize', 12)
title('Spikes of first neuron', 'FontSize', 14)
%}

%% FOR EACH NEURON, compute the encoding model

dt = 0.001;

% We should sample the signal and spikes so that we have at least 10 ...
```

```
    triad successes
% If no overlaps were found, we should consider 10 succeses * (2 * 500) ...
    windows * 3 events = 30000 samples
% If there are overlaps we are still considering 10 successes (and even ...
    more)

samples = 30000; % 30 seconds are used to model
tVec = 0:dt:samples*dt-dt;
spikes = spikes_success(1, 1:samples);
x = x_success(1:samples, :)';

samples_plot = 3000; % Only 2 seconds are plotted
plot_signal_spikes(tVec(1:samples_plot), spikes(1:samples_plot), ...
    x(2,1:samples_plot), 'Time [s]', 'x(t)', 'Cropped success x(t) ...
    (vertical position) with spikes from neuron 1')

%{
% Cropped signal
figure(5)
plot(tVec, x(2,:))
title('Cropped signal');
xlabel('Time [s]', 'FontSize', 12); ylabel('Stimulus (y position ...
    movement)', 'FontSize', 12)

% Cropped spikes
figure(6)
bar(tVec, spikes)
title('Cropped spikes');
xlabel('Time [s]', 'FontSize', 12); ylabel('Spikes', 'FontSize', 12)
%}

%% Low pass filtering the signal to find the period (consider y ...
    position movement)

Ts = dt; % Sampling Interval (s)
Fs = 1/Ts;  % Sampling Frequency (Hz)
Fn = Fs/2;  % Nyquist Frequency (Hz) -> maximal possible one to use as ...
    the cutoff frequency in the filter
tVec_filterX = 0:Ts:(length(x(2,:))-1)*Ts;

Fc = 1; % Low pass filter at a cutting frequency of ¬1Hz
wn = Fc/Fn; % We need to normalize the frequency so that it lays ...
    between 0 and 1.
% Since the Nyquist frequency is the biggest we could choose, we divide it
% by it so that the maximum normalized value was 1.

% The order is just to specify how fast the trend of the magnitude in the
% frequency domain drops to 0. The highest, the fastest it drops.
b_1_X = fir1(100,wn,'low');
b_2_X = fir1(300,wn,'low');
b_3_X = fir1(500,wn,'low');

% filtered signal
a = 1; % fir filter does not have poles (transfer function denominator ...
    = 1)

figure()
plot(tVec_filterX(1:samples_plot),x(2,1:samples_plot)); hold on;
X_1 = filtfilt(b_1_X,a,x(2,1:samples_plot));
plot(tVec_filterX(1:samples_plot),X_1(1:samples_plot)); hold on;
X_2 = filtfilt(b_2_X,a,x(2,1:samples_plot));
plot(tVec_filterX(1:samples_plot),X_2(1:samples_plot)); hold on;
X_3 = filtfilt(b_3_X,a,x(2,1:samples_plot));
plot(tVec_filterX(1:samples_plot),X_3(1:samples_plot)); hold off;
```

```
grid on;
legend('original signal','filtered order 100', 'filtered order 300', ...
    'filtered order 500'); lgd.FontSize = 12;
title({'Filtered success kinematic vector x(t)'}, 'Fontsize', 14)
xlabel('Time [s]','FontSize', 12); ylabel('x(t)','FontSize', 12)

% We are going to consider the order 300 filter
[pks,locs] = findpeaks(X_2);
period = mean(diff(locs))*dt; % in seconds

%% TIME DELAY ESTIMATION --> through mutual information
% Generate result arrays
ground_truth_fr_prob_results = {};
time_delay_results = {};
mutual_info_results = {};
K_results = {};
ground_truth_spikes_results = {};


for neuron = 1:20
    spikes = spikes_success(neuron, 1:samples);

    %-----------
     % ESTIMATION OF ground truth firing probability
     % Creating Gaussian Kernel with the parameters that led to a ...
        minimum rmse
     % in window-exponential case in simulating data
     mean_value = 0; sigma = 1.9650;
     wide = 5*sigma;
     x_tunning = [-wide:.1:wide];
     y_tunning = normpdf(x_tunning, mean_value, sigma); % Gaussian kernel.
     ground_truth_fr_prob = conv2(spikes, y_tunning, 'same');
     ground_truth_fr_prob = ...
        ground_truth_fr_prob/max(ground_truth_fr_prob); % normalize so ...
        that maximum is probability 1
     % Make mean values to coincide
     mean_ratio = (sum(spikes)/length(spikes))/mean(ground_truth_fr_prob);
     if mean_ratio < 1
        ground_truth_fr_prob = ground_truth_fr_prob *  mean_ratio;
     end
     ground_truth_fr_prob_results{end+1} = ground_truth_fr_prob;

     % Plot the resulting estimation of firing probability
     plot_signal_spikes(tVec(1:samples_plot), spikes(1:samples_plot), ...
        ground_truth_fr_prob(1:samples_plot), 'Time [s]', 'Firing ...
        probability fr * dt', sprintf('NEURON %d, ground truth firing ...
        probability', neuron))

     %-------------------------------------------
     % Time delay ESTIMATION: mutual information with feature vector y(t)

     % p_spk
     p_spk = sum(spikes)/length(spikes);
     p_nospk = 1 - p_spk;

     mutual_info_v = [];
     for time_delay_try = [1:1:0.5*period/dt]
         x_delayed_try = [zeros(length(x(:,1)), time_delay_try), ...
            x(:,1:end - time_delay_try)];

         cropped_x_delayed_try = x_delayed_try(:,time_delay_try + 1:end);
         cropped_ground_truth_fr_prob = ...
            ground_truth_fr_prob(time_delay_try + 1:end);
        cropped_spikes = spikes(time_delay_try+1:length(spikes));

        %_____________________________________________
        % K ESTIMATION, through Spike Triggered Average (STA)
        cropped_x_delayed_try_spk = cropped_x_delayed_try(:, ...
            find(cropped_spikes));
        guessed_K = (cropped_x_delayed_try * ...
            cropped_x_delayed_try')^(-1) * ...
            sum(cropped_x_delayed_try_spk,2);

        % OPTION 2: using estimated fprob
        % guessed_K = (cropped_x_delayed_try * ...
            cropped_x_delayed_try')^(-1) * cropped_x_delayed_try * (eps ...
            + cropped_ground_truth_fr_prob'); %TRET LOG!!!

        %___________
        % Compute y = K*x
        cropped_y_delayed_try = guessed_K' * cropped_x_delayed_try;

        %___________
        % Compute mutual information
        % f_y
        [f_y, y_values] = kernel_smoothing(cropped_y_delayed_try);
        p_y = f_y(y_values);
        % p_y|spk
        [f_y_spk, foo] = ...
            kernel_smoothing(cropped_y_delayed_try(find(cropped_spikes)));
        p_y_spk = f_y_spk(y_values);
        [f_y_nospk, foo] = ...
            kernel_smoothing(cropped_y_delayed_try(find(cropped_spikes ...
            == 0)));
        p_y_nospk = f_y_nospk(y_values);


        % Mutual information
        spk_term = @(y) f_y_spk(y) .* p_spk .* log2(f_y_spk(y)/f_y(y));
        nospk_term = @(y) f_y_nospk(y) .* p_nospk .* ...
            log2(f_y_nospk(y)/f_y(y));
        MI_function = @(y) spk_term(y) + nospk_term(y);

        mutual_info = integral(MI_function, ...
            min(cropped_y_delayed_try), max(cropped_y_delayed_try));
        mutual_info_v = [mutual_info_v; time_delay_try mutual_info];
    end

    %___________
    % Low pass filtering mutual information

    Ts = dt; % Sampling Interval (s)
    Fs = 1/Ts;  % Sampling Frequency (Hz)
    Fn = Fs/2;  % Nyquist Frequency (Hz) -> maximal possible one to use ...
        as the cutoff frequency in the filter
    tVec_filterMI = 0:Ts:(length(mutual_info_v(:,2))-1)*Ts;

    % --> Which filter order should we choose?
    % We have to plot the power spectrum fft of the signal. Whenever we ...
        find a
    % tremendous drop of the power, we have the cutoff frequency Fc
    % After it, we will have to check visually if the order is suitable

    N = length(mutual_info_v(:,2));
    xdft = fft(mutual_info_v(:,2));
    xdft = xdft(1:N/2+1);
    psdx = (1/(Fs*N)) * abs(xdft).^2;
    psdx(2:end-1) = 2*psdx(2:end-1);
    freq = 0:Fs/length(mutual_info_v(:,2)):Fs/2;

    figure()
    y_fft = 10*log10(psdx);
    plot(freq(1:30), y_fft(1:30)) % plot only a few samples
    grid on
    title(sprintf('NEURON %d, Periodogram of mutual information using ...
        FFT', neuron), 'FontSize', 14)
    xlabel('Frequency (Hz)', 'FontSize', 12)
    ylabel('Power/Frequency (dB/Hz)', 'FontSize', 12)

    % We need to find the first minimum, that will be the cutoff frequency.
    periodogram = 10*log10(psdx);
    [peaks_periodogram, position_peaks_periodogram] = ...
        findpeaks(periodogram);
    [min_periodogram, freq_min_periodogram] = ...
        min(periodogram(1:position_peaks_periodogram(1)));
    Fc = freq_min_periodogram; % We obtain the cutoff frequency

    % 'fir1' uses a Hamming window to design an nth-order lowpass FIR
    % filter with linear phase (will be symmetric between 0 - pi and pi ...
        - 2*pi)
    % Normalized frequency wn is the cutoff frequency, frequency at ...
        which the normalized gain of the filter is -6 dB.
    % Below this frequency, the signal should be attenuated.
    % Hence we should set the cutoff frequency to a value equal to the ...
        Fc found.
    % As the function fir1 needs it to be normalized so that it lays ...
        between 0
    % and 1, since the Nyquist frequency is the biggest we could ...
        choose, we divide it
    % by it so that the maximum normalized value was 1. -> Fc/fn
    % We will use both order 10 and 20. We'll choose which order to choose
    % visually.

    wn = Fc/Fn;
    b_1_MI = fir1(10, wn,'low'); % wn given as multiples of pi
    b_2_MI = fir1(20, wn,'low');

    % filtered signal
    a = 1; % fir filter does not have poles (transfer function ...
        denominator = 1)

    figure()
    plot(tVec_filterMI,mutual_info_v(:,2)); hold on;
    MI_1 = filtfilt(b_1_MI,a,mutual_info_v(:,2));
    plot(tVec_filterMI,MI_1); hold on;
    MI_2 = filtfilt(b_2_MI,a,mutual_info_v(:,2));
    plot(tVec_filterMI,MI_2); hold off;
    xlabel('Time delay \Delta t [s]', 'FontSize', 12);
    ylabel('Mutual information', 'FontSize', 12);
    legend('original signal','filtered order 10', 'filtered order 20'); ...
        lgd.FontSize = 12;
    title(sprintf('NEURON %d, Filtered mutual information', neuron), ...
        'Fontsize', 14)

    % We will use the filter of order 10
    figure()
    freqz(b_1_MI,a,512);
    title(sprintf('NEURON %d, Lowpass filter frequency response (order ...
        10)', neuron), 'Fontsize', 14)
    % X axis shows multiples of pi/samples

    %-----------
    % We choose the parameters that give to a maximum mutual information
    % (already low filtered):

    [max_mutual_info, index] = max(MI_1);
    params = mutual_info_v(index,:);
    optimal_time_delay = params(1);
    optimal_mutual_info = params(2);

    % From this point on, we will be working with the cropped shifted
    % signals: shift signals by the time delay and only consider those ...
       samples after the time
    % delay as we are missing the signal information from the previous ones
     optimal_x_delayed = [zeros(length(x(:,1)), optimal_time_delay), ...
        x(:,1:end - optimal_time_delay)];
     cropped_optimal_x_delayed = optimal_x_delayed(:,optimal_time_delay ...
        + 1:end);
     cropped_ground_truth_fr_prob = ...
        ground_truth_fr_prob(optimal_time_delay + 1:end);
     cropped_spikes = spikes(optimal_time_delay+1:length(spikes));

     % Build optimal K
     cropped_optimal_x_delayed_spk = cropped_optimal_x_delayed(:, ...
        find(cropped_spikes));
     optimal_K = (cropped_optimal_x_delayed * ...
        cropped_optimal_x_delayed')^(-1) * ...
        sum(cropped_optimal_x_delayed_spk,2);
     % Compute y
     cropped_optimal_y_delayed = optimal_K' * cropped_optimal_x_delayed;
     optimal_y_delayed = [zeros(1, optimal_time_delay), ...
        cropped_optimal_y_delayed];

     plot_signal_spikes(tVec(1:samples_plot), spikes(1:samples_plot), ...
        optimal_y_delayed(1:samples_plot), 'Time [s]', sprintf('shifted ...
        y'), sprintf('NEURON %d, Shifted feature vector y(t - \Delta t) ...
        matched with spikes', neuron))

    % Save results
    time_delay_results{end+1} = optimal_time_delay;
    mutual_info_results{end+1} = optimal_mutual_info;
    K_results{end + 1} = optimal_K;
    ground_truth_spikes_results{end+1} = cropped_spikes;
end

%% TASK-RELATED NEURON SELECTION
% Only those neurons related with the activity will be considered for the
% analysis. We'll choose the ones before the huge drop in it.

optimal_mutual_infos = cell2mat(mutual_info_results);
[sorted_mutual_infos, sorted_index] = sort(optimal_mutual_infos, ...
    'descend');

figure()
plot(sorted_mutual_infos)
ax = gca;
ax.XTick = 1:size(sorted_index,2);
ax.XTickLabel = string(sorted_index);
grid on;
xlabel('neuron', 'FontSize', 12); ylabel('Optimal mutual information', ...
    'FontSize', 12)
title('Optimal mutual information neuron decreasing order', 'FontSize', 14)
related_neurons = sorted_index(1:10);

```

```
%% ENCODING MODEL ESTIMATION
% Generate results arrays
f_y_results = {};
y_values_results = {};
f_cropped_y_results = {};
cropped_y_values_results = {};

for neuron = related_neurons
    spikes = spikes_success(neuron, 1:samples);

    % Load estimations
    ground_truth_fr_prob = ground_truth_fr_prob_results{neuron};
    optimal_time_delay = time_delay_results{neuron};
    optimal_K = K_results{neuron};

    % From this point on, we will be working with the cropped shifted
    % signals: shift signals by the time delay and only consider those ...
       samples after the time
    % delay as we are missing the signal information from the previous ones
    optimal_x_delayed = [zeros(length(x(:,1)), optimal_time_delay), ...
       x(:,1:end - optimal_time_delay)];
    cropped_optimal_x_delayed = optimal_x_delayed(:,optimal_time_delay ...
       + 1:end);
    cropped_ground_truth_fr_prob = ...
       ground_truth_fr_prob(optimal_time_delay + 1:end);
    cropped_spikes = spikes(optimal_time_delay + 1:length(spikes));
    tVec = 0:dt:(length(cropped_spikes)-1)*dt;
    % Compute y
    cropped_optimal_y_delayed = optimal_K' * cropped_optimal_x_delayed;

     %-------------------------------------------
     % f() ESTIMATION through Bayes Theorem and Kernel density ...
        estimation (KDE)
     % f = p(spk, y)/p(y)

     % f_y
     [f_y, y_values] = kernel_smoothing(cropped_optimal_y_delayed);
     p_y = f_y(y_values);
     % p_y|spk
     [f_y_spk, foo] = ...
        kernel_smoothing(cropped_optimal_y_delayed(find(cropped_spikes)));
     p_y_spk = f_y_spk(y_values);
     [f_y_nospk, foo] = ...
        kernel_smoothing(cropped_optimal_y_delayed(find(cropped_spikes ...
        == 0)));
     p_y_nospk = f_y_nospk(y_values);

     %-----------
     % Plots
     %%Denominator p(Ks)
     normalization = p_y;

     figure()
     plot(y_values, normalization)
     xlabel('y', 'FontSize', 12); ylabel('p(y)', 'FontSize', 12)
     title(sprintf('NEURON %d, normalization density function p(y)', ...
        neuron), 'FontSize', 14)

     %%Numerator p(spk, Kx) = p(Kx|spk)p(spk)
     % p(Kx|spk)
     likelihood = p_y_spk;

     figure()
     plot(y_values, likelihood)
     xlabel('y', 'FontSize', 12); ylabel('f(y|spk)', 'FontSize', 12)
     title(sprintf('NEURON %d, likelihood density function f(y|spk)', ...
        neuron), 'FontSize', 14)

     % p(spk)
     prior = sum(cropped_spikes)/length(cropped_spikes); % scalar number

     %%Posterior probability
     posterior = (likelihood*prior)./(normalization + eps);
     if (max(posterior) - min(posterior) > 1000)
        posterior = (likelihood*prior)./(normalization + 0.001);
     end

     figure()
     plot(y_values, posterior)
     xlabel('y', 'FontSize', 12); ylabel('f(spk|y) estimation', ...
        'FontSize', 12)
     title(sprintf('NEURON %d, posterior f(spk|y), estimation of ...
        nonlinear function f(y)', neuron), 'FontSize', 14)
     f_y_estimation = griddedInterpolant(y_values, posterior);% Convert ...
        it to a function

     f_y_results{end+1} = f_y_estimation;
     y_values_results{end+1} = y_values;

     %-----------
     % Corrections
     % When using kernel_smoothing(), the estimated probability ...
        function is
     % smoothed in the edges (going to 0 in the nearby values of the
     % edges).
     % When computing the posterior, a division by '0' is produced and the
     % results are not accurate (at least graphically they make no sense).
     % That's why we cut the results from both sides an amount of samples
     % (y_precision) so that we can properly see the resulting posterior
     % function.

     % 1st cut
     [min_value, low_cut] = min(posterior(1:length(posterior)/3)); % ...
        minimum in 1st 1/3
     [max_value, high_cut] = ...
        max(posterior(length(posterior)/2:length(posterior))); % ...
        maximum in 2nd half
     high_cut = high_cut + length(posterior)/2 - 1;
     %[max_value, high_cut] = max(posterior); % maximum in general
     if low_cut > high_cut
        low_cut = 1;
     end

     %{
     if neuron == 14
        [foo, low_cut] = min(posterior(1:length(posterior)/10));  % ...
            minimum in 1st 1/10
        [max_value, high_cut] = max(posterior);
     elseif neuron == 4
        [foo, pos_max] = max(posterior);
        [foo, pos_min] = min(posterior(pos_max:pos_max+10));
        low_cut = pos_max + pos_min - 1;
        [foo, pos_max2] = max(posterior(low_cut:end));
        high_cut = low_cut + pos_max2 - 1;
     end
     %}

     cropped_posterior = posterior(low_cut:high_cut);
     cropped_y_values = y_values(low_cut:high_cut);
     cropped_f_y_estimation = griddedInterpolant(cropped_y_values, ...
        cropped_posterior); % Convert it to a function

     figure()
     plot(cropped_y_values, cropped_posterior); hold off;
     xlabel('y', 'FontSize', 12); ylabel('f(spk|y) estimation', ...
        'FontSize', 12)
     title(sprintf('NEURON %d, cropped posterior f(spk|y), estimation ...
        of nonlinear function f(y)', neuron), 'FontSize', 14)

     % Save iteration results
     f_cropped_y_results{end+1} = cropped_f_y_estimation;
     cropped_y_values_results{end+1} = cropped_y_values;
 end

 %% LINEAR Finetunning

 % Values obtained from fit exponential with  g = fittype('a*x+b');
 % and then fit = fit(cropped_y_values',cropped_posterior',g);
 %{
 g = fittype('a*x+b');
 linear_coeffs = [];
 for i = 1:length(related_neurons)
     cropped_y_values = cropped_y_values_results{i};
     cropped_posterior = f_cropped_y_results{i}(cropped_y_values);
     fit = fit(cropped_y_values',cropped_posterior',g)
     linear_coeffs = [linear_coeffs; fit.a fit.b]
 end
 %}
 load('linear_coeffs.mat')

 linear_coefs_results = {};
 linear_functions = {};

 for i = 1:length(related_neurons)
     neuron = related_neurons(i);
     y_values = y_values_results{i};
     posterior = f_y_results{i}(y_values);
     cropped_y_values = cropped_y_values_results{i};
     cropped_posterior = f_cropped_y_results{i}(cropped_y_values);

     a_estimated_linear = linear_coeffs(i,1);
     b_estimated_linear = linear_coeffs(i,2);

     linear_estimated = a_estimated_linear * cropped_y_values + ...
        b_estimated_linear;
     % Should make sure that it is 1 at its most
     unit_index = min(find(linear_estimated ≥ 1));
     if unit_index > 0 % if it is not empty
        linear_estimated = [linear_estimated(1:unit_index-1), ones(1, ...
            length(linear_estimated) - unit_index + 1)];
     end

     figure()
     plot(y_values, posterior); hold on;
     plot(cropped_y_values, linear_estimated, 'color', red); hold on;
     plot(cropped_y_values, cropped_posterior, 'color', blue, ...
        'lineWidth', 2); hold off;
     xlabel('y', 'FontSize', 12); ylabel('f(y)', 'FontSize', 12)
     legend('estimated f_a(  )', 'fitted exponential')
```

```
     title(sprintf('NEURON %d, exponential fit of nonlinear function ...
         f(y)', neuron), 'FontSize', 14)

    % Convert fitting into function
    linear_f = griddedInterpolant(cropped_y_values, linear_estimated); ...
        % Convert it into function

    linear_coefs_results{end + 1} = [a_estimated_linear, ...
        b_estimated_linear];
    linear_functions{end + 1} = linear_f;
end

%% EXPONENTIAL Finetunning

% Values obtained from fit exponential with  g = fittype('a*exp(b*x)+c');
% and then fit_i = fit(cropped_y_values',cropped_posterior',g);
%{
g = fittype('a*exp(b*x)+c');
exponential_coeffs = [];
for i = 1:length(related_neurons)
    cropped_y_values = cropped_y_values_results{i};
    cropped_posterior = f_cropped_y_results{i}(cropped_y_values);
    fit = fit(cropped_y_values',cropped_posterior',g)
    exponential_coeffs = [exponential_coeffs; fit.a fit.b fit.c]
end
%}
load('exponential_coeffs.mat')

exponential_coefs_results = {};
exponential_functions = {};

for i = 1:length(related_neurons)
     neuron = related_neurons(i);
     y_values = y_values_results{i};
     posterior = f_y_results{i}(y_values);
     cropped_y_values = cropped_y_values_results{i};
     cropped_posterior = f_cropped_y_results{i}(cropped_y_values);

     a_estimated_exponential = coeffs(i,1);
     b_estimated_exponential = coeffs(i,2);
     c_estimated_exponential = coeffs(i,3);

     exponential_estimated = a_estimated_exponential * ...
         exp(b_estimated_exponential * cropped_y_values) + ...
         c_estimated_exponential;
     % Should make sure that it is 1 at its most
     unit_index = min(find(exponential_estimated ≥ 1));
     if unit_index > 0 % if it is not empty
         exponential_estimated = ...
             [exponential_estimated(1:unit_index-1), ones(1, ...
             length(exponential_estimated) - unit_index + 1)];
     end

     figure()
     plot(y_values, posterior); hold on;
     plot(cropped_y_values, exponential_estimated, 'color', yellow); ...
         hold on;
     plot(cropped_y_values, cropped_posterior, 'color', blue, ...
         'lineWidth', 2); hold off;
     xlabel('y', 'FontSize', 12); ylabel('f(y)', 'FontSize', 12)
     legend('estimated f_a(  )', 'fitted exponential')
     title(sprintf('NEURON %d, exponential fit of nonlinear function ...
         f(y)', neuron), 'FontSize', 14)


    % Convert fitting into function
    exponential_f = griddedInterpolant(cropped_y_values, ...
       exponential_estimated); % Convert it into function

    exponential_coefs_results{end + 1} = [a_estimated_exponential, ...
       b_estimated_exponential, c_estimated_exponential];
    exponential_functions{end + 1} = exponential_f;

end

%% TUNING RESULTS

for i = 1:length(related_neurons)
    neuron = related_neurons(i);

    % Load original f_y
    y_values = y_values_results{i};
    posterior = f_y_results{i}(y_values);

    % Load cropped f_y
    cropped_y_values = cropped_y_values_results{i};
    cropped_posterior = f_cropped_y_results{i}(cropped_y_values);

    % Load finetunning functions and limits
    linear_f = linear_functions{i};
    linear_limits_i = linear_limits{i};
    linear_min_index = linear_limits_i(1); linear_max_index = ...
       linear_limits_i(2);

    exponential_f = exponential_functions{i};
    exponential_limits_i = exponential_limits{i};
    exponential_min_index = exponential_limits_i(1); ...
       exponential_max_index = exponential_limits_i(2);

    figure()
    plot(y_values, posterior); hold on;
    plot(cropped_y_values, linear_f(cropped_y_values)); hold on;
    plot(cropped_y_values, exponential_f(cropped_y_values)); hold on;
    plot(cropped_y_values, cropped_posterior,'LineWidth',2, 'color', ...
       blue); hold off;
    %plot(cropped_y_values(exponential_min_index:exponential_max_index), ...
       cropped_posterior(exponential_min_index:exponential_max_index),'LineWidth',2); .
       hold off;
    grid on;
    xlabel('y', 'FontSize', 12); ylabel('f(y)', 'FontSize', 12)
    %legend('Estimated f(y)', 'Linear fitting', 'Exponential fitting', ...
       'Linear estimation range','Exponential estimation range'); ...
       lgd.FontSize = 12;
    legend('Estimated f(y)', 'Linear fitting', 'Exponential fitting', ...
       'Estimation range'); lgd.FontSize = 12;
    title(sprintf('NEURON %d, linear and exponential fit of nonlinear ...
       function f(y)', neuron), 'FontSize', 14)

end

%% MODEL RESULTS

% Generate results arrays
predicted_fr_prob_results = {};
predicted_spikes_results = {};

```

```
for i = 1:length(related_neurons)
     ground_truth_fr_prob = ground_truth_fr_prob_results{i};

    %-----------------------------------------------------------------------
    % PREDICTION OF firing probability through predicted model
    % Load spikes and signal x
    neuron = related_neurons(i);
    spikes = spikes_success(neuron, 1:samples);
    optimal_time_delay = time_delay_results{neuron};
    optimal_mutual_info =  mutual_info_results{neuron};

    % Shift x by the time delay, only consider those samples after the time
    % delay as we are missing the signal information from the previous ones
    optimal_x_delayed = [zeros(length(x(:,1)), optimal_time_delay), ...
       x(:,1:end - optimal_time_delay)];
    cropped_optimal_x_delayed = optimal_x_delayed(:,optimal_time_delay ...
       + 1:end);
    cropped_spikes = spikes(optimal_time_delay+1:length(spikes));
    tVec = 0:dt:(length(cropped_spikes)-1)*dt;

    % Load model parameters
    K = K_results{i};
    f_y = f_y_results{i};
    y_values = y_values_results{i};

    linear_f_y = linear_functions{i};
    exponential_f_y = exponential_functions{i};

    %-----------
    % Compute feature space y = K * x
    cropped_predicted_y = K' * cropped_optimal_x_delayed;

    figure()
    plot(tVec(1:samples_plot), cropped_predicted_y(1:samples_plot))
    xlabel('Time [s]', 'FontSize', 12); ylabel('y = K*x estimation', ...
       'FontSize', 12)
    title(sprintf('NEURON %d, y = K*x', neuron), 'FontSize', 14)

    %-----------
    % Compute firing probability
    % The function f provides with the firing probability fr*dt= f(y)
    % Hence, we have to divide by dt the result to obtain the desired ...
       firing rate fr
    predicted_fr_prob = f_y(cropped_predicted_y);
    predicted_fr_prob_linear = linear_f_y(cropped_predicted_y);
    predicted_fr_prob_exponential = exponential_f_y(cropped_predicted_y);

    predicted_fr_prob_results{i} = [predicted_fr_prob; ...
       predicted_fr_prob_linear; predicted_fr_prob_exponential];

    figure()
    plot(tVec(1:samples_plot), predicted_fr_prob(1:samples_plot)); hold on;
    plot(tVec(1:samples_plot), ...
       predicted_fr_prob_linear(1:samples_plot)); hold on;
    plot(tVec(1:samples_plot), ...
       predicted_fr_prob_exponential(1:samples_plot)); hold on;
    plot(tVec(1:samples_plot), ...
       ground_truth_fr_prob(1:samples_plot),'color', dark_blue); hold off;
    xlabel('Time [s]', 'FontSize', 12); ylabel('\lambda = f(y) ...
       estimation', 'FontSize', 12);
    ylim([0, 1]);
    title(sprintf('NEURON %d, firing probability estimation', neuron), ...
       'FontSize', 14)
    legend('Predicted from model', 'Linear fit', 'Exponential ...
       fit','Estimated ground truth from reality'); lgd.FontSize = 12;

    % Generate spike train
    nTrials = 1;
    nBins = length(predicted_fr_prob);
    predicted_spikes = poissonSpikeGen(predicted_fr_prob, dt, nBins, ...
       nTrials);
    predicted_spikes_results{end+1} = predicted_spikes;
end

%% EVALUATION: Time Rescaling Theorem

KS_results = {};

for i = 1:length(related_neurons)
    neuron = related_neurons(i);

    % Load ground truth fprob(t)
    ground_truth_fr_prob = ground_truth_fr_prob_results{i};
    f_ground_truth_fr_prob = griddedInterpolant(ground_truth_fr_prob); ...
       % Convert it into function

    % Load fprob(y) approaches
    predicted_fr_prob_approaches = predicted_fr_prob_results{i};

    predicted_fr_prob = predicted_fr_prob_approaches(1,:);
    predicted_fr_prob_linear = predicted_fr_prob_approaches(2,:);
    predicted_fr_prob_exponential = predicted_fr_prob_approaches(3,:);
    f_predicted_fr_prob = griddedInterpolant(predicted_fr_prob); % ...
       Convert it into function
    f_predicted_fr_prob_linear = ...
       griddedInterpolant(predicted_fr_prob_linear); % Convert it into ...
       function
    f_predicted_fr_prob_exponential = ...
       griddedInterpolant(predicted_fr_prob_exponential);% Convert it ...
       into function

    % Load spikes, ground truth and predicted
    ground_truth_spikes = ground_truth_spikes_results{i};
    predicted_spikes = predicted_spikes_results{i};
    tk = find(predicted_spikes);

    %___________
    % Kolmogorov-Smirnov test
    opt = struct('DTCorrelation', 1, 'sampleRate', dt, 'neuron', ...
       neuron, 'color', dark_blue);
    KS = DBR_calc_approaches(predicted_fr_prob_approaches, ...
       ground_truth_spikes, opt);

    KS_results{end+1} = KS;
end


%___________
% KS metric comparison

KS = [];
KS_linear = [];
KS_exponential = [];

for i = 1:length(related_neurons)
    KS_approaches = KS_results{i};
    KS = [KS KS_approaches(1)];
    KS_linear = [KS_linear KS_approaches(2)];
    KS_exponential = [KS_exponential, KS_approaches(3)];
end

figure()
plot(KS); hold on;
plot(KS_linear); hold on;
plot(KS_exponential); hold off;
ax = gca;
%ax.XTick = 1:size(related_neurons);
ax.XTickLabel = string(related_neurons);
xlabel('Related neurons', 'FontSize', 12); ylabel('KS metric', ...
    'FontSize', 12);
title('KS metric comparison', 'FontSize', 14);
legend('f(y) predicted from model', 'Linear fit', 'Exponential fit');  ...
    lgd.FontSize = 12;

%% METRICS
experimental_metrics = KS_results;
save('experimental_metrics', 'experimental_metrics');
```

8.3.2 Depending functions

The following functions are those defined functions that were used in the main code.

```
function spikeMat = poissonSpikeGen_prob(fr_prob, nBins, nTrials)
% fr_prob: vector of instantaneous firing probabilities (fr(t)*dt)
% nBins: number of time steps (bins)
% nTrials: number of trials
spikeMat = rand(nTrials , nBins) < fr_prob; % Uniformly distributed
end
```

```
function [f_p, values] = kernel_smoothing(var)
set(gcf,'Visible','off')
histogram = histfit(var,100,'kernel');
fit = histogram(2);
values = fit.XData; p = fit.YData;
p = p/length(var);

f_p = griddedInterpolant(values, p, 'linear', 'nearest'); % Convert it ...
    to a function
end
```

```
function [ Z, U, xAxis, KSSorted, ks_stat ] = computeKSStats( ...
    spikeTrain, lambda, opt )
% COMPUTEKSSTATS Compute related KS statistics.
%   Input:
%       spikeTrain    - 1 * #timeslot
%       lambda        - 1 * #timeslot
%       opt           - need opt.sampleRate, opt.DTCorrelation
%   Output:
%       Z: rescaled spike times
%       U: Zjs tranformed to be uniform(0,1)
%       xAxis: x-axis of the KS plot
%       KSSorted: y-axis of KS plot
%       ks_stat: the KS statistic. Maximum deviations from the 45
%               degree line for each conditional intensity function.

if (opt.DTCorrelation == 1)
    pk = lambda .* (1/opt.sampleRate);
    pk = max(pk, 1e-10);
    minDim = min(length(spikeTrain), length(lambda));
    pk = pk(1:minDim);
    spikeTrain = spikeTrain(1:minDim);

    pk = nanmin(nanmax(pk, 0), 1);
    intValues = ksdiscrete(pk, spikeTrain, 'spiketrain');
    intValues = intValues';
else
    pk = lambda .* (1/opt.sampleRate);
    pk = max(pk, 1e-10);
    minDim = min(length(spikeTrain), length(lambda));
    pk = pk(1:minDim);
    spikeTrain = spikeTrain(1:minDim);

    spikes = find(spikeTrain ≠ 0);
    spikes = [1 spikes'];
    intValues = zeros(1, length(spikes)-1);

    for spikeIdx = 1:length(spikes) -1
        accum = sum(pk(spikes(spikeIdx):spikes(spikeIdx+1)));
        intValues(1, spikeIdx) = accum;
    end
end

Z = intValues;
U = 1 - exp(-Z);
KSSorted = sort(U, 'ascend');
N = size(KSSorted, 2);
if (N ≠ 0)
    xAxis = (([1:N]-.5)/N);
    ks_stat = max(abs(KSSorted - (([1:N]-.5)/N)));
end

end


function [rst,varargout] = ksdiscrete(pk,st,spikeflag)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ksdiscrete.m
% written by Rob Haslinger, December 2009
%
% This function performs time rescaling of ISIs based upon the discrete
% time version of the time rescaling theorem as described in Haslinger,
% Pipa and Brown (2010?).  This method corrects for biases in the KS plot
% caused by the temporal discretization.
%
% This function can be called in two ways
%
% 1) input the discrete time conditional probabilities "pk"  where ...
    0≤pk≤ 1
% and the spike train "spiketrain" which has elements either equal to 0 (no
% spike) or 1 (spike). There is also a flag 'spiketrain' to indicate that
% it is the full spike train.
%
% [rst,rstsort,xks,cb,rstoldsort] = ksdiscrete(pk,spiketrain,'spiketrain')
%
% 2) input the discrete time conditional probabilities "pk" and a list of
% the indicies "spikeind" of the bin indicies that the spikes are ...
   locaed in.
% There is also a flag 'spikeind' to indicate that the indicies are
% being given, not the full spike train
%
% [rst,rstsort,xks,cb,rstoldsort] = ksdiscrete(pk,spikeind,'spikeind');
%
% required output:
%
% rst : a vector of unsorted uniformly distributed rescaled times. This is
% the only output that is required.
%
% optional output, given in the order they appear in the list function
% outputs :
%
% rstsort : a vector of rescaled times sorted into ascending order
% xks : a vector of x axis values to plot the sorted rescaled times against
% cb : the value of the 95% confidence bounds
% rstoldsort : a vector of sorted rescaled times done without the discrete
% time correction
%
% To make a KS plot one would do
% plot(xks,rstsort,'k-');
% hold on;
% plot(xks,xks+cb,'k--',xks,xks-cb,'k--');
%
% To make a Differential KS plot one would do
% plot(xks,rstsort-xks,'k-');
% hold on;
% plot(xks,zeros(length(xks))+cb,'k--',xks,zeros(length(xks))-cb);
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Start with determining the inputs and some basic input error checking

    if nargin < 3 || nargin > 3
        error('Number of input arguments must be equal to 3');
    end

    % make pk into a column vector;

    [m1,m2]=size(pk);
        if (m1 ≠1 && m2 ≠1)
            error('pk must be a vector'); end
        if (m2>m1)
            pk=pk'; end
        [m1,m2]=size(pk);

    % make sure pk's are within [0,1]
    index=find(pk<0);
    if isempty(index) ≠1
        error('all values for pk must be within [0,1]');
    end
    index=find(pk>1);
    if isempty(index) ≠1
        error('all values for pk must be within [0,1]');
    end
    clear index;

    % make column vector of spike indicies

    if strcmp(spikeflag,'spiketrain') % spike train input

        [n1,n2]=size(st);
          if (n1 ≠1 && n2 ≠1); error('spike train must be a vector'); end
        if (n2>n1); st=st'; end
          [n1,n2]=size(st);
        if m1 ≠ n1; error('pk and spike train must be same length'); end

        spikeindicies=find(st==1);

        Nspikes=length(spikeindicies);

    elseif strcmp(spikeflag,'spikeind') % spike index input

        [n1,n2]=size(st);
          if (n1 ≠1 && n2 ≠1); error('spike indicies must be a ...
              vector'); end
        if (n2>n1); st=st'; end

        spikeindicies=unique(st);
        Nspikes=length(spikeindicies);

    end

    % check that those indicies are in [1:length(pk)];

    if(isempty(spikeindicies))
        rst = pk;
        return;
    end
    if spikeindicies(1)<1
         error('There is at least one spike with index less than 0');
    end
    if spikeindicies(Nspikes)>length(pk)
         error('There is at least one spike with a index greater than ...
             the length of pk');
    end

    % error checking done

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Now do the actual discrete time KS test
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % initialize random number generator
    s = RandStream('mt19937ar','Seed', sum(100*clock));
    RandStream.setGlobalStream(s);
    %rand('twister',sum(100*clock));

    % make the qk's

    qk=-log(1-pk);

    % make the rescaled times

    rst=zeros(Nspikes-1,1);
    rstold=zeros(Nspikes-1,1);

    for r=1:Nspikes-1

        total = 0;
        ind1=spikeindicies(r);
        ind2=spikeindicies(r+1);

        total=total+sum(qk(ind1+1:ind2-1));

        Δ=-(1/qk(ind2))*log(1-rand()*(1-exp(-qk(ind2))));

        total=total+qk(ind2)*Δ;

        rst(r)=total;

        rstold(r)=sum(qk(ind1+1:ind2));

    end

%     rst=1-exp(-rst);
%     rstold=1-exp(-rstold);

    % optional outputs

    rstsort=sort(rst);
    varargout{1}=rstsort;

    inrst=1/(Nspikes-1);
    xrst=(0.5*inrst:inrst:1-0.5*inrst)';
    varargout{2}=xrst;

    cb=1.36*sqrt(inrst);
    varargout{3}=cb;

    varargout{4}=sort(rstold);
end
```

```
function y_approaches = DBR_calc(fr_prob_approaches, spk,opt, exponential)
dark_blue = [0 0.286274522542953 0.47843137383461];
color_v = [0, 0.4470, 0.7410; 0.8500, 0.3250, 0.0980; 0.9290, 0.6940, ...
    0.1250];

y_approaches = [];
figure()
for i = 1:length(fr_prob_approaches(:,1))
    fr_prob = fr_prob_approaches(i,:);

    xAxis = [];
    KSSorted = [];
    for ks_trail = 1:20
    [ Z, U, xAxis(ks_trail,:), KSSorted(ks_trail,:), ...
        KSValue_GLM(ks_trail) ] = computeKSStats( spk', fr_prob'* ...
        opt.sampleRate, opt );
    end

    xAxis = mean(xAxis);
    KSSorted = mean(KSSorted);
    KSValue_GLM = mean(KSValue_GLM);
    y = KSValue_GLM/1.36*sqrt(sum(spk == 1));

    y_approaches = [y_approaches, y];

    if isfield(opt,'exponential') & opt.exponential == 1 & i == 2 ...
        color_line = color_v(3,:);
    else color_line = color_v(i,:);
    end

    plot(xAxis, KSSorted ,'linewidth', 2, 'color', color_line); hold on;
end

plot(((1:sum(spk == 1)))/sum(spk == 1), (1:sum(spk == 1))/sum(spk == ...
    1), 'Color', dark_blue, 'linewidth', 1);
plot((1:sum(spk == 1))/sum(spk == 1), (1:sum(spk == 1))/sum(spk == ...
    1)+1.36/sqrt(sum(spk == 1)),'--', 'Color', dark_blue, 'linewidth', 0.5);
plot((1:sum(spk == 1))/sum(spk == 1), (1:sum(spk == 1))/sum(spk == ...
    1)-1.36/sqrt(sum(spk == 1)),'--', 'Color', dark_blue, 'linewidth', 0.5);
axis([0 1 0 1]); hold off;
if isfield(opt,'neuron') title(sprintf('NEURON %d, KSS regression ...
    plot', opt.neuron), 'FontSize', 14)
else title(sprintf('KSS regression plot'), 'FontSize', 14)
end

if isfield(opt,'exponential')
    if opt.exponential == 1 legend('f(y) predicted from model', ...
        'Exponential fit', 'Ideal 45   line');  lgd.FontSize = 12;
    else legend('f(y) predicted from model', 'Linear fit', 'Ideal 45   ...
        line');  lgd.FontSize = 12;
    end
else legend('f(y) predicted from model', 'Linear fit', 'Exponential ...
    fit', 'Ideal 45   line');  lgd.FontSize = 12;
end
end
```

Bibliography

R. Barbieri, R. Kass, V. Ventura., and L. M Frank. The time-rescaling theorem and its application to neural spike train data analysis. *Neural Computation*, 14(2):325 – 346, 2002. URL https://pubmed.ncbi.nlm.nih.gov/11802915/.

A.E. Brockwell, A.L. Rojas, and R.E. Kass. Recursive bayesian decoding of motor cortical signals by particle filtering. *J. Neurophysiol.*, pages 1899–1907, 2004. URL ''.

S. Chen, X. Zhang, X. Shen, Y. Huang, and Y. Wang. Tracking fast neural adaptation by globally adaptive point process estimation for brain-machine interface. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 29:1690–1700, 2021. URL 'https://ieeexplore.ieee.org/document/9517295/authors#authors'.

T.M. Cover and J.A. Thomas. *Elements of Information Theory.* New York: Wiley, 1991.

M. S. Gazzaniga, R. B. Ivry, and G. R. Mangun. *Cognitive Neuroscience: the biology of mind.* Fourth Edition. W. W. Norton Company, 2014.

A.P. Georgopoulos, A.B. Schwartz, and R.E. Kettner. Neuronal population coding of movement direction. *Science*, 233:1416–1419, 1986.

D. Hegger. Poisson model of spike generation. 2000.

E. R. Kandel, J. H. Schwartz, and T. M. Jessell. *Principles of Neural Science.* Fourth Edition. McGraw-Hill, 2000.

C. McDonald, S. Westcott-McCoy, M. Weaver, J. Haagsma, and D. Kartin. Global prevalence of traumatic non-fatal limb amputation. *Prosthetics and Orthotics International*, 45:105 – 114, 2021. URL 'https://journals.lww.com/poijournal/Abstract/2021/04000/Global_prevalence_of_traumatic_non_fatal_limb.4.aspx'.

L. Paninski, M.R. Fellows, N.G. Hatsopoulos, and J.P. Donoghue. Spatiotemporal tuning of motor cortical neurons for hand position and velocity. *J. Neurophysiol.*, pages 515–532, 2004a. URL ''.

L. Paninski, S. Shoham, M.R. Fellows, N.G. Hatsopoulos, and J.P. Donoghue. Superlinear population encoding of dynamic hand trajectory in primary motor cortex. *J. Neurosci.*, pages 8551–8561, 2004b. URL ''.

A.B. Schwartz, D.M. Taylor, and S.I.H. Tillery. Extraction algorithms for cortical control of arm prosthetics. *Curr. Opin. Neurobiol*, 11:701–708, 2001.

J.J. Shih, D.J. Krusienski, and J.R. Wolpaw. Brain-computer interfaces in medicine. *Mayo Clinic Proceedings*, 87:268 – 279, 2012. URL 'https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3497935/'.

S. Shoham, L.M. Paninski, M.R. Fellows, N.G. Hatsopoulos, J.P. Donoghue, and A. Normann. Statistical encoding model for a primary motor cortical brain–machine interface. *IEEE Trans. Biomed. Eng.*, pages 1312–1322, 2005. URL ''.

E.P. Simoncelli, L. Paninski, J. Pillow, and O. Schwartz. Characterization of neural responses with stochastic stimuli. *The Cognitive Neurosciences*, pages 327–38, 2004. URL ''.

W. Truccolo, U. T. Eden, M. R. Fellows, J. P. Donoghue, and E. N. Brown. A point process framework for relating neural spiking activity to spiking history, neural ensemble, and extrinsic covariate effects. *J Neurophysiol*, 93:1074 – 1089, 2005.

Y. Wang and J.C. Principe. Instantaneous estimation of motor cortical neural encoding for online brain–machine interfaces. *Journal of neural engineering*, 7(5), 2010. URL 'https://doi.org/10.1088/1741-2560/7/5/056010'.

Y. Wang, J. Sanchez, and Principe J.C. Information theoretical estimators of tuning depth and time delay for motor cortex neurons. *3rd Int. IEEE/EMBS Conf. CNE '07*, pages 502–505, 2007. URL ''.

Y. Wang, J.C. Principe, A. R: C. Paiva, and J. Sanchez. Sequential monte carlo point process estimation of kinematics from neural spiking activity for brain machine interfaces. *Neural Comput.*, 21:2894–2930, 2009.

R.S. Williamson, M. Sahani, and J.W. Pillow. The equivalence of information-theoretic and likelihood-based methods for neural dimensionality reduction. 11, 2013. URL https://www.researchgate.net/publication/255965833_The_Equivalence_of_Information-Theoretic_and_Likelihood-Based_Methods_for_Neural_Dimensionality_Reduction.

W. Wu, Y. Gao, E. Bienenstock, J.P. Donoghue, and M.J. Black. Bayesian population decoding of motor cortical activity using a kalman filter. *Neural Comput.*, pages 80–118, 2006. URL ''.

B.M. Yu, C. Kemere, G. Santhanam, A. Afshar, S.I. Ryu, T.H. Meng, M. Sahani, and K.V. Shenoy. Mixture of trajectory models for neural decoding of goal-directed movements. *J. Neurophysiol.*, 97:1312–1322, 2007.