University of BRISTOL

Faculty of Engineering
DEPARTMENT OF COMPUTER SCIENCE

Calculating mutual information in metric spaces

- Bachelor of Science dissertation -

Kristian T. Krastev
Supervised by Dr. Conor J. Houghton

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Kristian Tonev Krastev, May 2016

Contents

Abstract

Mutual information tells us how much the uncertainty associated with one variable is reduced by the knowledge of another one [18, 2]. It is a useful tool for quantifying relationships and has many applications to statistical modelling - for example in various types of clustering where one aims to maximise the dependencies within a partition. Typically, like most information-theoretic quantities, mutual information is estimated for variables that are either discrete or take values in a coordinate space. However, many important data types have distance metrics or measures of similarity, but no coordinates on them. The spaces induced by these metrics are not manifolds and their integration measure is not so obvious. Datasets of this type are collected from electrophysiological experiments in neuroscience and gene expression data in genetics and biochemistry, but also in other fields like image analysis and data retrieval.

The purpose of this project is to implement and test an estimator for calculating mutual information between data, where one or both variables come from a metric space without coordinates. The model estimator itself has been described in [27], but has not been implemented and tested until now. It aims to provide a simple approach to the Kozachenko- Leonenko estimator for mutual information [5], that extends it in order to apply to the broader class of metric-space data.

The model is particularly relevant to neuroscience because it addresses the problem of calculating information theory quantities from the similarity measures on the space of spike trains. This is the application that motivates it and will serve as the main framework for producing the data, used to test and adjust it.

Application software will be developed in Python. It is the preferred choice over MATLAB for example, because it is a modern high-level language with nice syntax and structure that supports various coding styles. It also has libraries and packages which are analogous to the MATLAB functionality relevant to the task - NumPy, SciPy, PyLab, matplotlib and so on. In addition to this, it offers interfaces to other tools for computational neuroscience and mathematics, which could be useful for the future development of the project.

The implementation will apply the suggested model to fictive spike-train data generated using a stochastic model and data related to it that is produced by a deterministic neural simulation. The proposed thesis will aim to investigate the model's correctness and evaluate its performance on various tasks. The analysed results will serve its future application to real experimental neuroscientific data.

1 Background

1.1 Problem outline

Estimating mutual information in metric spaces

Information theory is the domain of applied mathematics that defines models and techniques for quantisation, storage and communication of information. It is based on probability theory and statistics and is therefore traditionally applied in spaces where an intuitive integration measure can be used in order for probability mass or density to be easily estimated. Typically these are coordinate spaces such as discrete vector spaces or integrable manifolds.

The model under investigation tackles the problem of estimating information-theoretic quantities, namely *mutual information* and *relative entropy*, for variables taking values in the broader class of *metric spaces*. These are sets with distances between all members defining a metric, without it necessarily invoking a coordinate system. An alternative approach is taken by defining a measure on metric spaces as the probability mass contained within a region.

This chapter introduces the theoretic and scientific basis underlying the model and its application. First, the standard notions of information theory are introduced, along with *distance metrics* and *metric spaces*. Next, the suggested approach to the outlined problem is described, and the metrics relevant to its application in the context of neuroscientific data are explored. Finally, the basic computational models for neural voltage dynamics are taken account of, as well as the tools and techniques used to simulate them numerically.

Measuring information

Entropy is the central and fundamental notion in information theory [18]. It envelops the properties one would require of an information measure. In particular, it quantifies the uncertainty associated with the outcome of a chance event, mathematically modelled by a random variable. That is, it gives a measure of the average amount of information necessary to describe the random variable. First coined by Shannon [2], the *entropy* or *average self-information* of a random variable X, taking values in a set of outcomes $\mathcal{X}$, is defined as the negative logarithms of distinct- outcome probabilities $\{p(x)|x \in \mathcal{X}\}$ summed and weighted over their probability distribution. This is given by the formula:

$$H(X) = -\sum_{x\in\mathcal{X}} p(x)\log_2 p(x) \tag{1}$$

Taking a logarithm to the base two, entropy measures the minimal expected size of outcome- encoding in binary bits, though this can easily be scaled to other units using the properties of logarithms. Bits can be thought of as the number of binary (yes or no) questions it would take on average to determine the outcome of an event modelled by a random variable if the questions are ordered efficiently - in descending order of outcome probability.

Shannon's proof of correctness of the definition above relies on the necessity for it to possess three key properties. Firstly, the information measure of a variable must be continuous in the probability of its outcome - ranging in $[0, 1]$. Secondly, if the variable is uniformly distributed - which means that there is maximum uncertainty or *informativeness* associated with it - there must be a monotonically increasing relationship between the number of possible values it takes and its entropy. And third, if a choice among the set of outcomes is split into successive choices among subsets, the entropy of the whole set should equal the weighted sum of entropies of the splits - for example:

$$H(\frac{1}{2},\frac{1}{3},\frac{1}{6}) = H(\{\frac{1}{2}\},\{\frac{1}{3},\frac{1}{6}\}) + \frac{1}{2}H(\{2\cdot\frac{1}{3}\},\{2\cdot\frac{1}{6}\}) = H(\frac{1}{2},\frac{1}{2}) + \frac{1}{2}H(\frac{2}{3},\frac{1}{3}) \tag{2}$$

Being defined in terms of the probability distribution of the variable of interest, the notion of entropy can be extended to reflect joint (eq. 3) and conditional (eq. 5) probability distributions.

If X and Y are two events and $p(x, y)$ is the probability of the joint occurrence of their outcomes x and y, the entropy of the joint event is

$$H(X,Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log_2 p(x,y) \tag{3}$$

with the property

$$H(X,Y) \leq H(X) + H(Y). \tag{4}$$

The above is an equality if and only if X and Y are independent, in other words iff $p(x,y) = p(x)p(y)$.

For every outcome y of Y there is a conditional probability $p(x|y)$ that the outcome of X is x. The average of the entropy of X over outcomes of Y, weighted by the probability of each outcome y, gives the conditional entropy of X given Y:

$$\begin{aligned} H(X|Y) &= \sum_{y\in\mathcal{Y}} p(y)H(X|Y=y) = -\sum_{y\in\mathcal{Y}} p(y) \sum_{x\in\mathcal{X}} p(x|y)\log_2 p(x|y) \\ H(X|Y) &= -\sum_{y\in\mathcal{Y}}\sum_{x\in\mathcal{X}} p(x,y)\log_2 p(x|y) \end{aligned} \tag{5}$$

This quantity measures the uncertainty associated with X on average if the outcome of Y is known.

The quantity of interest to this project - *mutual information* - is based on the concept of *relative entropy*, also known as *Kullback-Leibler divergence*. Relative entropy gives a measure of the distance between two probability distributions. It is defined as the mean weighted logarithm of their *likelihood ratio* of the distributions:

$$D(p||q) = \sum_{x\in\mathcal{X}} p(x)\log_2 \frac{p(x)}{q(x)} \tag{6}$$

In the context of coding theory, the relative entropy $D(p||q)$ measures the inefficiency of encoding a random variable resulting from the assumption that its distribution is q while it is in fact p. That is, encoding the outcome of X would take on average $D(p||q) = H(q) - H(p)$ extra bits resulting in $H(p) + D(p||q)$ bits instead of $H(p)$ bits. Although it is often called *Kullback-Leibler distance* it is not a true distance metric since it does not symmetric and it does not satisfy the triangle inequality.

Mutual information is defined as the relative entropy between the joint distribution of X and Y - $p(x,y)$ and the product distribution $p(x)p(y)$:

$$I(X;Y) = \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log_2 \frac{p(x,y)}{p(x)p(y)} \tag{7}$$

It measures the amount of information one variable contains about another one or in other words, how much knowing one variable decreases uncertainty about another one. The mutual information between two variables equals zero if and only if they are statistically independent. This makes it a more powerful tool for establishing a relationship between two variables than *correlation* because it is capable of describing a relationship even if it is not linear or monotonic.

One last information-theoretic concept is explored here that will be used in the context of testing the mutual information estimator later on. Namely, this is the *data-processing inequality* according to which there is no way to increase the information content shared between two signals by means of physical manipulation [18]. In other words this means that information is generally lost but never gained when transmitted through a noisy channel [25].

This in turn relies on the idea of what in probability theory is called a *Markov chain.* Random variables X, Y, Z are said to form a Markov chain $X \to Y \to Z$ if the conditional probability distribution of Z depends only and Y and it is independent of the distribution of X, that is the joint probability mass function of the three is given by

$$p(x,y,z) = p(x)p(y|x)p(z|y). \tag{8}$$

This implies X and Z are conditionally independent given Y:

$$p(x,z|y) = \frac{p(x,y,z)}{p(y)} = \frac{p(x,y)p(z|y)}{p(y)} = p(x|y)p(z|y). \tag{9}$$

The data processing inequality states that

$$X \to Y \to Z \Rightarrow I(X;Y) \geq I(X;Z) \tag{10}$$

This is because X and Z are conditionally independent and thus $I(X;Z|Y) = 0$ - there is no information about X that Z contains and Y does not. The Markov chain is usually modelling a random process that goes from one state to another with the transition depending only on the preceding state. Although not directly related to the task at hand, this concept will be useful when analysing data recorded from the activity of consecutively connected neurons. In particular the mutual information between neural responses should provide some insight about the neural network's connectivity.

Metrics and similarity measures

Given a space $\mathcal{X}$ a *distance metric* is a function $Dis : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, such that for $\forall x, y, z$ it has the properties:

1. **Positive:** if $x \neq y \Rightarrow Dis(x,y) > 0$
2. **Symmetric:** $Dis(x,y) = Dis(y,x)$
3. **Triangle inequality:** $Dis(x,z) \leq Dis(x,y) + Dis(y,z)$

Perhaps the most familiar distance metric is *Euclidean distance* - the ordinary "straight line" distance between two points in a vector space. It is defined as the square root of the sum of squared differences between all vector components in an N-dimensional space:

$$Dis_2(x,y) = \left(\sum_{i=1}^{N} |x_i - y_i|^2 \right)^{1/2} \tag{11}$$

The notion is generalised by *Minkowski distances* as a set of L_p*-norms* for $p \geq 1$ - these are guaranteed to be the distance metrics for vector spaces which comply with the triangle inequality:

$$Dis_p(x,y) = \left(\sum_{i=1}^{N} |x_i - y_i|^p \right)^{1/p} \tag{12}$$

Euclidean distance is thus the L_2*-norm*, whereas the L_1*-norm* - where one "moves" only in orthogonal directions parallel to the coordinate axes - is known as *Manhattan distance*.

Numerous statistical and distance-based models employ Minkowski distance metrics in probability estimation, function regression, clustering and classification tasks as they are intrinsically compatible with data that can be naturally represented in a coordinate vector space. There exist however many other types of distance metrics that can be applied without necessarily representing the domain of interest as a vector space. Such metrics can for example be very useful for measuring *dissimilarity* between patterns in string matching.

In information theory and coding theory, *Hamming distance* counts the number of positions (components) in which two vectors differ. It is of particular importance to string matching, as an expression of dissimilarity between strings based on their number of differing symbols (or bits). Alternatively, it be thought of as the minimum number of symbol swaps, or errors, required to transform one string into another. Although Hamming distance is not a true L_p norm, it does satisfy the triangle inequality and is indeed a true distance metric. Under the assumption that $0^0 = 0$ and $x^0 = 1$ for $x \neq 0$, it can be formalised as the L_0*- norm*:

$$Dis_0(x,y) = \sum_{i=1}^{N} (x_i - y_i)^0 = \sum_{i=1}^{N} I[x_i = y_i] \tag{13}$$

This kind of metrics can in turn be generalised to the class of *edit-distance* metrics, by considering different cost functions for insertion, deletion and substitution, possibly tailored to a particular problem or purpose. In the case of spike trains, there exist a range of edit-distance metrics which take into account structural characteristics based on spike timing, such as repeating patterns, that are considered to carry the semantics of neural signalling. This kind of meaning cannot always be conveyed by metrics relying on mapping spike trains to a vector space. On the other hand, there is also another type of spike-train metrics, which take a different approach to achieve the same task - by mapping spike trains into the vector spaces of continuous functions. Both of these induce non-coordinate spaces. They will be discussed in detail in the next section.

The problem at hand arises from the use of these metrics. A measure of dissimilarity between any two members of the set of possible spike trains together with the set itself, form a metric space. A method is needed for estimating information-theoretic quantities, and therefore probabilities, in a metric space. There is no meaningful coordinate system in such metric spaces but this does not rule out alternative methods for estimating probability densities. The proposed model [27] takes the approach of the Kozachenko-Leonenko [5] entropy estimator, which estimates local densities based on k-nearest-neighbour statistics. It modifies and extends it by using probability as a measure of volumes in neighbourhoods, instead of depending on a space dimensionality for this purpose, and thus addresses the problems associated with measuring information in the metric spaces of spike trains.

Being able to estimate such information theory quantities is important as it can aid the study and development of neurodynamics coding theories and serve as a useful tool for quantifying relationships between neural activity. The relevance of such a model however is not constrained exclusively to the metric spaces of spike trains as there are many other contexts in which non- coordinate distance metrics are used.

1.2 Approach

Estimating information: possible routes

One of the principal objectives of neuroscience is to discover the mechanisms used by nerve cells to communicate information. It is commonly accepted that neural information processing relies on the transmission of sequences of stereotypical events. These are spikes in the potential difference across a neuron's membrane, taken as a function of time. They are usually referred to as *spikes* or *action potentials* and a sequence of their times of occurrence is what forms *spike trains.* The biophysical description of the process behind spike production will be discussed in the next section.

Computational neuroscience is interested in investigating the structural properties of spike trains from a statistical point of view in order to identify the features conveying information. Although this is an open problem, it is known that these include not only obvious features, such as the number of spikes fired over a time period, but also precise spike timing and more subtle features dependent on it, like patterns of intervals or action potentials - both over time and across a population [16]. The theoretical framework employed to unravel these statistical properties is grounded in the foundations of information theory. Being able to quantify the information conveyed between neurons, combined with the appropriate experimental techniques and stimulations, can be used to determine the key statistical features contained in spike trains [16].

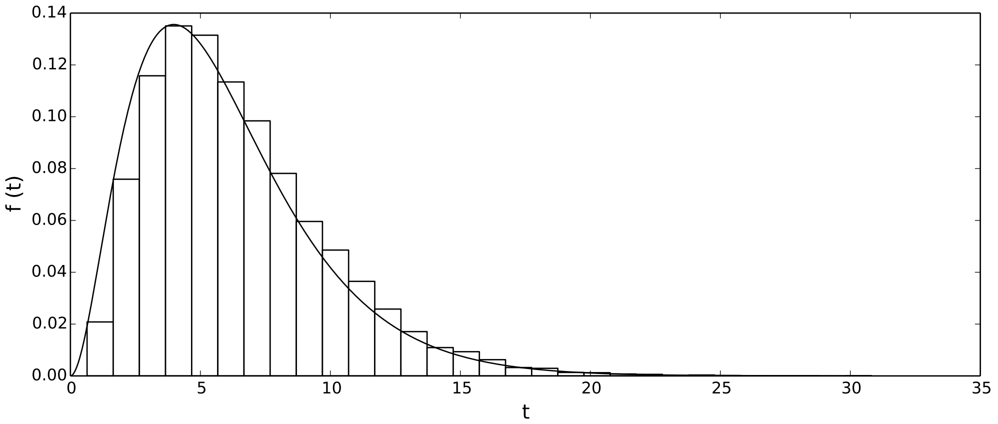


Figure 2: Example of a typical histogram method used to approximate a continuous gamma function at a discrete sampling rate subject to some quantisation. The recorded values can then be binned across the domain of the sampled variable to approximate probability densities.

The most straight-forward technique for information estimation on continuous signals sampled at some frequency is to break up the domains of the variable(s) of interest into partitions of finite size referred to as "bins", and approximate the standard analytic definition discretely, using the fractions of sample points taking values in the ranges of these bins as local density estimates [17]. A similar kind of binning strategy can also be used for embedding spike trains into a discrete space. The temporal window of observation of a neural response is subdivided into discrete time slices. Each of these bins can then correspond a distinct dimension in the vector space of possible binary words provided that the bins are small enough to contain no more than one spike. This is the traditional direct method for working out information quantities on spike trains developed by Bialek and Strong *et al.* [11].

The problem with the above approach is that the bins must be very narrow - down to a millisecond [9] - in order to capture spike timing with good precision. This requires estimation of a tremendous amount of response probabilities as the number of possible spike distributions increases exponentially with the model's resolution - for example $\approx 2.23 \times 10^{491}$ possible words over just a quarter of a second at 1 ms precision. As the bias of such estimates is roughly proportional to the number of possibilities, it takes unrealistically large amounts of sample data in order for them to be accurate [16]. In fact, this is one of the main difficulties associated with measuring information-theoretic quantities in neuroscience. Furthermore high-dimensional spaces tend to be very sparse, which makes the issue even bigger.

The sampling problem is generally addressed by non-parametric estimation methods. The Kozachenko-Leonenko entropy estimator [5] is one such method which addresses the issue for data taking values in vector spaces. It is described as *"a little-known asymptotically unbiased 'binless' estimator of differential entropy"* [16]. The basic idea behind this method is to use k-nearest neighbour distance statistics to estimate local densities, as first proposed in [4]. It has substantial computational advantages for estimating the entropy of a continuous distribution in a Euclidean vector space and is guaranteed to be unbiased in the limit of infinite sample data [5]. In [16] the Kozachenko-Leonenko method is adopted for the estimation of the rate of transmitted information through a neural "channel", which depends on entropy estimation. In this paper however a dimensionality assumption is made, which is resolved through a rather unnatural foliation of the space of spike trains - it is split into two components: a continuous one representing spike timing and a discrete one expressing the number of spikes in a neural response. It is demonstrated that for a limited-sized sample data the method outperforms considerably traditional estimators relying on binning.

Although this approach to entropy estimation dates back to [4], for a long time it had not been adapted to estimate mutual information. A simple way to do this would be to make use of the chain rule for entropy and the definition of mutual information:

Chain rule

$$H(X,Y) = H(X) + H(Y|X) = H(Y) + H(X|Y) \tag{14}$$

Mutual Information

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) \tag{15}$$

From the above it follows that mutual information can be expressed as:

$$I(X;Y) = H(X) + H(Y) - H(X,Y) \tag{16}$$

The *k-th* nearest neighbour estimator can be applied to individual terms. However this could mean that statistical errors incur from all individual entropy estimates. In [17] two slightly different algorithms both based on the ideas of Kozachenko and Leonenko are given which deal with this issue and are shown to produce satisfactory results. These however still depend on the dimensionality of the variables and cannot be applied directly to the broader class of metric spaces.

Houghton and Tobin first propose a method for calculating mutual information in metric spaces in [23]. Here the kernel density estimation technique (KDE) for approximating probability distributions is adapted to estimate local densities on metric space data. The model is motivated by the difficulty of estimating mutual information between discrete stimuli and spike-train responses.

It is noticed that for large enough sample sets the KDE estimator resembles a *k-th* nearest neighbour estimator such as the one proposed in [17]. By using different values of k in the subspaces of each variable the terms depending on their dimensionalities cancel. The method is tested against a histogram approach, which essentially follows the binning strategy, on fictive data, modelled to mimic the properties of spike trains and electrophysiological data. It is shown that the KDE estimator considerably outperforms the histogram approach as sample size and dimensionality/number of "spikes" increase. The metric-space method's estimation error is low and decreases as the amount of data grows, in contrast to the binning one which retains a relatively high error rate throughout.

This leads to the model of interest, proposed by Houghton in [27], which builds on the results seen in [23] and gives a method for estimating mutual information in both the cases when one variable is discrete and when both of the variables come from a metric space. The derivation is more straight-forward and intuitive - some of the complications associated with the KDE technique are avoided and the terms dependent on dimensionality in the *k-th* nearest neighbour estimator introduced in [17] are avoided. In addition a formula is derived for estimating the Kullback-Leibler divergence between two probability distributions in the same metric space.

The proposed model: probability as a measure in open balls

The lack of coordinates necessitates the introduction of an alternative strategy for measuring volumes in a metric space. The basic idea in [27] is that if $\mathcal{X}$ is a metric space it is possible to measure the distance $d(x, y)$ between any two points x and y and it is therefore possible to define a region, or *open ball*, B_ϵ around a point x_i such that

$$B_\epsilon = \{t \in \mathcal{X} : d(x, t) < \epsilon\}. \tag{17}$$

The volume V of the ball can then be estimated from the marginal probability mass contained in it - that is by counting the number of sample points it contains.

The probability that such a region contains exactly k out of N possible sample points is given by the binomial distribution:

$$P_k(x_i) = \binom{N}{k} F_i^k (1 - F_i)^{N-k} \tag{18}$$

Here F_i is the probability mass contained in region B. If we assume that the probability mass function is constant within the region, then it can be approximated by

$$F_i \approx V p_X(x_i). \tag{19}$$

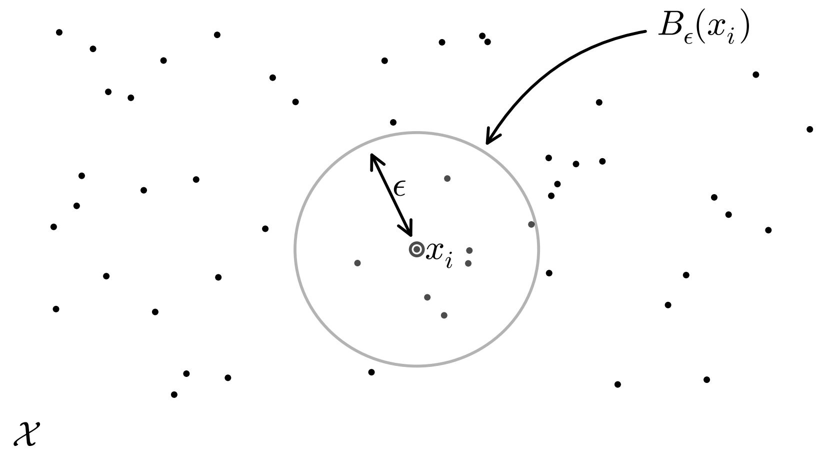


Figure 3: Open ball around of radius ϵ around instance x_i in a metric space with non-manifold geometry $\mathcal{X}$.

The justification for this is that variation in $p_X(x_i)$ should be negligible for the purposes of this approximation as long as the ball B is small enough. The same assumption underlies the models in [5, 17]. On the other hand, the expected number of points in the region is given by

$$\langle k \rangle = N F_i. \tag{20}$$

Letting #[B] denote the number of points in region B as estimated from the data, from the above it follows that

$$N V p_X(x_i) \approx \#[B(x_i, V)]. \tag{21}$$

Expressing equation 1 for a finite sample set of size N as

$$H(X) = -\frac{1}{N}\sum_{i=1}^{N}\log_2 p_X(x_i) \tag{22}$$

and using the above approximation to estimate

$$\log_2 p_X(x_i) \approx \log_2 \frac{\#[B(x_i, V)]}{NV} \tag{23}$$

the entropy of a random variable X in metric space, using region-based local density estimates, can be given as

$$H(X) \approx \log_2 N + \log_2 V - \frac{1}{N}\sum_{i=1}^{N}\log_2 \#[B(x_i, V)]. \tag{24}$$

This formula is similar to the ones proposed in [5, 17] but it is simpler. The main difference is that the probability is estimated using the expected number of points in a region rather than the size of the ball containing a given number of points in a coordinate space. This approach is taken in order to avoid quantities that depend on integrable manifolds.

Nonetheless a way to measure the volume of the region is needed. Since there is no obvious measure on a metric space without coordinates, it can be defined in terms of the probability distribution estimated from the data as the fraction of points falling in the region:

$$V \approx \frac{\#[B]}{N}. \tag{25}$$

Using such a measure, together with the above approximation for $p_X(x)$, the probability would always equal one and therefore it would always give a trivial estimate of entropy equal to zero. Probability cannot be used as a measure to estimate the entropy of a single random variable as it becomes self-referential. Furthermore the entropy of a variable depends strictly on the measure used in the domain of the variable. Mutual information however is not measure-dependent and it involves more than one variable. Therefore one probability distribution can be used as a measure for estimating others. This is the key idea underlying the suggested model.

In [27] two cases are considered reflecting two types of neuroscientific experiments. In the first, one of the variables is discrete, representing the stimulus - for example the location of a laboratory mouse on a 2D arena, and the other one takes values in a metric space - such as a spike train in the space of a similarity metric. In the second one, both random variables take values in metric spaces.

First the case when one variable is discrete is introduced. Given a discrete set of stimuli $\mathcal{S}$ of size $|\mathcal{S}| = n_s$, each presented exactly n_t times (for simplicity), a total number of $N = n_s n_t$ spike-train responses from a set $\mathcal{R}$ are elicited. Using a similarity metric on the space of spike trains, regions around each sample point are defined with ϵ chosen such that they contain exactly h neighbouring samples, that is $V = h/N$. With the total probability, $p_R(r)$ used as a measure, the entropy equals zero as seen before, but it can be used for defining an estimate of conditional entropy, based on the conditioned probability:

$$p_{R|S=s}(r) \approx \frac{\#[B]}{n_t V}. \tag{26}$$

Here $\#[B]$ counts how many of the n_t responses to stimulus s are among the h nearest neighbours of r_i in the sample set as whole. This is analogous to the approximation from equation 21. The entropy of R, conditioned on stimulus s is then estimated by

$$\begin{aligned} H(R|S = s) &\approx -\frac{1}{n_t}\sum_{i=1}^{n_t}\log_2 \frac{\#[B(r_i, V)]}{n_t V} \\ &\approx -\frac{1}{n_t}\sum_{i=1}^{n_t}\log_2 \frac{n_s\#[B(r_i, h/N)]}{h} \end{aligned} \tag{27}$$

Averaging over $s \in \mathcal{S}$ as in the equation for conditional entropy (eq. 5) this gives

$$H(R|S) \approx -\frac{1}{N}\sum_{i=1}^{N}\log_2 \frac{n_s\#[B(r_i, V)]}{h}. \tag{28}$$

Using equation 15 the mutual between R and S information is derived as

$$I(R;S) \approx \frac{1}{N}\sum_{i=1}^{N}\log_2 \frac{n_s\#[B(r_i,V)]}{h} \tag{29}$$

since $H(R) = 0$ using the same measure.

In the case when both variables S and R take values in metric spaces the probability mass functions $p_S(s)$ and $p_R(r)$ are used to measure volumes in $\mathcal{S}$ and $\mathcal{R}$ resulting in entropy estimates equal to zero. But these measures induce a measure on $\mathcal{S}\times\mathcal{R}$, which is the space where stimulus-response sample points live. A region, or *square*, in $\mathcal{S}\times\mathcal{R}$ is then defined as the cross-section of open balls in $\mathcal{S}$ and $\mathcal{R}$:

$$S\left(s_i, r_i, \frac{h_1}{N}, \frac{h_2}{N}\right) = \left\{(s,r)\in\mathcal{S}\times\mathcal{R} : s \in B_S\left(s_i, \frac{h_1}{N}\right), r \in B_R\left(r_i, \frac{h_2}{N}\right)\right\}, \tag{30}$$

that is the set of stimulus-response pairs where s is one of the h_1 nearest neighbours of s_i and r is one of the h_2 nearest neighbours of r_i.

Under this new measure the volume of the region is

$$\text{vol } S\left(s_i, r_i, \frac{h_1}{N}, \frac{h_2}{N}\right) = \text{vol } B_S\left(s_i, \frac{h_1}{N}\right)\cdot \text{vol } B_R\left(r_i, \frac{h_2}{N}\right) \approx \frac{h_1 h_2}{N^2}. \tag{31}$$

And thus the mutual information is estimated by

$$I(R;S) \approx \frac{1}{N}\sum_{i=1}^{N}\log_2 \frac{N\#[S(s_i, r_i, h_1/N, h_2/N)]}{h_1 h_2}. \tag{32}$$

The resolution of this model given by the h_1 and h_2 parameters needs to be chosen appropriately. If they are too large the approximation given in equation 20 becomes less accurate due to the assumption that probability mass function is constant in the ball. If they are made too small on the other hand the accuracy of estimates in equations 21 and 26, where the mean value is estimated by counting, is decreased.

It is important to note that when the two variables are independent the above equation gives an estimate of zero as required by the definition of mutual information. This is because there is a probability equal to h_1/N for each of the h_2 points in $B_R(r_i, h_2/N)$ to be also in $B_S(s_i, h_1/N)$ which means that there are on average h_1h_2/N points in $S(s_i, r_i, h_1/N, h_2/N)$, giving a probability estimate of one.

The same approach is applied to estimate the Kullback-Leibler diversion between the probability distributions of two random variables taking values in the same metric space. This time however the distribution of the second variable is used as a measure of the volume of the regions around instances of the first one. That is if M instances are sampled from R and N - from S, then

$$\begin{aligned} d(R||S) &\approx \frac{1}{M}\sum_{i=1}^{M}\log_2 \frac{p_R(r_i)}{p_S(r_i)} \\ &\approx \frac{1}{M}\sum_{i=1}^{M}\log_2 \frac{N\#[B(r_i, h/N]}{Mh} \end{aligned} \tag{33}$$

where B contains the h instances sampled from S that are closest to r_i and $\#[B]$ gives the number of points sampled from R falling in the ball. Measuring the Kullback-Leibler divergence between the joint and product distributions of (s,r) pairs in a metric space $\mathcal{X}^2$ in turn reduces to the formula for mutual information: setting the volume to h_1h_2/N^2 for N samples, equation 33 reduces to equation 32.

The model aims to show that the Kozachenko-Leonenko approach for information estimation can be applied in this simple form to sample spaces without coordinates. The problem that inspires it arises from the difficulty of estimating information on electrophysiological data embedded in high-dimensional Euclidean spaces, but it can be applied to a much broader range of context. If successful it could provide a framework for calculating information shared between any variables taking values in a space where a suitable similarity metric is defined. In the next part of this section the problem of assigning distance between spike trains is discussed and two state-of-the art solutions currently available are presented.

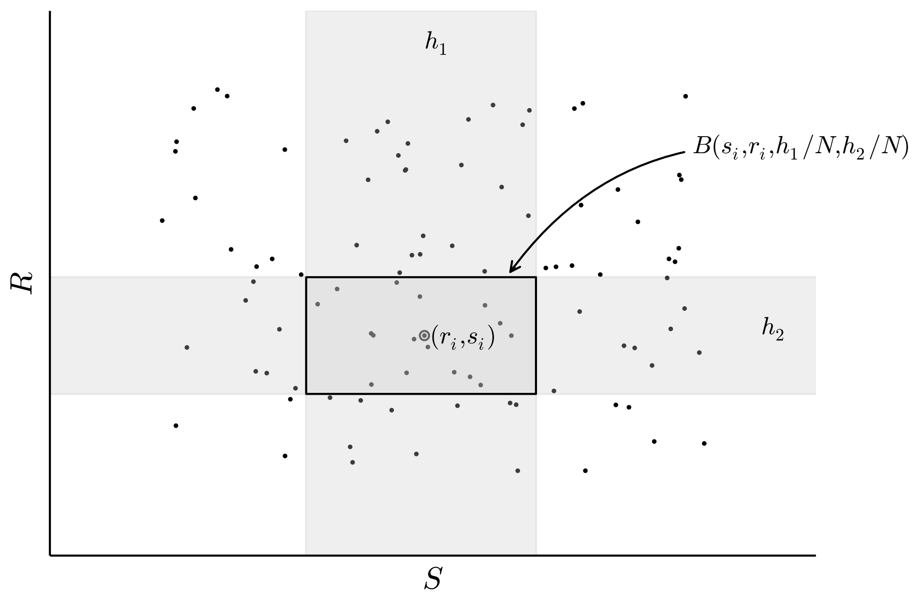


Figure 4: Probabilities are estimated in $\mathcal{S} \times \mathcal{R}$ by counting the data points in region B, which is the cross-section of the balls around s_i of volume $\frac{h_1}{N}$ and around r_i of volume $\frac{h_2}{N}$.

Spike-train metrics

Being able to compare neural activity patterns over time and across a population of neurons is of fundamental importance to understanding the semantics of neural signalling. The spike-train metrics described in [21] provide a principled approach to the problem. First, two methods for comparing pairs of spike trains from single neurons are introduced, which are then extended to the comparison of spatio-temporal patterns of population activity. The aim is to construct a mathematical framework for analysing neural coding - both in terms of firing rates and precise spike timing.

The basic idea is to start with a very general geometric description of the space of spike trains, thereby allowing for it to reflect their specific physiological properties and avoiding fitting them to an arbitrary standard which may not be suitable. Indeed, although Euclidean vector spaces have proven to be very useful in many cases, spike trains do not seem to comply with their rules. If every dimension represents a spike time for example, similar spike trains with different number of spikes would have to be parametrized in different-dimensional spaces. There is also no natural way for subtracting one spike train from another and no physiological motivation for assuming that “adding” the same spike train to two spike trains preserves the distance between them. This is why the space of spike trains is simply considered to be a metric space - one where distances between points can be measured.

Having said this, the first family of distances considered - the van Rossum kernel-based metrics [15], first map spike trains into a vector space of functions and then use Euclidean distance to calculate the distance between them. This method however avoids using a binning strategy to embed a spike train into a discrete vector space with dimensionality given by spike- count or the number of bins. Instead, it maps a spike-train to a continuous function by convolving it with a linear filter. This allows for the distance to be calculated directly using the L_2-norm metric for functions. And although if computed numerically this has the structure of Euclidean distance in high-dimensional discrete vector space, such a space does not ever need to be used explicitly. The distance can be solved analytically and taken as is, to form a metric space of spike trains.

Here the van Rossum metric with an exponential kernel is considered. This filter mimics the synaptic conductance dynamics resulting from a spike. It expresses the idea that the distance between spike trains should reflect the difference in their effect on other neurons by drawing a caricature of the effect of a spike on the synaptic conductivity.

A spike train $\mathbf{u} = \{u_1, u_2, \ldots, u_m\}$ is mapped to a real function using a kernel $h(t)$:

$$\mathbf{u} \mapsto f(t; \mathbf{u}) = \sum_{i=1}^{m} h(t - u_i), \tag{34}$$

the kernel itself is defined as

$$h(t) = \begin{cases} 0, & t < 0 \\ \frac{1}{\tau} e^{-t/\tau}, & t \geq 0 \end{cases} \tag{35}$$

where τ is a time scale associated with the synapse. The distance between spike trains $\mathbf{u}$ and $\mathbf{v}$ is given by the L_2 distance between the corresponding functions:

$$d(\mathbf{u}, \mathbf{v}; \tau) = \sqrt{\int dt [f(t; \mathbf{u}) - f(t; \mathbf{v})]^2} \tag{36}$$

For the causal exponential filter this integral can be solved analytically [22] to express the distance as a summation in terms of exponentials:

$$d(\mathbf{u}, \mathbf{v}; \tau) = \sqrt{\sum_{i,j} e^{-|u_i - u_j|/\tau} + \sum_{i,j} e^{-|v_i - v_j|/\tau} - 2 \sum_{i,j} e^{-|u_i - v_j|/\tau}}. \tag{37}$$

When an action potential arrives at a synapse it causes an abrupt rise in the conductivity of the post-synaptic membrane. Eventually this results in a spike in the membrane potential of the post- synaptic neuron. This event occurs on a time scale much smaller than the rest of neural electrodynamics, which is why it is modelled as instantaneous:

$$f \rightarrow f + \delta f. \tag{38}$$

The increment δf depends on the scale chosen - it can be set to 1. After the spike has arrived the conductivity is assumed to drop at a constant rate

$$\tau \frac{d}{dt} f = -f \tag{39}$$

hence the exponential decay. This construct can incorporate another aspect of synaptic conductance - its dynamics when multiple spikes arrive within a short time interval. Since there is a great variety of synapses, the limit to their conductance is variable too. In many synapses the arrival of a spike has a diminishing effect on subsequent impulses arriving shortly after it. This can be modelled by adding an extra parameter but will not be discussed here as it is not necessary for the purposes of this project.

The other family of spike-train distances is the one of edit-length metrics, exemplified by the Victor-Purpura metric [7, 8]. As mentioned above this is similar in spirit to the Hamming distance used in string matching. In particular, it defines the distance between two sequences of spikes as the cost of moulding one into the other. As a result, the signals are not modelled by any intermediate representation and the distance can be computed directly given a pair of spike trains following a set of rules.

The main concept associated with an edit-distance is a *cost function* defined in terms of a set of *elementary steps*, each assigned a non-negative individual cost. In the context of strings of symbols over a finite alphabet this usually includes the three basic operations - insertion, deletion and substitution. The total cost $c(\gamma)$ of transforming one sequence into another is thus given by the set of elementary steps that can be used to complete the task at a minimal overall cost:

$$d(\mathbf{u}, \mathbf{v}) = \min_{\gamma} c(\gamma) \tag{40}$$

As long as the cost of a step equals the cost of inverting it this distance is certainly symmetric. For the purposes of comparing spike trains, the basic elementary step consists of inserting or deleting a spike and has a unit cost, which sets a scale on the metric. This guarantees there is a finite distance between any pair of spike trains given by replacing all spikes from one train with the spikes of another one, and equal to the sum of their lengths. In the

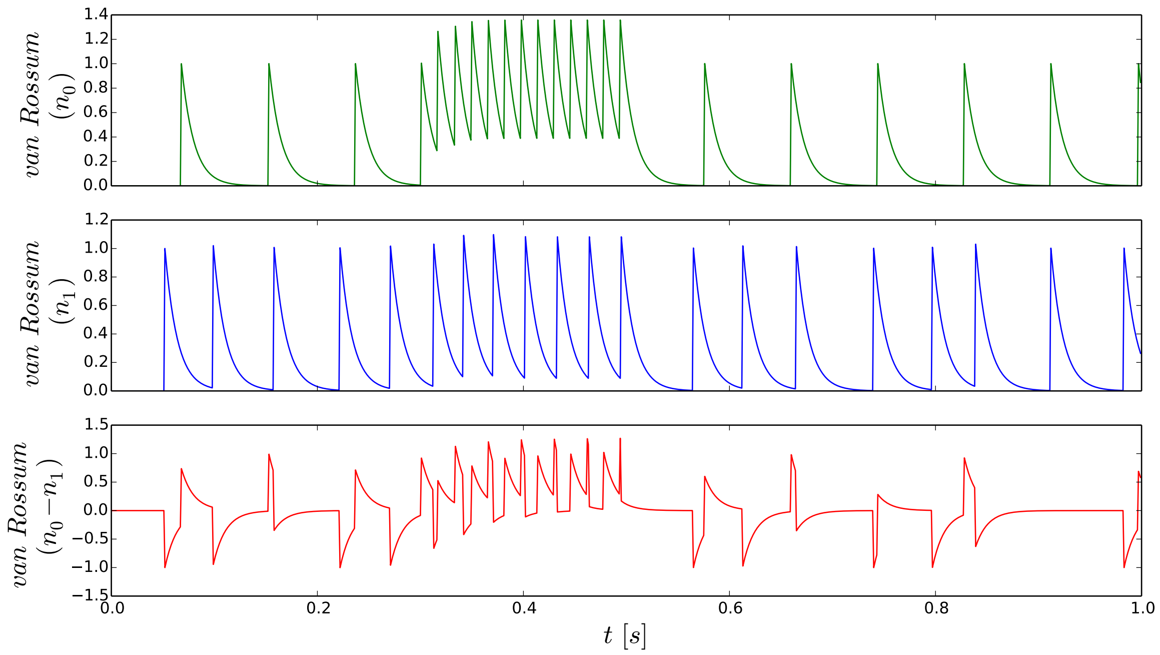


Figure 5: **(top, middle)**Mapping spike trains to functions using the van Rossum metric with an exponential kernel. **(bottom)** Subtracting one function from the other.

simple case of the Victor-Purpura “spike-time” metric, a second elementary step is added consisting of moving a spike by some amount of δt. This step has a cost of $q\delta t$ associated with it, where the parameter q determines the cost per distance for moving a spike. This means that the cost of moving a spike cannot exceed the cost of deleting it and inserting a new one at the desired position in time: $q\delta t < 2$, so the maximum distance by which any spike can be moved to achieve minimum total cost is given by a time-scale of $2/q$ determining the sensitivity of the metric. The problem of finding an optimal sequence of steps can addressed efficiently using a dynamic programming procedure:

```
Input: spike trains $\mathbf{u}$ of length $m$, $\mathbf{v}$ of length $n$
Output: $d(\mathbf{u}, \mathbf{v}) = \min_\gamma c(\gamma)$
$G \leftarrow$ array of size $m \times n$
$G_{i,0} \leftarrow i$ for $i \in [1, m]$, $G_{0,j} \leftarrow j$ for $j \in [1, n]$
for $i \leftarrow 2$ to $m$ do
    for $j \leftarrow 2$ to $n$ do
        $G_{i,j} \leftarrow min(G_{i-1,j-1} + q|\mathbf{u}_i - \mathbf{v}_j|, G_{i-1,j} + 1, Gi, j - 1 + 1)$
    end
end
return $G_{m,n}$
```

Algorithm 1: Matching one spike-train to another at minimum cost

The above algorithm takes advantage of the properties of minimum cost-sequences which exclude inefficient steps such as moving a spike from one train past a spike from the other, or making redundant moves in combination with insertions or deletions. Every entry $G_{i,j}$ denotes the distance between the sub-trains containing the first i spikes from $\mathbf{u}$ and the first j spikes from $\mathbf{v}$. At each step, the algorithm considers the last spikes from the respective sub-trains and makes a minimum-cost choice among three possibilities: 1) matching the two spike trains, 2) deleting the last spike from one train, or 3) inserting a new spike into it matching the last one of the other. In this way the dynamic programming paradigm is applied - avoiding recursion by storing the results from previous computations and building the final result from the bottom up.

Both the kernel-based van Rossum metric and the edit-length Victor-Purpura metrics can be extended to compare population activity by extending the domain in which spikes occur with another variable apart from time - the "neuron of origin". This involves adapting each of them in specific ways reflecting their mathematical properties, but abstracting away from these details an important parallel can be drawn between multi neuronal metrics and another similarity measure used in image analysis - the*Earth Mover Distance* [12]. The concept is that an image, represented by a histogram is thought of as a 3D terrain, and the similarity between it and another image is measured in terms of the labour - volume×distance, required to match one terrain to the other. In order to do so a cost function is defined consisting of elementary steps for moving adding and subtracting mass. This approach can in turn be extended to incorporate higher-dimensional objects such as video, volumetric images or functional data. The cost for moving events does not need to be dimension-invariant in any of these cases nor does it need to reflect Euclidean distances. This example aims to show that alternative metrics can be used in many contexts and methods for estimating probability and information quantities in the spaces they give rise to can be useful to various applications.

The two types of spike-train similarity measures described above have very different mathematical structures but they are both quite successful in capturing the differences between neural activities in terms of their effects on other neurons. One of the ways this is tested and verified is by examining their performance when used to cluster together spike trains elicited in response to the same stimulus. Interestingly, regardless of their distinct properties the kernel-based and edit-length metrics perform very similarly on this task. This can be explained by noting that under certain settings the metrics themselves treat spike trains in an analogous fashion. These will not be discussed here - instead the variants introduced will both be used to test the information estimator in order to verify its validity and test its performance in metric spaces with varied underlying structure.

The next and final section of this chapter explores the general physiological properties of nerve cells and the mechanisms they use to connect and interact in the nervous system, along with the mathematical models used to describe them relevant to this project. It does not aim to encompass the full picture of their neurological understanding, but rather to motivate the computational model used to implement a simulation of a network of spiking neurons in software. This network will serve as the basis of an environment for configuring experiments that generate spike-train data which will be used to test the mutual information estimator.

1.3 Scientific and technical aspects

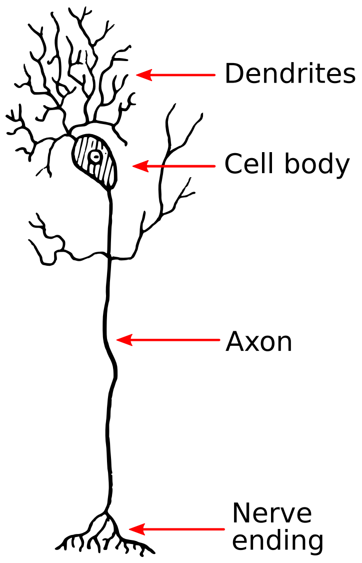


Figure 6: Simple diagram of a neuron from Wikimedia Commons.

The nerve cell

The nerve cell, or *neuron*, is arguably one of the most interesting and important cells found in the bodies of nearly all animals. It is absent only in a few very simple multicellular organisms such as sponges. Together with the *glial* cells (Greek for glue), which support them structurally and metabolically, they constitute the nervous systems that govern the activity of organisms. This is accomplished through the communication and processing of information in the form of electrical signals transmitted between neurons connected in networks.

Morphologically the neuron can be seen as being comprised of three main components. Through the *dendrites*, signals are received by the cell body, or *soma*, which processes them and in turn transmits a signal along the *axon* reaching out of it towards other cells. The characterising property of neurons is that they are electrically excitable by other neurons. Most commonly, they connect to each other via chemical synapses mediating electrical charges between their axons and dendrites. In this context neurons are distinguished as pre-synaptic and post-synaptic. There are two types of synapses: excitatory - making the post-synaptic neuron more likely to signal, and inhibitory - suppressing its signalling.

An electric impulse is elicited by the nerve cell when the potential difference across its membrane exceeds a certain threshold - typically around -55 mV. When the cell is at rest, the electric potential inside is lower than the potential outside of it. The potential difference is referred to as *membrane potential*, amounting to about -70 mV when at ease. This state is actively maintained by *ion pumps* - mechanised protein channels moving electrically charged particles across the membrane of the cell using energy. An ubiquitous example is the sodium-potassium adenosine triphosphatase enzyme. It pumps three positively charged sodium (Na+) ions out of the cell for every two positive potassium (K+) ions it pumps inwards, resulting in a loss of one atomic charge per pumping cycle. In this way potassium is concentrated inside the soma whereas sodium is pushed out. The negative voltage equilibrium is a result of this saturation. The random thermal motion of potassium ions causes them to diffuse outwards despite of the lower potential inside the cell body, which on the other hand attracts sodium. It takes the reciprocal activity of the ion pumps to counteract this force and compensate for the leak of charge.

A neuron can be either excitatory or inhibitory in the sense that as a pre-synaptic neuron it can form either only excitatory or only inhibitory synapses to other, post-synaptic neurons. Chemical synapses work by virtue of the release of substances called *neurotransmitters* from the terminal of the axon of the pre-synaptic neuron due to the arrival of an electrical impulse. The released neurotransmitter fits like a key in a lock into binding sites called *receptors* situated in the dendrites of the post-synaptic neuron. The receptors open up specialised *ligand-gated* ion channels and allow ions through the membrane of the post-synaptic neuron for a short time period. Depending on the type of the synapse and of the ion channels this leads to a small increase or decrease in its membrane potential.

When the voltage reaches its threshold value this activates the opening of yet another type of ion channels allowing an abrupt influx of sodium inside the cell during a narrow time window - 1–2 $[ms]$. This produces a non-linear cascade in the membrane potential called an *action potential* or spike. This impulse then propagates through the axon, being continuously amplified in the same fashion by the opening and closing of voltage-gated channels along its length.

The story given here abstracts away many details important to an accurate biological and physical description of the nerve cell and how it works. Nevertheless it is sufficient to give an outline of the behaviour of neural voltage dynamics for the purposes of the mathematical model used to describe it introduced below.

The leaky integrate and fire model

In this project a variant of what is known as the leaky integrate and fire model [1] is used to simulate the behaviour of spiking neurons wired together in a network. This simplified model represents the voltage dynamics of a neuron as a first-order ordinary differential equation. An intuitive analogy can be made between this equation and a similar one giving a simple model for the rate of change in the height of water in a leaky bucket that is being continuously filled from the top.

The basic idea is that the rate l at which water leaks out is proportional to the height h of the water inside the bucket as it determines the pressure at its bottom. Assume for simplicity that l also depends on a single constant G, reflecting the size of a hole on the bottom and some other physical factors. The relationship between the two is expressed as

$$l = Gh. \tag{41}$$

Let i denote the rate at which water pours into the bucket and C - the area of the base (the sides of the bucket are considered to be straight). The rate of change of the volume of water contained in the bucket is then

$$\frac{dCh}{dt} = i - Gh \tag{42}$$

and consequently the rate of change of its height is

$$\frac{dh}{dt} = \frac{1}{C}(i - Gh). \tag{43}$$

The height is used as an analogue for the voltage - greater height/voltage means greater difference in pressure/potential between the the two ends of the water-column/cell-membrane. In the model for the membrane potential the cross-sectional area of the bucket C is replaced by the electric capacitance of the membrane C_m and G is substituted with G_m - the conductance of the membrane. As mentioned above the equilibrium voltage at which there is zero current leaking through the membrane equals -70 mV and is actively maintained by the sodium- potassium pumps - this is called the *reversal potential* of the membrane, denoted E_L. The leak current out of the cell is then given by Ohm's law as $G_m(V - E_L)$. Equation 42 is thus rewritten as

$$\frac{dC_mV}{dt} = I - G_m(V - E_L) \tag{44}$$

where I is the input current. In the original experiment by Lapicque [1] this is injected directly into the cell with an electrode - usually denoted I_e, but it could also stand for currents coming in through the synapses. Usually the equation is divided across by the conductance. The resistance is then denoted $R_m = 1/G_m$ and $\tau_m = C_m/G_m$ gives a time scale for the membrane:

$$\tau_m \frac{dV}{dt} = E_L - V + R_m I. \tag{45}$$

This equation does not model the non-linear effect of the voltage reaching the threshold of about -55 mV which produces the spike. This is accounted for in a more detailed equation given by the Hodgkin Huxley model [3] which includes terms for the currents due to the opening and closing of the voltage-gated channels. However, since the time-scale at which these non-linear dynamics occur is very narrow compared to the rest of the process, they can be modelled as occurring instantaneously. The equation will not be solved analytically but rather integrated numerically at discrete time steps in software using an approximation technique. Therefore a spike can be added by hand by setting the voltage to a value above zero once the threshold is reached and then immediately resetting it to to a value near the rest point of -70 mV. In fact, for the purposes of estimating information in spike trains, spikes will not even need be added but just recorded before V is reset.

The strategy described above for simulating neuronal activity numerically on a computer makes it easier to solve the model for a variable input current due to the synapses. The synaptic current is the last ingredient left to define in order to complete the model so that it can be used to simulate the dynamics of a population of integrate and fire neurons interacting together in a network. It will be modelled by the following equation

$$I_s(t) = g_s s(t)(E_s - V) \tag{46}$$

where E_s is the reversal potential of the synapse and $g_s s(t)$, later also denoted G_s, is the conductance at the synapse - g_s describes the synaptic strength, and s(t) models the opening and closing of the ligand-gated channels due to the arrival of a spike:

$$s(t) = G_{max} e^{-t/\tau_s} \tag{47}$$

where t_s is the time since the pre-synaptic neuron's last spike, τ_s is the time-scale of the synapse, and G_{max} is a constant used to control the scale of the conductance. The exponential decay models the unbinding of neurotransmitter which closes the ligand-gated channels after they open. In a model that strives for accuracy G_{max} can be replaced by t, but here it is assumed that the gates open instantaneously and the control parameter serves the purpose of tuning the simulation.

The full equation for the voltage can now be given as

$$\tau_m \frac{dV}{dt} = E_L - V + R_m I_e + R_m \sum_{i=1}^{N} I_s(t, i) \tag{48}$$

for a neuron with N pre-synaptic neurons connected to it. The constant input current I_e is optional and can be used to stimulate particular neurons or just to stabilise the network.

Numerical methods

The technique used to compute a numerical solution to the membrane potential equation is known as the Runge-Kutta method [6]. It is based on the Taylor series expansion of a function around a point t_0:

$$f(t) = \sum_{i=0}^{\infty} \frac{1}{n!} \frac{d^n f}{dt^n}\bigg|_{t=t_0} (t - t_0)^n. \tag{49}$$

This gives an iterative method for efficiently solving a differential equation of the form

$$\frac{df(t)}{dt} = F(f, t) \tag{50}$$

to some approximation by expanding f around discrete time-steps split by some small interval δt, up to a term of some order N. In other words $f(t + \delta t)$ can be worked out from $f(t)$ by using its derivative F:

$$f(t + \delta t) = f(t) + F(f, t)\delta t + \frac{1}{2} F'(f, t)\delta t^2 + O(\delta t^3). \tag{51}$$

The simplest way to go about this is to ignore the δt^2 and smaller terms - this is known as the *Euler method.* Sometimes one might expect that the errors would have different signs and cancel often enough to give a good approximation. However this is not always the case - they might add and the $O(\delta t^2)$ error could cause a considerable offset. The Runge-Kutta methods address this issue by considering a number of different Taylor expansions so that the termsδt^2 and smaller cancel. The most common version is the classical Runge- Kutta fourth order method known as "Num.recepies.in.C":

Given an initial condition $f(t_0)$ let

$$\begin{aligned} t_n &= t_0 + n\delta t \\ f_n &= f(t_n) \end{aligned} \tag{52}$$

then define the following four coefficients

$$\begin{aligned} k_1 &= \delta t F(f_n, t_n) \\ k_2 &= \delta t F(f_n + k_1/2, t_n + \delta t/2) \\ k_3 &= \delta t F(f_n + k_2/2, t_n + \delta t/2) \\ k_4 &= \delta t F(f_n + k_3, t_n + \delta t) \end{aligned} \tag{53}$$

finally, compute the value of f at time t_{n+1} as

$$f_{n+1} = f_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{54}$$

The k_i coefficients are increments based on estimates of the slope of the function at different points through interval of length δt. The weighted average of the four of them is taken ensuring the error is $O(\delta t^5)$ without explicitly computing higher-order derivatives of f. This approximation gives a considerable performance improvement over the Euler method and provides good enough precision for exploring neural voltage dynamics computationally.

Poisson neurons

In a real situation, timing between successful action potentials elicited by neurons from many parts of the brain appears to be very irregular under a wide variety of circumstances. It is even often observed that spike trains produced in response to the same stimulus over consecutive trials although similar would vary considerably both in terms of their firing rates and spike times [13]. This is due to noise propagating through networks of deep connectivity, but consequently it is sometimes reasonable use a stochastic model for generating fictive spiking behaviour. In the previous section a recipe for simulating a spiking neuron has been introduced but such a neuron needs variable input in order to exhibit interesting firing behaviour. Instead of constructing complex networks or encoding this variability explicitly in the external stimulation current, it is useful to be able to generate the spiking of some of the neurons in a network as random sequences with some average firing rate. In this way they can be used to provide input to others simulated using the physiologically inspired model.

In theoretical neuroscientific the problem of estimating the probability that a particular spike train occurs arises from the need to model the relationship between a stimulus and a response. The total number of possible spike sequences over a meaningful time interval is usually excessively large so estimating the probability of each one is infeasible. Therefore a statistical model based on a limited set of observed responses is needed to estimate their probability distribution. This is typically done in an attempt to predict the stimulus that produced a given response or to quantify the probability of the response itself occurring.

Since spikes are considered to be stereotyped events proceeding at a very tight time-scale, they are idealised as occurring instantaneously - an assumption that has been at work throughout this exposition by the sheer use of spike trains. For a time bin, small enough to ensure there can be no more than one spike occurring in it, the probability of a spike can be determined by the firing rate of the neuron. This statistic is generally not sufficient to predict the probability of a spike train because the joint probability of two spikes occurring in particular time-slots is not necessary equal to the product of their individual probabilities of doing so. In fact, in many situations there is some dependence - for example if one spike follows closely after another it is quite likely that presence of one influences the occurrence of the other. Having said this, if action potentials are assumed to be statistically independent then the firing rate can indeed be used to estimate the probability of any given sequence of spikes.

A stochastic process generating a sequence of events such that there is no dependence between an event and the history of preceding events is called a *Poisson process*. If the probability of an individual event occurring is invariable the process is said to be a *homogeneous* Poisson process. This kind of process provides an extremely useful model for irregular neuron firing. Below a formal definition is given based on the description in [13], followed by a simple procedure for generating Poisson spike trains on a computer that is used in the implementation part of the project.

Let $r(i) = \rho$ be a constant equal to the average spike-rate of the nerve cell. T will denote the length of the time period which is broken up into M bins of length $\delta t = T/M$. The probability of a spike occurring in a specific bin is then $\rho\delta t$. Because the spikes are believed to occur independently at equal probability, the probability $P[t_1, t_2, ..., t_n]$ that an ordered sequence of n spikes occurs over T with spike i falling between $t_i + \delta t$ can be expressed in terms of the probability $P_T[n]$ that any sequence of n spikes occurs in that period:

$$P[t_1, t_2, ..., t_n] = n! P_T[n] \left(\frac{\delta t}{T}\right)^n \tag{55}$$

$n!$ gives the number of ways to order the spikes and the $(\delta t/T)^n$ factor arises from the probability density of the n spike times being multiplied by the width of the time bins. $P_T[n]$ is given by

$$P_T[n] = \lim_{\delta t \to 0} \binom{M}{n} (\rho\delta t)^n (1 - \rho\delta t)^{M-n}. \tag{56}$$

The binomial expansion is the same as the one used in equation 18 - the combination $\binom{M}{n} = M!/n!(M-n)!$ gives the number of ways to pick the bins with spikes in them, $(\rho\delta t)^n$ is the probability of n spikes occurring in n specific bins and $(1-\rho\delta t)^{M-n}$ is the probability of not having spikes in the remaining $M - n$ bins. Here this is taken at the limit of $\delta t \to 0$ reflecting assumption that the bins are small enough to avoid collisions between spikes. As a result M grows without a bound and $M - n \approx M = T/\delta t$ for a fixed n. Using this approximation

$$\lim_{\delta t \to 0} (1 - \rho\delta t)^{M-n} = \lim_{\delta t \to 0} ((1+\epsilon)^{1/\epsilon})^{-\rho T} = e^{-\rho T} \tag{57}$$

where $\epsilon = -\rho\delta t$ and $e = \exp(1) = \lim_{\delta t \to 0}(1+\epsilon)^{1/\epsilon}$ by the definition of Euler's constant in series form. Then for a large enough M another approximation can be made:

$$M!/(M-n)! \approx M^n = (T/\delta t)^n \tag{58}$$

to give the formula for the Poisson distribution:

$$P_T[n] = \frac{(\rho T)^n}{n!} e^{-\rho T}. \tag{59}$$

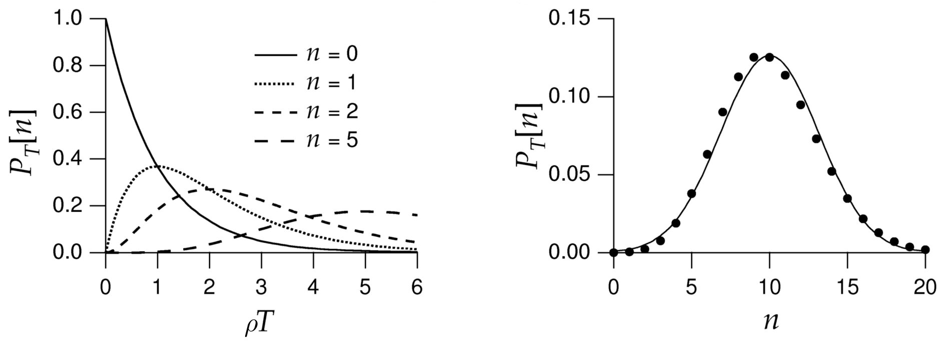


Figure 7: **(left)** The probability that a homogeneous Poisson process generates n spikes in a period T plotted for $n = 0, 1, 2$, and 5 generalised as function of ρT to apply for any rate; **(right)** The probability of n spikes occurring for $\rho T = 10$ (dots) compared with a Gaussian distribution with mean and variance equal to 10 (line). The figure is from [13].

figure 5 on the left shows this plotted for different values of n as a function of ρT in order to apply for any value of the constant rate. The larger the number of spikes n is the longer T needs to grow in order for it to reach its maximum, i.e. The longer T is the more likely it becomes to have more spikes occurring. On the right the probability mass function over possible numbers of spikes in is plotted showing that it matches a normal distribution around the expected fire rate.

Now a simple procedure for generating spike trains from a Poisson distribution can be devised using that the estimated probability of firing a spike within an interval of length δt is $\rho\delta t$. As the program progresses through time steps a random number between zero and one is generated every time. If the probability of a spike is higher than that number the time is recorded and added to the spike train. The pseudocode for this is given below. This approach can be used even if the fire rate $r(t)$ is not estimated by a constant but depends on time, as long as it varies slowly with respect to the time bin δt.

```
**Input:** time period $T$, step length $\delta t$, expected firing rate $\rho$
**Output:** spike train $\mathbf{u}$, spike count $c$
$\mathbf{u} \leftarrow [\,]$
**for** $i \leftarrow 1$ **to** $T/\delta t$ **do**
    $x_{rand} \leftarrow rand(0, 1)$
    **if** $\rho\delta t > x_{rand}$ **then**
        $\mathbf{u} \leftarrow \mathbf{u} \cup [i\delta t]$
        $c \leftarrow c + 1$
    **end**
**end**
**return** $\mathbf{u}, c$
```

Algorithm 2: Generating a Poisson neuron

This concludes the introductory part. We now have all the necessary components to build a model experiment for testing the formula. The next Chapter describes the experimental conditions simulated in software to generate spike-train data for the model estimator. This is followed by a documentation of the Python implementation of a simulation environment consisting of a spiking neural network, two spike-train metric measurements, the mutual information estimator and the experimental procedures. This produces fictive data using a deterministic neural model that captures only the essential principle of the real biophysical process. However, the level of abstraction is sufficient to guarantee statistical relationships between spike train data analogous to the ones expected to be seen in a real situation. The results form the conducted trials are then discussed and analysed in Chapter four.

2 Testing the model

2.1 Idea

Information theory establishes a relationship between the informativeness and the probability distribution of the outcomes of random variables. It extends our understanding about the meaning of structure in signals by enabling us to measure how useful a piece of information is based on how likely we are to observe it. The concept of a *random variable* itself, which in spite of its misleading name serves as a baseline mathematical model for real processes, is better understood by extending the methods of probability theory and statistics with the ones of information theory. The main motivation for applying information theoretic analysis to neuroscientific data is quantifying the relationship between a stimulus and a response.

The estimation problem left aside, we are interested in measuring the information carried in a spike-train response about a given stimulus. The stimulus can be the spiking of a pre-synaptic neuron or it can be represented by a discrete variable modelling some experimental condition. Effectively, we would like to separate the structure, or "randomness", contained in spike-train responses that is directly influenced by a specific stimulus from other content, which may be due to other stimuli or noise propagating through the network. This is exactly what mutual information (MI) measures. As seen in equation 15, mutual information can be thought of as the uncertainty associated with one variable taking away what would be left of it if the other variable were known. In other words, it measures the information content present in a variable due to its relationship with another variable. This information measurement has no dependency on the actual values the variable takes or any statistic associated with them. It is rather a statistic of the minimum coding length needed to capture their improbability and as such is a direct result of their probability distribution.

In order to test an information estimator using just a computer we want to design a computational model generating multiple random variables related in a predefined way. We would then make a hypothesis about the expression of this relationship in terms of mutual information and make the respective measurements using the formula in question. Combining the framework of computational neuroscience with the spike-train metrics approach we can investigate the performance of the metric-space model for mutual information in the context of neural spike trains and check if the results match our hypothesis. The kind of data which is collected from electrophysological experiments with neurons can be mimicked by simulating some neurons using the integrate-and-fire model while feeding them randomly generated pre-synaptic spiking with firing rates coming from a Poisson distribution. As outlined above the spike trains generated using the Poisson model imitate neural activity resulting from the rich connectivity observed in many real neural networks.

To produce the desired statistical relationships we create an artificial neural network incorporating both types of neuron models. By simulating the network's voltage dynamics over a series of trials under the same experimental conditions, we generate sample sets of the spike-train variables modelled by its comprising neurons. We then apply the metric-space method for mutual information estimation, which relies on nearest-neighbour statistics between spike-trains in each of these sets. Finally, we repeat this process while varying certain experimental parameters so we can observe the resulting tendencies in the variation of the mutual information estimates and compare them to our expectations.

2.2 Experimental network set-up

Abstracting away implementation details we first define the experimental conditions to be simulated. We start by choosing a simple network topology which enables us to clearly establish the relationships between the spike trains that will be produced by individual neurons. This way we can predict the effect of these relationships on the mutual information to be measured.

The voltage dynamics of integrate-and-fire (IF) neurons, as modelled by equation 48, depend on two input components. On one hand, we have the constant current I_e, modelling stimulation through an electrode injected directly into the soma. This current causes simple rhythmic spiking provided it is high enough. On the other hand, there is the combination of synaptic currents, which at any given time depend strictly on the spiking times of pre-synaptic neurons - our stimuli. The sets of spiking patterns elicited by the stimuli need to contain some kind of "randomness" to ensure that the observed information propagation between them and the responses is representative of the general case. This is where the Poisson model comes in, enabling us to generate diverse pre-synaptic spiking. Furthermore, each of the simulated neurons should process spiking from multiple sources so that there is some extra structure in its response. The mutual information estimator must be able to distinguish between the two influences.

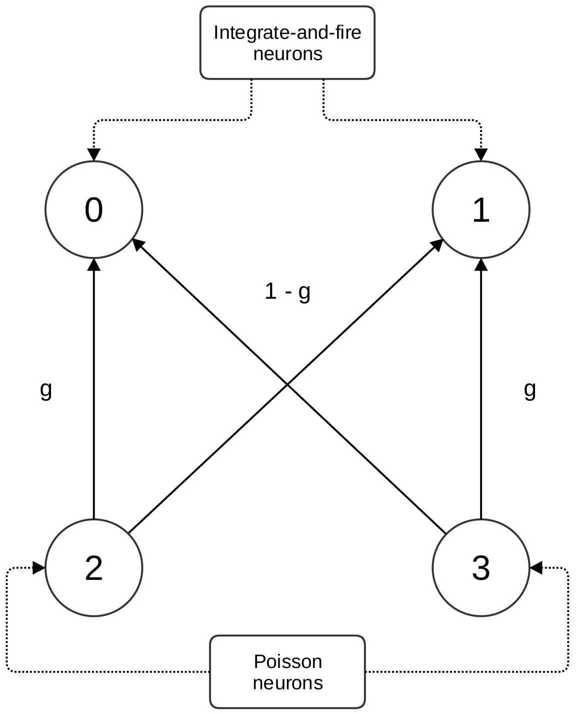


Figure 8: The experimental network layout

The simple network connectivity used in the experiments, displayed in figure 8, is chosen to fit the above requirements. We have two Poisson neurons, labelled $\textcircled{2}$ and $\textcircled{3}$, which spike randomly with predefined average firing rates. These mean rates will be chosen randomly from a range of possible values but we will get back to this again. The important point is that the produced spike trains will be different and completely independent from each other every time. We also have neurons $\textcircled{0}$ and $\textcircled{1}$, whose membrane potential will be simulated numerically with the leaky integrate and fire model. The two Poisson neurons synapt to each of the two simulated neurons. This means that as the simulation proceeds their spiking times affect the synaptic conductance of IF neurons, and hence their voltage dynamics. We will be simulating excitatory synapses throughout. The synaptic conductivity - that is the porousness of the post-synaptic membrane - will depend on the strength of the synaptic connection, labelled with g in the figure above for synapses $\textcircled{2} \rightarrow \textcircled{0}$ and $\textcircled{3} \rightarrow \textcircled{1}$. This parameter takes values in the interval $(0, 1]$ and the connection strength for synapses $\textcircled{2} \rightarrow \textcircled{1}$ and $\textcircled{3} \rightarrow \textcircled{0}$ is set to g's complement to one: $g_{2,1} = g_{3,0} = 1 - g$. Thus the effects of the two stimuli are in a kind of contra-proportional relation whereby increasing one of them we decrease the other. An analogous relation must be exhibited between the respective mutual information measurements for the model estimator to pass our validity test.

2.3 Experiment 1: Mutual information vs. synapse strength

This is the core test we are interested in conducting. We step up g and, as we progress, at each step we simulate the experimental network multiple times in order to generate four spike-train sample sets. Treating the spike trains recorded from each neuron as a separate metric-space variable we apply the mutual information estimator pairwise using one of the two spike-train metrics presented in Chapter 1.2. That is, for each connectivity strength, we calculate the MI between Poisson-generated stimuli and IF-simulated responses, as well as across - between pairs of neurons of the same type. Now, the stimuli are generated using a pseudo-random number facility on a computer, following the procedure given in algorithm 2. The average rate is also chosen uniformly from a small range for every new spike train generated independently by each of the two Poisson neurons at each simulation trial for every value of g, varying between zero and one. Therefore we can expect the proximity between the generated stimuli in the space of spike trains to be somewhat uniformly distributed across some region. The key point is that the responses of IF neurons are related to the stimuli. Consequently, the proximity between them would also contain some randomness, but more importantly similar stimuli will be close together and the same will be true about the similar responses to them. This is the statistical property that the estimator at hand exploits.

There are several parameters that need setting. Let $T_{\#}$ denote the number of simulation trials per experimental phase (synapse strength step). This number needs to be large enough for the estimates to be meaningful. The sizes of open balls determined by the h_1 and h_2 parameters in the model estimator need to be chosen in accordance with $T_{\#}$. The volumes must be small enough for the constant probability mass assumption to be valid for the balls, but large enough to provide representative statistics for estimating densities. The developed software implements the experiment in a generic way so that it can be used to simulate it with any valid settings configuration. We also need to set a resolution in the independent variable - the synaptic strength $g = g_{2,0} = g_{3,1}$. Let G denote the number of sample points of g over $(0, 1]$, that the pairwise mutual information will be calculated for. A higher sampling rate on g will make the analysis of the relationship between MI and connection strength more accurate and less susceptible to noise and outliers. In addition, we have the simulation parameters associated with the modelled neurons. Here we use some standard settings for the IF neurons and introduce some randomness to their initial states and to the fire rates of Poisson neurons as discussed earlier.

We now devise the exact procedure followed for conducting this experiment. A control flow diagram is given in figure 9 below. This is a hight-level description of the underlying simulation process, focusing on the recipe for conducting the experiment and not on its ingredients. Most of the experimental parameters, listed in the initialisation block, are represented by their symbolic identifiers - they can be experimented with further. Below them are listed the standard numeric settings for IF neurons used throughout the tests. The chosen network topology is used as an example. We iterate through the range of values of g, at each step updating the connectivity matrix, producing new sets of $T_{\#}$ spike-train samples from each neuron, and calculating the pairwise mutual information between them. As we go along we collect some statistics on the MI estimates. We are interested in investigating the relationship of the mutual information of the signals of two neurons to the strength of the synaptic connection between them. Our hypothesis is that MI estimates will increase linearly as the synapse strengthens. Therefore we would like to measure the linear correlation between the two and hence we compute the *Pearson product-moment coefficient*:

$$\begin{aligned}\rho_{MI,g} = \frac{cov(MI,g)}{\sigma_{MI}\sigma_g} &= \frac{E[(MI - E[MI])(g - E[g])]}{\sqrt{E[(MI - E[MI])^2]}\sqrt{E[(g - E[g])^2]}} \\ &= \frac{E[MIg] - E[MI]E[g]}{\sqrt{E[MI^2] - E[MI]^2}\sqrt{E[g^2] - E[g]^2}}\end{aligned} \tag{60}$$

To calculate the expected values involved in the last expression we sum up the respective quantities as we proceed through the experimental phases, and normalise them in the end. This coefficient measures how closely the relationship of the two variables resembles a straight line - its magnitude reflects the slope too and its sign determines the direction of the relationship. If the correlation coefficient is positive - $\rho \in (0, +1]$ - reflecting positive covariance, then there is a positive linear relationship between the two variables - they grow together. If there is negative correlation between the variables - $\rho \in [-1, 0)$ - then one of them decreases as the other grows. If $\rho = \pm 1$ there is total correlation between the variables meaning the sampled points must lay on the same line. Zero correlation means there is no linear relationship - the variables could potentially be independent or there could be some symmetric structure to their relationship.

We are also interested in modelling and graphically depicting the general tendency of this hypothetical linear relationship. To estimate MI as a linear function of g we find the line which minimises the deviations of the calculated data points. This is also known as the *least-squares* method for linear regression. We have collected mutual information measurements at G synapse-strength sample points and would like to find a line $\hat{MI} = \hat{a} + \hat{b} \cdot g$ minimising the sum of squared residuals $\sum_{i=1}^{G}(MI_i - (a + bg_i))^2$. In order to estimate the slope $\hat{b}$ and the intersect $\hat{a}$ we take the partial derivatives of this expression with respect to b and a and set them to zero. As a consequence $\sum_{i=1}^{G}(MI_i - (\hat{a} + \hat{b}g_i)) = G(\bar{MI} - \hat{a} - \hat{b}\bar{g}) = 0$ which is an appealing result. This method actually does not handle errors in the data very well, but it is good enough to help us draw a basic picture of the correlation. Although we try to have some noise in the simulation, we are still not dealing with *real* signals here. Assuming everything works as intended and given a high-enough resolution on the model, we are not expecting significant outliers in the MI estimates. The slope of the line is not too important to us here, nor is the intersect, since they are scale-dependent. The regression line can be thought of as a way to predict the value of the mutual information estimate, for a given synapse strength g with minimum error on average.

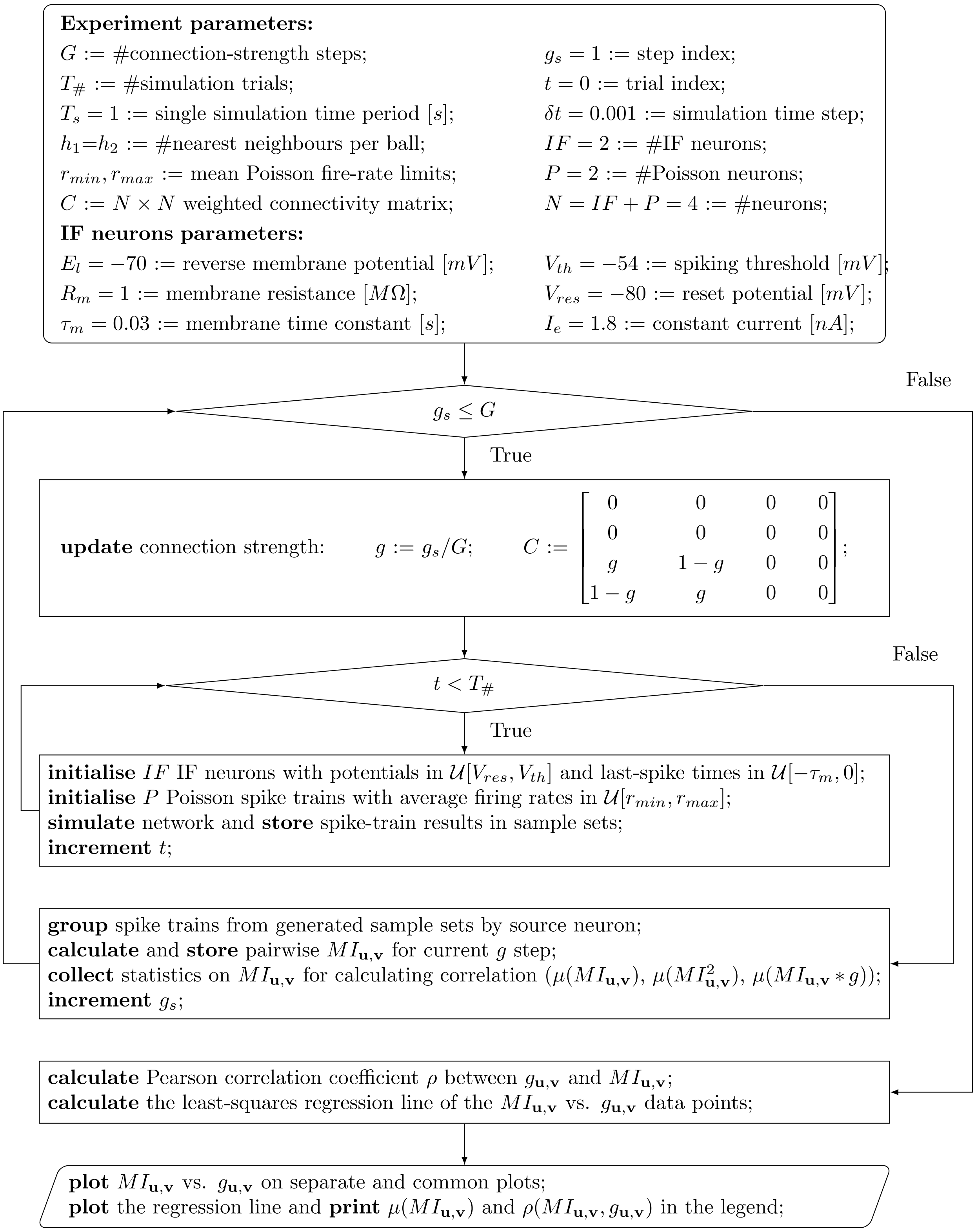

Experiment parameters:
G := #connection-strength steps;
$g_s = 1$:= step index;
$T_\#$:= #simulation trials;
$t = 0$:= trial index;
$T_s = 1$:= single simulation time period $[s]$;
$\delta t = 0.001$:= simulation time step;
$h_1 = h_2$:= #nearest neighbours per ball;
$IF = 2$:= #IF neurons;
r_{min}, r_{max} := mean Poisson fire-rate limits;
$P = 2$:= #Poisson neurons;
$C := N \times N$ weighted connectivity matrix;
$N = IF + P = 4$:= #neurons;
IF neurons parameters:
$E_l = -70$:= reverse membrane potential $[mV]$;
$V_{th} = -54$:= spiking threshold $[mV]$;
$R_m = 1$:= membrane resistance $[M\Omega]$;
$V_{res} = -80$:= reset potential $[mV]$;
$\tau_m = 0.03$:= membrane time constant $[s]$;
$I_e = 1.8$:= constant current $[nA]$;
$g_s \leq G$
True
False
update connection strength: $g := g_s/G$; $C := \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ g & 1-g & 0 & 0 \\ 1-g & g & 0 & 0 \end{bmatrix}$;
$t < T_\#$
True
False
initialise IF IF neurons with potentials in $\mathcal{U}[V_{res}, V_{th}]$ and last-spike times in $\mathcal{U}[-\tau_m, 0]$;
initialise P Poisson spike trains with average firing rates in $\mathcal{U}[r_{min}, r_{max}]$;
simulate network and **store** spike-train results in sample sets;
increment t;
group spike trains from generated sample sets by source neuron;
calculate and **store** pairwise $MI_{\mathbf{u},\mathbf{v}}$ for current g step;
collect statistics on $MI_{\mathbf{u},\mathbf{v}}$ for calculating correlation ($\mu(MI_{\mathbf{u},\mathbf{v}})$, $\mu(MI^2_{\mathbf{u},\mathbf{v}})$, $\mu(MI_{\mathbf{u},\mathbf{v}} * g)$);
increment g_s;
calculate Pearson correlation coefficient ρ between $g_{\mathbf{u},\mathbf{v}}$ and $MI_{\mathbf{u},\mathbf{v}}$;
calculate the least-squares regression line of the $MI_{\mathbf{u},\mathbf{v}}$ vs. $g_{\mathbf{u},\mathbf{v}}$ data points;
plot $MI_{\mathbf{u},\mathbf{v}}$ vs. $g_{\mathbf{u},\mathbf{v}}$ on separate and common plots;
plot the regression line and **print** $\mu(MI_{\mathbf{u},\mathbf{v}})$ and $\rho(MI_{\mathbf{u},\mathbf{v}}, g_{\mathbf{u},\mathbf{v}})$ in the legend;

Figure 9: Flow diagram of Experiment 1.

This experiment enables us to test whether the estimator captures the variation in mutual information as the relationship between the variables changes. Since we have no analytic way of calculating the true underlying quantities of information, we use the synapse strength as a proxy independent variable that puts them into context with our estimates. Although the function we are approximating is unknown, we have an intuition about its linear behaviour, due to our knowledge of the relationship being measured and how it varies. Therefore we measure the correlation and fit a line into the data to investigate the hypothetical linear relationship. There are two different approaches to estimating mutual information in the space of spike trains. Both the traditional time-binning procedure [11] and the investigated metric-space technique are only approximations of the *real* mutual information. We are not going to compare them here because they are principally different. The binning method needs to work at millisecond resolution to be accurate and would only be feasible to compute on very short spike trains ($20 - 50[ms]$). The scale of this experiment is set to $1[s]$ per network simulation and the average spiking rates of Poisson stimuli are in range $(10 - 50)$ in order to capture the information propagation in a very simple network. Ideally the two methods should converge to the ground truth as sample sizes increase. This can be tested through further, more elaborate investigation. However, as long as the estimator captures the linear relationship between MI and g, we can rely on its validity and correctness. Through the simple linear relationship we measure its relative accuracy and performance using correlation analysis. Good performance on this test still leaves margin for error but it narrows it down to bias and systematic error. This will be sufficient for the purposes of this essay.

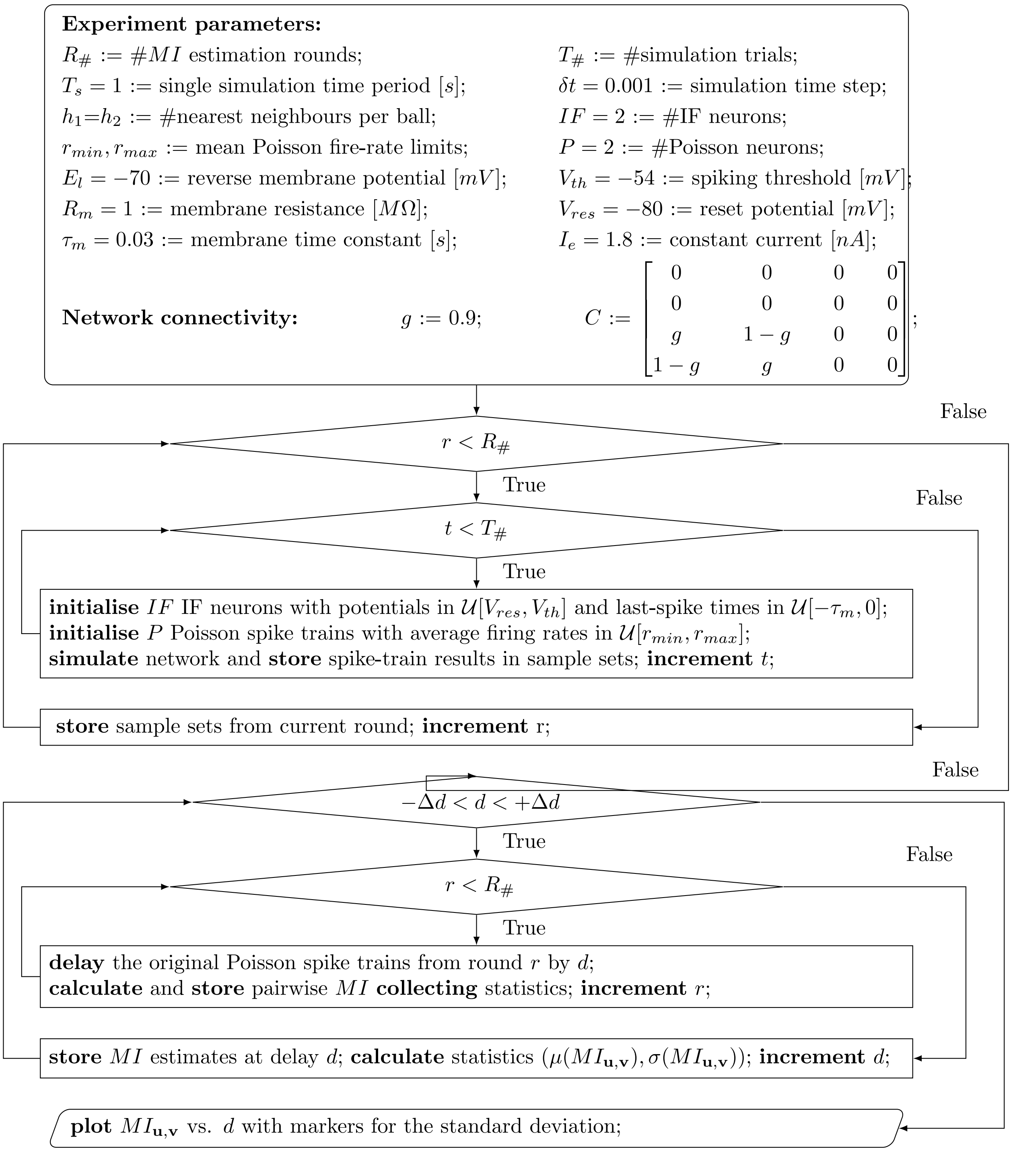
2.4 Experiment 2: Mutual information vs. stimulus delay

This is another example of an experiment where we vary the relationship between the two spike-train variables in a predictable way and examine the resulting variation in mutual information estimates. This time, instead of altering an experimental condition, we meddle with the spike trains themselves. They are shifted by adding or subtracting a delay period from their comprising spike times while preserving the original time scope. This is done for a set of delay periods in a symmetric range around zero. We start by first generating the spike-train data sets from the four neurons multiple times. The process is repeated over a number of rounds $R_{\#}$ in order to estimate mutual information multiple times at each delay step and observe how the average varies with the delay. We iterate through the delay points using the same sample sets to estimate MI but setting a different delay to the originally produced Poisson spike-train they contain. Optionally a specific delay can be added to the original Poisson spike trains in advance. This does not change the experiment in an essential way since the spikes lost through that shift would not be recovered by the iterative shifting afterwards and hence a zero delay is always expected to reflect maximum mutual dependency between the two spike-train variables in question.

We apply the estimator $R_{\#}$ times at each delay and calculate the mean μ and standard deviation σ of these estimates every time. Like in the previous experiment, the initial states of IF neurons and the average firing rates of Poisson neurons are chosen stochastically for every single simulation. We use the same network topology and the synapse strengths are reciprocal like in Experiment 1, but this time they are kept constant - $g_{2,0} = g_{3,1} = g = 0.9$ and $g_{2_1} = g_{3,0} = 1 - g = 0.1$. We will only be interested in measuring the relationship between neurons connected by the strong synapses. There will be variation in the MI estimates at each delay step resulting form the different independent data sets used to produce them. This ensures that the results are representative of the estimator's performance on average but makes them noisy too. We therefore use the mean and standard error statistics to capture the general tendency in the change of mutual information over stimulus delay.

The aim of this experiment is to test the model estimator's sensitivity to the synchronisation of related signals. While the nature and scale of the original relationship is kept the same, the causality it contains is distorted linearly in order to examine whether this is reflected in the estimated mutual information. The exhibition of such a property will further validate the model. In the context of neuroscience, delays between stimuli and responses can be important and sometimes characteristic of the source of neural activity. However, delays between related signals can occur in many other signal-processing contexts and mutual information can be a used for detecting them comparatively and quantifying their effect. We expect the average MI estimates to increase as the negative delay increases towards zero and to then decay as the delay grows in the positive direction.

The control flow diagram below gives a schematic description of the procedure followed for conducting the experiment. Again, the low-level simulation details are abstracted away in favour of the algorithmic structure of the process. The neuron simulation settings are the same as in the first experiment, and analogously the parameters determining the resolution of the experiment are represented by their identifier as they can vary. The implementation details and the exact settings used for running the tests follow in the two subsequent Chapters.



Experiment parameters:

$R_{\#}$:= $\#MI$ estimation rounds; $T_{\#}$:= #simulation trials;

$T_s = 1$:= single simulation time period $[s]$; $\delta t = 0.001$:= simulation time step;

h_1=h_2 := #nearest neighbours per ball; $IF = 2$:= #IF neurons;

r_{min}, r_{max} := mean Poisson fire-rate limits; $P = 2$:= #Poisson neurons;

$E_l = -70$:= reverse membrane potential $[mV]$; $V_{th} = -54$:= spiking threshold $[mV]$;

$R_m = 1$:= membrane resistance $[M\Omega]$; $V_{res} = -80$:= reset potential $[mV]$;

$\tau_m = 0.03$:= membrane time constant $[s]$; $I_e = 1.8$:= constant current $[nA]$;

Network connectivity: $g := 0.9$; $C := \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ g & 1-g & 0 & 0 \\ 1-g & g & 0 & 0 \end{bmatrix}$;

$r < R_{\#}$

$t < T_{\#}$

initialise IF IF neurons with potentials in $\mathcal{U}[V_{res}, V_{th}]$ and last-spike times in $\mathcal{U}[-\tau_m, 0]$;
initialise P Poisson spike trains with average firing rates in $\mathcal{U}[r_{min}, r_{max}]$;
simulate network and **store** spike-train results in sample sets; **increment** t;

store sample sets from current round; **increment** r;

$-\Delta d < d < +\Delta d$

$r < R_{\#}$

delay the original Poisson spike trains from round r by d;
calculate and **store** pairwise MI **collecting** statistics; **increment** r;

store MI estimates at delay d; **calculate** statistics $(\mu(MI_{\mathbf{u},\mathbf{v}}), \sigma(MI_{\mathbf{u},\mathbf{v}}))$; **increment** d;

plot $MI_{\mathbf{u},\mathbf{v}}$ vs. d with markers for the standard deviation;

Figure 10: Flow diagram of Experiment 2.

2.5 Reverse engineering the network

Reverse engineering or network inference is the task of predicting links between variables by analysing their relationship based on a set of data. This is an open problem appearing in many fields of science. It is of significant importance to chemistry, bioinformatics and systems biology in particular - for the reconstruction of the underlying structure of gene regulatory, metabolic and cell signalling networks [26]. Research on how changes in these networks affect information transmission has led to the development of rigorous frameworks for addressing the problem using the tools of information theory. While many of these methods are ad hoc - tailored to a specific type of network, exploiting limiting assumptions - there is also a number of principled approaches addressing the problem in its general form. This implies that no assumptions about the structure of the variables and the nature of their relationship

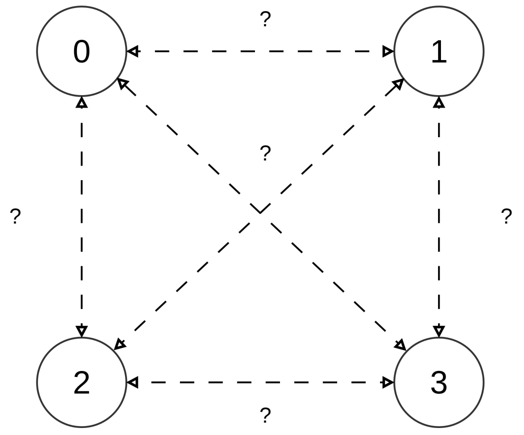


Figure 11

are made. In order to deduce connections using only the statistical features of the data, many of these strategies use mutual information to measure the mutual dependence between variables. There is no best method for tackling this problem and most solutions do not achieve high prediction accuracies, but for small enough networks with simple topologies they can be applied with some success.

In our case, the strength of the synapse between two neurons should be proportional to the estimated mutual information between their signals. In this context MI directly suggests itself as a tool for network inference. If the estimates are accurate we should be able to use them to reconstruct the connections between the four neurons in the experimental network and possibly deduce their relative strengths and directions. We will explore some of the basic principles that can be applied to accomplish this goal.

A review of information-theoretic network inference methods [24, 26], suggests two main approaches to the problem. The Entropy Metric Construction (EMC) method, later extended with the Entropy Reduction Technique (ERT) [10, 14], designed for reverse engineering chemical reaction mechanisms, are based on entropy estimates that we are not interested in in the course of this project. They use mutual information as a distance measure and infer connectivity by minimising the conditional entropy between variables. There is also a CMC - equivalent to EMC but based on correlation. The second approach to reconstructing connections between variables is based on their mutual information and the Data Processing Inequality (DPI) discussed in Chapter one. The ARACNE method [19, 20] applies the DPI to distinguish between direct and indirect interactions. It narrows down to removing the minimum-pairwise-MI edge from each possible triplet of variables with pairwise mutual information above a a certain threshold. There are also extensions of this method considering higher-order indirect interactions and particular types of networks in the context of bioinformatics.

Here we will not be making a scientific inquiry of network inference methods but rather remain interested in examining how indicative the MI estimates obtained using the metric-space formula are of the underlying neural network topology. We will therefore limit our efforts to simply applying DPI in the spirit of ARACNE along with some basic logical inference based on very general assumptions. In order to do this we will first need to generate some experimental results. The observations made on the inference task will be documented along with them in Chapter 4.

3 Implementation

3.1 Overview

The implementation process for this project started with the study of neural electrodynamics and the integrate-and-fire model discussed in Chapter 1.3. Initially a single *IF* neuron was simulated using the Euler method for approximating the function of its membrane potential. Afterwards a simulation of two neurons coupled through a synaptic connection was experimented with. Once some ground was covered on the basic principles of simulating neurons, a more principled approach to simulating a spiking neural network in software had to be adopted. Of course, there is no single best way to do this. The main requirements for this implementation followed from its purpose, but it was also built with the intention to be scalable and reusable for future work. The simulation environment was developed in an object-oriented way in order to achieve more flexibility and generality. The network simulation is designed to support networks of any size and topology containing *IF* as well as Poisson neurons with outward connections only, used as stimuli. The toolset is completed by the two introduced spike-train metrics and the model estimator implementation. This is all compiled in the script "**neuro.py**" which is used as an external library containing all the functionality for implementing experiments in separate scripts.

This Chapter gives a review of all the key components of the Python code implementing the simulation environment and of the implementation process itself. The software in its final version seen here is the result of several iterations of object-oriented construction. Some examples of early simulations for testing the network are given to demonstrate the correctness of the computational model. The code implementing the two core experiments is included in the appendix.

3.2 Integrate and fire neurons

The primary element of our neural network model is the **neuron** object. A snippet containing the code is given in figure 12. It represents a structure storing all the information associated with the state of an *IF* neuron at a given time and all the parameters involved in the differential equation modelling its voltage dynamics. The equation itself is built into it as the function **f**. Its numerical solution used to approximate the voltage of the neuron **v** (V) at discrete time-steps takes place in another object dedicated to simulation because it will depend on the dynamics of other neurons in the network as simulation time proceeds. The times of the spikes recorded by simulating the voltage dynamics - that is the spike train of a **neuron** is stored in a list called **sTrain**, as well as the last spike time explicitly - **sTime**. The **neuron** has an index, **idx**, and a **type**='IF' to identify it in the connectivity matrix of a simulated network and to distinguish it from the other neuron objects storing spike trains generated with the Poisson model. It also stores the *IF* simulation parameters: the reverse potential **e_l** (E_l), the spiking threshold and reset level - **v_th** and **v_res** (V_{th}, V_{res}), the resistance of the membrane **r_m** (R_m), constant input current **i** (I_e), the time constant of the membrane **t_m** ($\tau_m = C_m/G_m$), a refractory period **t_ref** (τ_{ref}), and a reversal potential of the synapse **e_s** (E_s). It does not take account of the synaptic conductance **G_s** (G_s) which is passed as a parameter to **f** as the dynamics of the synapse are dealt with in a separate object.

```python
class neuron (object):
  def __init__(self, idx, v0, st, el=None, vth=None, vres=None, rm=None,
               i=None, tm=None, tref=None, es=None):
    self.id     = idx                             #id in connectivity matrix
    self.type   = 'IF'                            #type of neuron
    self.v      = v0                              #membrane potential
    self.sTime  = st                              #time of last spike
    self.sTrain = []                              #spike times of the neuron
    self.e_l    = E_l   if el   == None else el   #reverse potential
    self.v_th   = V_th  if vth  == None else vth  #threshold potential
    self.v_res  = V_res if vres == None else vres #reset potential
    self.r_m    = R_m   if rm   == None else rm   #membrane resistance
    self.i      = I_e   if i    == None else i    #constant input current
    self.t_m    = t_M   if tm   == None else tm   #membrane time constant
    self.t_ref  = t_Ref if tref == None else tref #refractory period
    self.e_s    = 0     if es   == None else es   #synapse reverse potential

# returns value of f(V) = dV/dt
  def f(self, V, G_s):
    f=(self.e_l - V + self.r_m*G_s*(self.e_s-V) + self.r_m*self.i)/self.t_m
    return f
```

Figure 12: The *IF* **neuron** object (**neuro.py**).

These simulation properties are set with a ternary operator so that they can be read in from both optional input parameters and global variables - an extension not taken advantage of - the former is used throughout the implementation. This is intended for the integration of the whole environment along with the experiments in a command-line interface tool for conducting simulations. The plot below shows the voltage of a single **neuron** simulated for one second, with a constant current just high enough to make it spike. This particular example is produced earlier using the Euler approximation. The voltage is reset every time the threshold is reached.

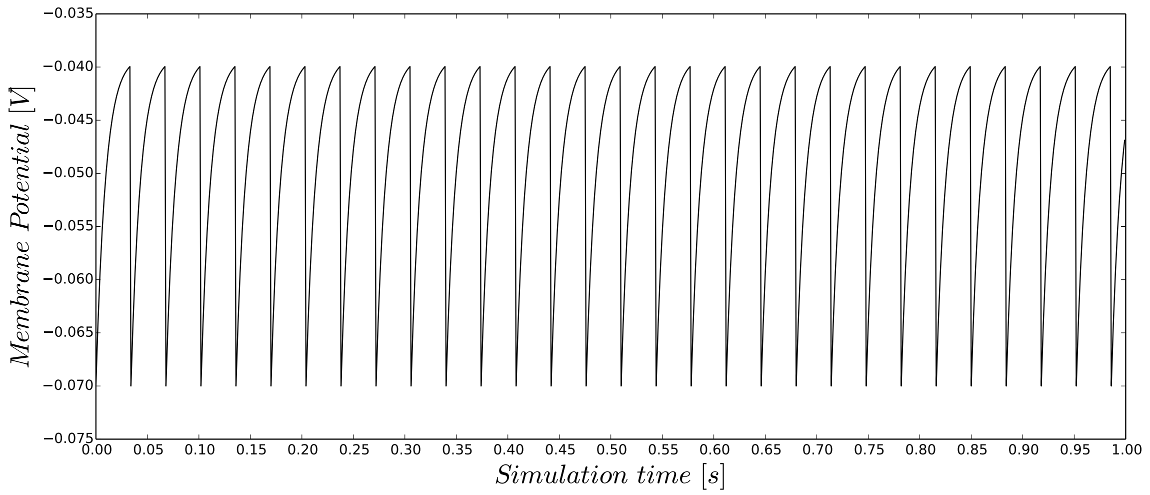


Figure 13: The membrane potential of a single IF neuron stimulated with $E_l = V_{res} = -70$ $[mV]$, $V_{th} = -40$ $[mV]$, $R_m = 10$ $[M\Omega]$, $I_e = 3.1$ $[nA]$, $\tau_m = 10$ $[ms]$ for a period $T = 1$ $[s]$ at time step $\delta t = 1$ $[ms]$, without a refractory period $\tau_{ref} = 0$, and without any synaptic currents.

3.3 Poisson neurons

The other type of neurons, producing spike-train variables representing the stimuli, are implemented using the procedure for generating random spike trains with firing rates coming form a Poisson distribution around a pre-set average, given in algorithm 2. The **pNeuron** object contains an identifier **id**, a **type** equal to 'P', a spike train **sTrain** and last-spike time **sTime**, like the *IF* **neuron**. Its spike train is generated at initialisation based on the **sRate** parameter and its state does not change during simulation apart from updating **sTime**.

```
class pNeuron (object):
  def __init__(self, idx, sRate, T, dt, st):
    self.id     = idx            #index in connectivity matrix
    self.type   = 'P'
    self.sTrain = []             #spike times
    self.sTime  = st             #time of last spike
    self.count  = 0              #spike count over period T
    self.T      = T

    time = np.arange(0, T, dt)
    for i, t in enumerate(time) :
      x = np.random.rand()
      if x < sRate*dt :
        self.sTrain += [t]
        self.count += 1

  def delay(self, d):
    i = 0
    if d >= self.T or d <= -self.T :
      self.sTrain = []
      return
    while i < len(self.sTrain):
      self.sTrain[i] += d
      if self.sTrain[i] > self.T or self.sTrain[i] < 0:
        self.sTrain.pop(i)
      i += 1
```

Figure 14: The **pNeuron** object implementing Poisson-generated stimuli (**neuro.py**).

It also has the **delay** function which shifts all spikes in **sTrain** by a parameter time-period **d**, deleting the ones outside the scope of the simulation period **T**. That is of course, provided **d** is smaller than **T**, otherwise it simply returns an empty spike train. This is the delay used in experiment 2.

A simple test was designed to verify that the procedure for generating spike trains using the Poisson model produces the desired results. It was run a hundred times to generate spike trains of length one second, at time-step $\delta t = 0.001$ $[s]$, with the average firing rate set to 120. The actual rates - the counts of spikes contained in each of the generated spike trains - were recorded and counted. A histogram was then produced, which can be seen in figure 15, plotting the number of generated spike trains of each rate. This histogram will be different every time the test is run, due to the stochastic nature of the process. However it was observed over a number of trials that the resulting distribution over spike rates does have the correct shape and the mass of rates close to the average consistently dominates the mass of rates further away from it.

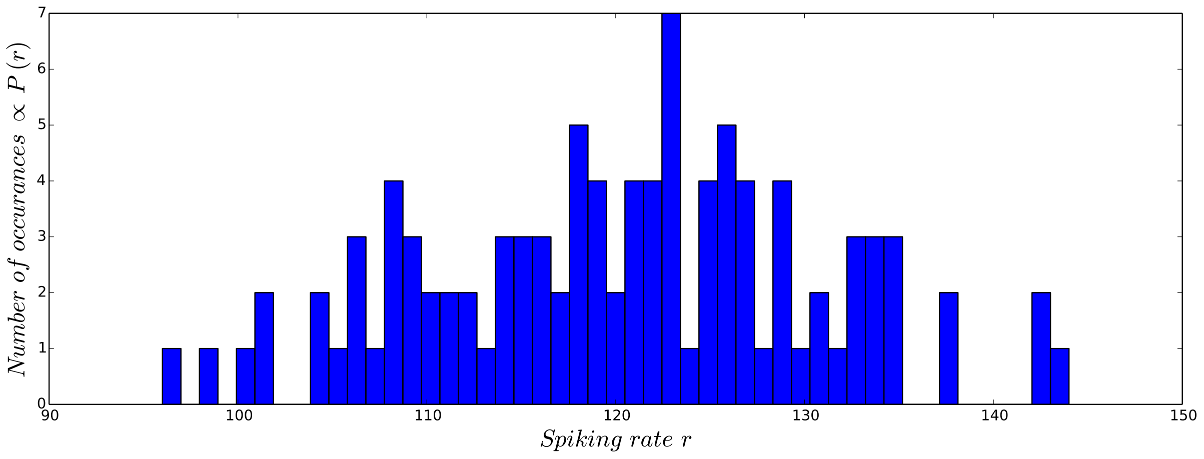


Figure 15: A histogram plotting the numbers of spike trains over their spiking rates for a hundred spike trains generated using the Poisson model with an average rate of 120.

3.4 Synapses

Now that we have the objects for the two types of neurons set up we move on to the one which is responsible for keeping track of the state of the connections between them - the **synapse** object. This will not really represent a single chemical synapse though - that would have meant that the **neuron** objects need to keep track of the individual synapses connecting other neurons to them, or alternatively the job could have been assigned to the connectivity class. Here this has been avoided through a simplification. The **synapse** object keeps track of the states of all the ingoing connections into a **neuron** at a given point in time and sums up all of their conductances to compute the total synaptic conductance of an IF neuron - G_s, at that point. This is only possible under the assumption that all of these actual synapses are of the same type - either inhibitory or excitatory. Strictly speaking, although a neuron is either inhibitory or excitatory, depending on the kind of synapses it forms to connect to other neurons, it can have both types of neurons synapting to it. Here, all the synapses through which an IF neuron in the network receives its inputs are assumed to be of the same kind - depending on the reverse potential for the synapse E_s stored in the **neuron** object itself (zero for excitatory or negative for inhibitory). This in fact is not an unreasonable assumption as it is usually the case in most experiments involving an area of a real nervous system. Most importantly, this simplification does not in any way violate any of our experimental conditions and suits the purpose of the network.

The code for the **synapse** object is given in figure 16 below. The **id** of a **synapse** corresponds to the id of the **neuron** it belongs to - one will be created to manage the synaptic conductance of each IF neuron in the network; **t_s** (τ_s) gives the time-scale which is also assumed to be the same for all the actual synapses the object manages; **sTs** is a list storing the time periods incurred since the last spikes of all the pre-synaptic neurons - these determine the conductances of individual synapses connecting them to the given neuron (see equations 46, 47), and will be updated at every time step of a simulation; **sGs** stores the strengths of these synapses (also involved in the equation for the synaptic currents - eq. 46) - they will remain the same throughout a single simulation and depend on how the connectivity was initialised. The total synaptic conductance is calculated by the **conduct** method. Now, due to the use of the Runge-Kutta method for approximating the membrane potential this is slightly more involved. There are three variables **Gs_1**, **Gs_23** and **Gs_4** stored in the **synapse** object which will be used to compute the four terms defined for the Runge Kutta fourth order expansion (eq. 53, 54). We will revisit these variables when we reach their use in the simulation part, but essentially they approximate the value of the summed conductance at different parts of the time-step - t_n, $t_n + \delta t/2$ and t_{n+1} respectively - to help approximate the voltage in the same fashion.

```
class synapse (object):
  def __init__ (self, idx, dt, ts=None) :
    self.id      = idx                         #index of post-synaptic neuron
    self.t_s     = t_S if ts == None else ts #time-scale of the synapse
    self.dt      = dt
    self.sTs     = []                          #times since pre-synaptic spikes
    self.sGs     = []                          #synaptic strengths
    self.Gs_1    = 0                           #summed synaptic conductance
    self.Gs_23 = 0
    self.Gs_4    = 0

# Sum-up and update synaptic conductance
  def conduct(self):
    self.Gs_1    = 0
    self.Gs_23 = 0
    self.Gs_4    = 0
    for i in range(len(self.sGs)):
      self.Gs_1 +=0.5*self.sGs[i]*exp(- self.sTs[i]/self.t_s)
      self.Gs_23+=0.5*self.sGs[i]*exp(-(self.sTs[i]+0.25*self.dt)/self.t_s)
      self.Gs_4 +=0.5*self.sGs[i]*exp(-(self.sTs[i]+self.dt)/self.t_s)
    return self.Gs_1
```

Figure 16: Code snippet for the synapse object **neuro.py**

3.5 Network connectivity

The connectivity of the neural network is expressed by a connectivity matrix like the ones given in the control flow diagrams of the experiments in Chapter 2. For a network consisting of N neurons this is an $N \times N$ matrix where the i, j-th entry (row, column) stores the strength of the synapse from neuron i to neuron j - taking values between zero and one. The connectivity class **cnet** serves as a mediator of information between neurons by updating their **synapses**. It is passed the connectivity matrix **cMat** and the lists of neurons and their synapses at initialisation - **nrns**, **syns**, and it in turn initialises the lists of connection strengths and times since last spikes in the **synapse** for every **neuron** based on its pre-synaptic neurons. It is created for every network simulation and is used to update the synapses after computing the membrane potentials of the IF neurons at each time step. This is done by calling the **update_sTs** function for every **synapse** of a neuron, which in turn finds the pre-synaptic neurons in **cMat** using the **list_pre_synapt_ns** method. The code for the **cnet** class is given in figure 17 below.

```
class cnet (object):
  def __init__(self, cMat, nrns, syns):
    self.cMat = cMat
#   iterate through neuron/synapse ids
    for j in range(len(syns)):
#     iterate through indices of neurons pre-synaptic to neuron j
      for i in ( self.list_pre_synapt_ns(syns[j]) ):
#       fill list of synaptic strengths and pre-synaptic spikes
        syns[j].sGs.append(cMat[i][j])
        syns[j].sTs.append(0 - nrns[i].sTime)

# update times since pre-synaptic spikes in synapse
  def update_sTs(self, synapse, nrns, t):
#   iterate pre-synaptic neurons ids
    for j,k in enumerate( self.list_pre_synapt_ns(synapse) ):
      synapse.sTs[j] = t - nrns[k].sTime

# return pre-synaptic neurons reaching parameter synapse
  def list_pre_synapt_ns (self, synapse):
    cMat = self.cMat; id = synapse.id
    return [item for sublist in np.nonzero(cMat[:,id]) for item in sublist]
```

Figure 17: Code snippet from the class managing the connectivity of the network (**neuro.py**).

3.6 Network simulation

The core component of the simulation environment is the **netSim** object used to simulate the voltage dynamics of IF neurons in a neural network. The code for it is given in figure 18 Its initialisation takes as input a list of neurons **Nrns**, a connectivity matrix **cMat**, a time period **T**, time-step **dt** and an optional **Nrns** can be of both the **neuron** and **pNeuron** classes defined above. The idea is that these are all initialised in advance and the simulation class is only responsible of generating results. Apart from consecutive simulations this can allow for the system to take in any network following the given format and simulate it. The randomly initialised neuron object used throughout the experiments could have easily been created inside the simulation itself but this more general approach can enable the simulation of specific inputs - for example another type of neurons containing spike-train stimuli can be incorporated. During the initialisation of the simulation the two types of neurons at hand are distinguished between and listed separately in **neurons** and **poissons**, all of them are kept track of in **allNrns**. The synapses for IF neurons, listed in **synapses**, are initialised in the simulation itself as they will only serve its purpose and not store any relevant data.

```
class netSim (object):
  def __init__(self, Nrns, cMat, T, dt, h_t=None):
    self.allNrns  = Nrns
    self.neurons  = []
    self.poissons = []
    self.synapses = []
    self.cMat     = cMat                      #connectivity matrix
    self.t        = np.arange(0, T, dt)       #time array
    self.dt       = dt                        #time step
    self.ht       = 0 if h_t==None else h_t   #chronologic time incurred
    for i in range(len(self.allNrns)) :       #neurons and synapse objects
      if self.allNrns[i].type == 'IF':
        self.neurons.append(self.allNrns[i])
        self.synapses.append(synapse(i, self.dt))
      elif Nrns[i].type == 'P':
        self.poissons.append(self.allNrns[i])
    self.cNet   = cnet(cMat, self.allNrns, self.synapses)
#   arrays storing simulation data - potential and conductivity over time
    self.vSim   = np.zeros([len(self.neurons), len(self.t)])
    self.gSim   = np.zeros([len(self.neurons), len(self.t)])
    self.raster = np.zeros([len(self.allNrns), len(self.t)])*np.nan

# Compute membrane potential at a single timeslice using RK4 appximation
  def getV(self, nrn, t, Gs_1, Gs_23, Gs_4):
    if nrn.v >= nrn.v_th :              #record spikes and reset potential
      nrn.v = nrn.v_res
    elif t < nrn.sTime + nrn.t_ref :    #hold reset if refracory period
      nrn.v = nrn.v_res
    else :                              #inegreate using RK4 method
      k1 = self.dt*nrn.f(nrn.v, Gs_1)
      k2 = self.dt*nrn.f(nrn.v + k1/2, Gs_23)
      k3 = self.dt*nrn.f(nrn.v + k2/2, Gs_23)
      k4 = self.dt*nrn.f(nrn.v + k3, Gs_4)
      nrn.v += 1/6*(k1 + 2*k2 + 2*k3 + k4)
    return nrn.v

# Simulate network
  def simulate(self):
    for i, t in enumerate(self.t) :
      for j in range(len(self.synapses)) :              #update conductances
        self.cNet.update_sTs(self.synapses[j], self.allNrns, t)
        self.gSim[j,i] = self.synapses[j].conduct()
      for j in range(len(self.neurons)) :               #update potentials
        self.vSim[j,i] = self.getV(self.neurons[j], t, self.synapses[j].Gs_1,
                         self.synapses[j].Gs_23,self.synapses[j].Gs_4)
        if self.neurons[j].v >= self.neurons[j].v_th : #record IF spikes
          self.neurons[j].sTime = t
          self.neurons[j].sTrain += [self.ht + t]
          self.raster[j][i] = j
      for p in range(len(self.poissons)):               #check for P-spikes
        if t in self.poissons[p].sTrain :
          self.poissons[p].sTime = t
          self.raster[self.poissons[p].id][i] = self.poissons[p].id
    for j in range(len(self.allNrns)) :                 #reset sTimes
      self.allNrns[j].sTime = - (t - self.allNrns[j].sTime)
    return self.vSim, self.gSim, self.raster
```

Figure 18: Code Snippet from the network simulation.

The results from the simulation are output in three formats - two 2D arrays - **vSim** and **gSim** - storing the computed voltages and synaptic conductances for each **neuron** at each time step as well as another 2D array of “nan” Python objects - **raster** - marking the occurrences of spikes from each neuron over the simulation period, used to produce a raster plot of the simulation. The raster plot simply plots a dot for every spike elicited by a neuron on its own time-line of the simulation. An example is given in figure 21, put into context later on. These outputs are for testing purposes only. They are never used in the experiments - only the recorded spike trains stored in each neuron are taken account of in order to compute mutual information. However, a dedicated object called **experiment** is created in **neuro.py**, for storing all the simulation results produced during an experiment, which can include the 2D arrays, as well as the neurons containing the spike trains. The production of these arrays impacts the time efficiency, but they are kept in the simulation procedure for convenience and debugging purposes.

Before moving on to the **simulate** method, implementing the actual simulation, we take a look at the function **getV**, used to compute the membrane potential of an IF neuron at a single time step. As previously discussed this is accomplished by applying the Runge-Kutta method for numerical integration. Here is where the three **Gs_#** parameters computed in **synapse.conduct** come in - they are passed as parameters to **getV** along with the **neuron** object - **nrn**, and the current time-stamp **t**. Before the membrane potential in **nrn.v** is updated it is checked if its value has reached the the spiking threshold, in which case it is set to the reset value **nrn.v_th**. If that in turn has occurred less than one refractory period **nrn.t_ref** ago the reset value is maintained. Otherwise the membrane potential is approximated by calculating the Runge-Kutta fourth order coefficients with the according **Gs_#** parameters.

The simulation consists of a main loop iterating through step indices and values in the time array **t**. Every iteration of this loop is split into three parts: a loop through the list of **synapse** objects for updating the synaptic conductances, a second one for updating the membrane potentials in **neuron** objects using **getV** and for recording spikes in their last-spike times **sTime**, their spike trains **sTrain** and the raster plot, and another one for updating the last-spike time **sTime** every time a new spike is reached in the pre-generated spike trains of **pNeurons** and recording them in **raster** too. An extra loop in the end of the simulation, after the main loop, is added to set the last-spike times in IF neurons to minus the time elapsed since their occurrence for their use in consecutive simulations. Two test experiments were conducted to verify that the network simulation operates correctly which are documented below.

Two coupled neurons

This first, very basic experiment simulates a network consisting of just two IF neurons synapting onto each other. It is run twice - with excitatory synapses ($E_s = 0$), and with inhibitory ones ($E_s = -80$ $[mV]$). The intention is to examine whether the synaptic stimulation affects the neurons' voltage dynamics in the expected way. The spikes of excitatory neurons make the neurons that they synapt to more likely to spike. Because of the cyclic coupling between the two neurons their spiking must become mutually dependent and therefore we expect their spiking times to come closer together over time, until they eventually synchronise. On the contrary, an inhibitory connection between neurons means that every pre-synaptic spike makes the post synaptic neuron less likely to fire, and consequently the spiking times of two coupled inhibitory neurons should diverge. The initial voltages of both neurons are assigned random values between the spiking threshold and the reset level. The rest of the settings used for the two network simulations are the same except for the reversal potential of the synapse E_s. The reversal potential of the membrane is $E_l = -70$ $[mV]$; the threshold $V_{th} = -54$ $[mV]$; the reset level $V_{res} = -80$ $[mV]$; the membrane's resistance $R_m = 10$ $[M\Omega]$; the constant input current $I_e = 1.8$ $[nA]$; the time-scale of the membrane $\tau_m = 20$; $[ms]$; the simulation period $T = 1$ $[s]$ and time step $\delta t = 1$ $[ms]$; again no refractory period $\tau_{ref} = 0$. The actual values assigned in the code for this experiment follow the strictly the scale of the metrics given here, which is somewhat realistic. This is dropped in the code for the two core experiments converting everything to a scale such that the resistance of the membrane equals 1 (rather than 10^7) in order to avoid working with large numbers. The values themselves are not exactly arbitrary either - they too are somewhat realistic or chosen deliberately. In the case of the constant stimulation current I_e, here it is lower than the one used for a single neuron simulation due to the presence of the synaptic currents. The strengths of the synapses between the two neurons are both set to 1. The discrete voltage data output by the two simulations is plotted for each pair of coupled neurons in figure 19.

The observed results have met our expectations consistently over multiple experimental trials. As visualised in the plot above, the excitatory neurons synchronise while the inhibitory ones diverge their signals. The synaptic connectivity of the network seems to be in order and we move on to a more involved experiment testing the dependencies between spike trains produced by the neural network more explicitly.

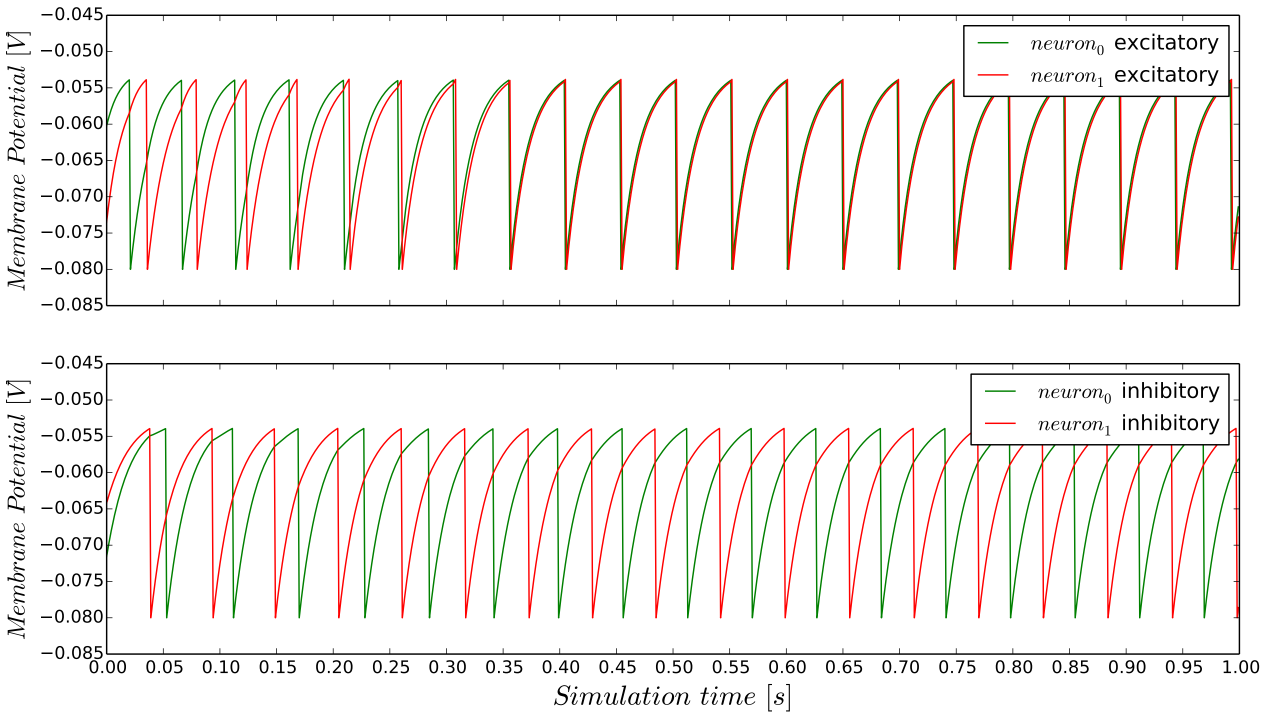


Figure 19: **(top)** The voltage dynamics of two randomly initialised coupled excitatory neurons - $E_s = 0$ - their spiking synchronises over time; **(bottom)** The synapses between the two neurons are inhibitory - $E_s = -80$ $[mV]$ - and their spike times diverge.

Propagation network

In this experiment we construct a network of IF neurons that has a layered topology and inspect how spike signalling propagates through it. The layout is given in figure 20. The network consists of sixteen neurons, plotted in four layers of four - the vertical columns of vertices. There is full connectivity between the layers, directed to the right in the graph. All of these synaptic connections are excitatory and have the same strength equal to 0.6. Initially, all the neurons are stimulated with the same level of direct input current $I_e = 1.8$ $[nA]$. Most of the other settings are the same as in the coupled neurons experiment, except for the time constant of the membrane $\tau_m = 30$ $[ms]$ and the refractory period $\tau_{ref} = 5$ $[ms]$. Here again, every neuron is initialised with a random voltage and last-spike time. The left-most layer is only stimulated by the constant input current while the following three receive synaptic currents from all the neurons in the previous layer as well. The spiking of neurons in one layer should cause an increase in the firing rates of neurons in the next layer. Therefore as long as the stimulation current is the same throughout the network, neurons in each consecutive layer are expected to have higher firing rates than the ones in the previous layer.

In order to produce some more complex behaviour we stimulate some of the neurons with a higher current for a short period of time. The simulation begins and proceeds for a while until at some point the input current I_e of neurons in the left-most layer is put up to 7.2 $[nA]$. This level is maintained for a short period of time, after which it is set back to the original level for the rest of the simulation period. This is realised by running three consecutive simulations of the same network as enabled by the **netSim** class. The external stimulation current is usually used as a constant background input as opposed to a more structured stimulus. This is why it is not implemented as an input parameter of the simulation, defined for every time-step. Instead, it is has been made possible for the same network to be simulated time and again, keeping track of the whole spike trains but outputting new voltage, conductance and raster maps every time. For the purpose of this experiment these results are assembled together for the whole period of the experiment.

We want to observe how the temporary increase in the stimulation current boosts the spiking frequency of neurons in the left-most layer, which in turn speeds up the spiking of neurons in the following ones. On the other hand, due to the layered full connectivity the spiking rates should generally increase in each consecutive layer. Overall

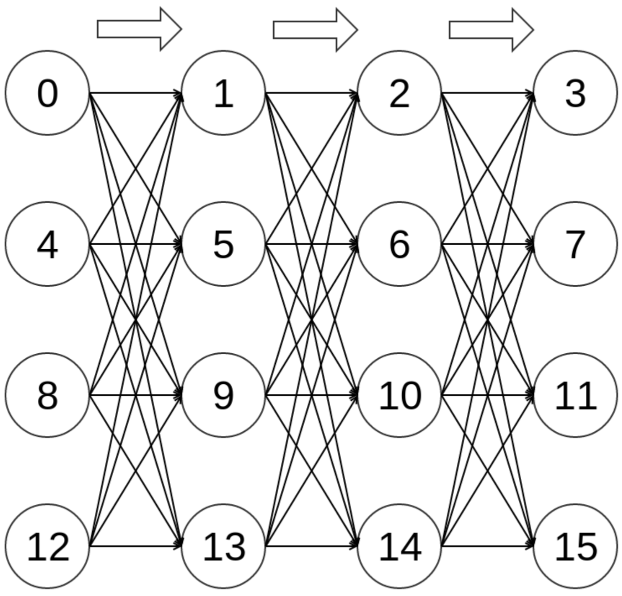


Figure 20: Layered network connectivity for the signal propagation experiment: each vertical layer of neurons is fully connected to the next one - left to right. All synaptic strengths are set to 0.6.

we expect to observe both of these effects and maybe some structure due to their joint presence.

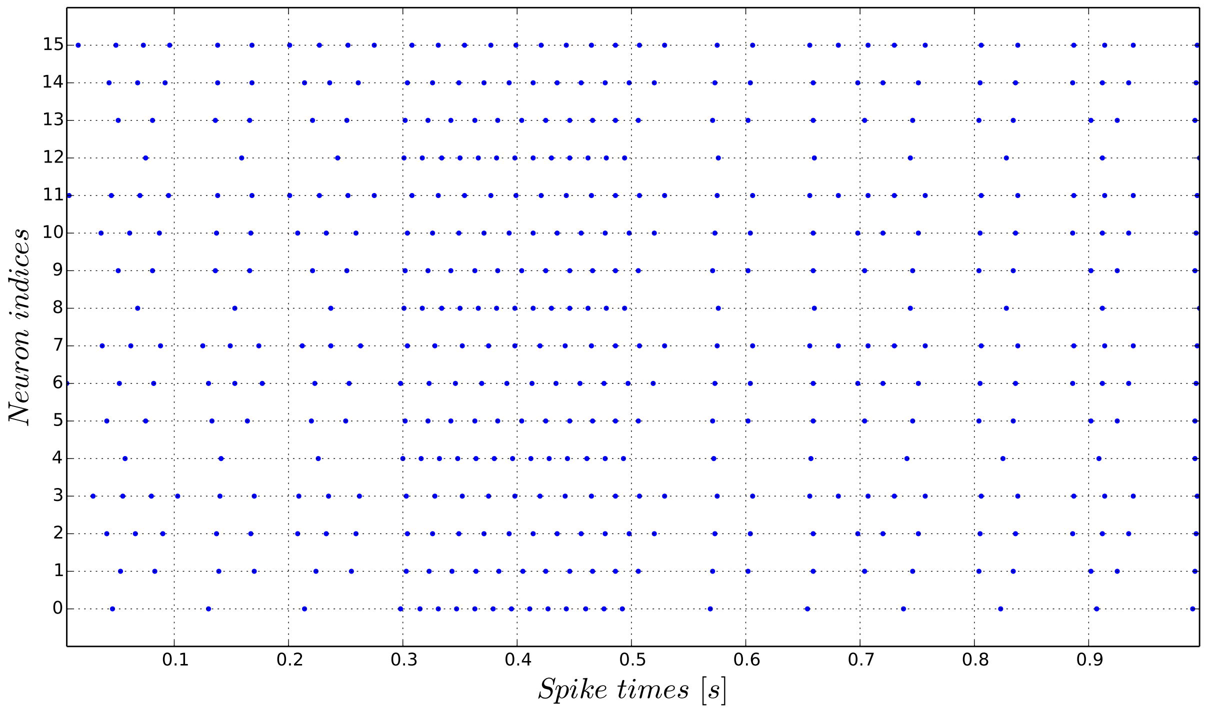


Figure 21: Raster plot of the results from the layered network experiment. Every row on the y- axis (0-16) corresponds to the spike train of a neuron as indexed in the graph in figure20.

The complete raster plot of the experiment is given in figure 21. The stimulation current for neurons 0, 4, 8 and 12 is raised 300 milliseconds into the simulation for 200 milliseconds, after which it is reset for another 500 milliseconds. During the increased- stimulation period we can clearly see all of the neurons firing quicker than they do through the rest of the simulation. Taking a closer look, we observe that while the firing rates of neurons in each consecutive layer increase during the first and third part of the simulation, they decrease in the same order during the second part - the period when the left-most layer is stimulated with current four times higher than the rest. The excitation effect fades as it progresses through the right-most three layers stimulated with the low current throughout. The total currents they receive are still smaller than the increased stimulation current of the left-most layer. In addition to this, there is some pattern to the excitation as the signal amplifies through the layers. The spikes of neurons from the three right-most layers are not exactly rhythmic but rather come in burst patterns of increasing length appearing periodically with the pulse of the left-most stimuli. This expected due to the non-linear influence of synaptic currents.

The results from this experiment verify that the implemented neural network model operates correctly. The neurons' responses to varying simulation conditions confirm that their signals propagate properly through the network. We can therefore rely on the experimental environment to produce spike-train variables with the intended relationships between them. Although all the data that will be used to test the information estimator is produced by this deterministic system it is guaranteed too have the statistical properties characteristic to neural signalling under our simplified experimental conditions and the integrate-and-fire model assumptions. This kind of simulated data serves as the benchmark for the mathematical hypothesis tested in this project. Positive results on it would serve as a clear indication of the applicability of the mutual information estimator to real neuroscientific and other data, which can be the subject of further research. The rest of the functionality of the experimental environment **neuro.py** implements the mutual information estimator along with the spike-train metrics and some auxiliary elements.

3.7 Computing distances

In order to apply the metric-space information estimator we first construct functions implementing the spike-train metrics. The van Rossum kernel-based distance is computed in a function called **vR_computeDistance** - the code for it can be seen in figure 22. It is passed two spike-train parameters **u** and **v**, and the time scale of the kernel **tau** (τ). We use $\tau = 12$ $[ms]$ throughout - a value tested to produce good results [21]. The explicit function mapping through the exponential kernel filter is avoided as we are only interested in the value of the distance. The metric is calculated in the form given in equation 37 - the three double loops add up the tree involved sums and the square root is returned in the end.

```
def vR_computeDistance(u, v, tau) :
  dist = 0
  for i in range(len(u)):
    for j in range(len(u)):
      dist +=   math.exp(- abs(u[i] - u[j])/tau)
  for i in range(len(v)):
    for j in range(len(v)):
      dist +=   math.exp(- abs(v[i] - v[j])/tau)
  for i in range(len(u)):
    for j in range(len(v)):
      dist -= 2*math.exp(- abs(u[i] - v[j])/tau)
  dist = math.sqrt(dist)
  return dist
```

Figure 22: The van Rossum distance function implementation (**neuro.py**).

```
def VP_computeDistance(t1, t2, q) :
# manage corner cases
  if len(t1) == 0 and len(t2) == 0:
    return 1000
  if len(t1) == 0 :
    return len(t2)
  if len(t2) == 0 :
    return len(t1)
# compute using dynamic programming algorithm
  G = np.zeros([len(t1), len(t2)])
  G[:, 0] = [i for i in range(len(t1))]
  G[0, :] = [i for i in range(len(t2))]
  for i in range(1, len(t1)) :
    for j in range(1, len(t2)) :
      G[i][j] = min(G[i-1][j-1] + q*abs(t1[i]-t2[j]), G[i-1][j]+1, G[i][j-1]+1)
  return G[-1][-1]
```

Figure 23: The Victor-Purpura metric function (**neuro.pty**).

The Victor-Purpura edit-length distance is computed in the function **VP_computeDistance** (figure 23 above) following the dynamic programming procedure given in algorithm 1. It also takes as parameters two lists of ordered spike times **t1** and **t2**, as well as the coefficient **q** - the cost per distance in time for moving a spike. The basic step of cost 1, for insertion an deletion, sets a scale on the metric such that $q\delta t$ must be in the range $(0, 2)$ for a spike train to be moved over a distance δt - otherwise it would be cheaper to delete it and insert it in the destination position. Therefore we need to set the time-scale of the metric by choosing *deltat* to be the maximum time scope for moving a spike and dividing 2 by it to determine **q**: $q < 2/\delta t_{max}$. We choose δt_{max} to be the same magic value of 12 $[ms]$ used for the van Rossum metric. We therefore use $q < 2/0.012 = 166$ as the cost coefficient throughout.

We also need some data structure to store the calculated pair-wise distances for a set of spike trains in an organised way. We therefore implement another function, **getDistanceMap**, computing the standard-type distance matrix of a list of spike trains, **trains**, given a parameter distance metric, **metric** - a simple object is created to store the identity of the metric used. For a set with N entries this is a $N \times N$ matrix where every i, j-th entry contains the distance between spike trains i and j, as indexed in the set. In our case this matrix will be symmetric as the distance metrics themselves are symmetric $d(\mathbf{u}, \mathbf{v}) = d(\mathbf{v}, \mathbf{u})$ - computing both triangles twice is avoided in the function as these matrices will be computed very many times during experiments - once for each set of spike trains produced by each neuron in a MI estimation round. Still the whole matrix is filled as this makes the ordering in the estimator easier to write. The code for the distance map is given in figure 24 below.

```
def getDistanceMap(trains, metric):
  N = len(trains)
  dMap = np.zeros([N,N])

  for i in range(N):
    for j in range(N):
      if dMap[j][i] != 0 :
        dMap[i][j] = dMap[j][i]
      elif i != j :
        if metric.type == 'vR':
          dMap[i][j] = vR_computeDistance(trains[i], trains[j], metric.tau)
        elif metric.type == 'VP':
          dMap[i][j] = VP_computeDistance(trains[i], trains[j], metric.q)

  return dMap
```

Figure 24: A function to calculate the distance matrix of a set of spike trains **neuro.py**.

The two implemented metrics have slightly different but very similar time complexities, both dependent on the numbers of spikes in the trains. If their lengths are m and n, then the van Rossum calculation **vR_computeDistance** is dominated by $O(m^2)$ or $O(n^2)$ - whichever is bigger, while **VP_computeDistance** is $O(mn)$ but it also takes $O(mn)$ extra space allocation for the dynamic programming array, which in Python does not come cheap in terms of time. However we are not too concerned with efficiency here. Both of the metrics are implemented due to their simplicity and in order to cross-verify the experimental results using two of the spike-train distance measures most relevant to computational neuroscience.

3.8 Estimating mutual information

The implementation of the mutual information estimator is simple and straight-forward. The code for the function **computeMI** is given in figure 25. It takes five parameters: two sets of spike trains, **S** and **R**, of equal sizes, representing the outcomes of a stimulus and response metric-space variables, a spike-train metric, **metric**, and the sizes of the two open balls **h1** and **h2** - that is the numbers of nearest-neighbour spike trains contained in the regions around instances from the two sets, used to estimate their probability mass. These regions need to be smaller than the size of the sample set. Two distance maps are constructed - one for each set of outcomes. Every stimulus spike-train **s** from **S** corresponds to a response spike train **r** from **R** with the same index. We want to find the subset of **h1** closest spike trains to **s** in **S** and the one of **h2** closest spike trains to **r** in **R**, and then count the stimulus-response pairs shared across both subsets. This is accomplished using **numpy.argsort()**, by sorting in ascending order the row in the distance matrix corresponding to the spike train in question, and taking the original column indices of the first **h1** or **h2** entries respectively. The size of the cross-section of the two subsets of indices is then calculated on line 376. This computes us the $\#[S(s_i, r_i, h_1, h_2)]$ term of equation 32 and we apply the MI formula for two metric-space variables adding up the logarithms in the loop (line 377) and normalizing by N in the end (378). Every spike-train instance is counted in the neighbourhood around it so that the size of cross-section of the two regions for a given stimulus-response pair would always be greater or equal than one. This serves as a statistical correction and ensures that the probability estimates will always be positive and their logarithms will be successfully computed.

```python
def computeMI(S, R, metric, h1, h2) :
  if len(S) != len(R) :
    raise Exception("|S| != |R| !")
  N = len(S)
  if h1 >= N or h2 >= N:
    raise Exception("h1 or h2 >= N !")
  h1h2 = h1*h2
  MI = 0
  dMap_S = getDistanceMap(S, metric)
  dMap_R = getDistanceMap(R, metric)
  for i in range(N):
    b_si = dMap_S[i].argsort()[:h1+1]
    b_ri = dMap_R[i].argsort()[:h2+1]
    count = len(np.intersect1d(b_si, b_ri))
    MI += math.log((N*count)/h1h2, 2)
  MI = MI/N
  return MI
```

Figure 25: The function calculating mutual information between stimulus-response pairs from two spike-train sets by applying the metric-space estimator $I(R;S) \approx \frac{1}{N}\sum_{i=1}^{N} \log_2 N\#[S(s_i, r_i, h_1, h_2)]/h_1 h_2$.

This concludes the implementation documentation. The development of the software for this project in Python involved a lot of learning about the language and decision making on the design of some of the implemented components. Though some of the decisions could have been different from the standpoint of the gained experience, it has been a dynamic and creative process overall.

The next Chapter discusses the results produced using the software tools developed during the course of this project and their implications for the model estimator.

4 Results

The experiments described in Chapter 2 were run many times with various settings configurations and using both of the implemented spike-train metrics. In this Chapter we examine the results that were produced, taking account of the exact experimental parameters used and their effects. To compensate for the lack of comparison with the standard time-discretisation approach to estimating mutual information between spike trains, by Bialek and Strong [11], we investigate how the MI estimates produced using the metric-space model vary as sample sizes grow. We are interested in seeing if they appear to converge to some underlying ground truth which is not readily accessible. Finally, we apply the network inference principles discussed in Chapter 2.5 to try and gain insight about the connectivity between neurons in a network based on the mutual information between their activity.

4.1 Experiment 1

In this experiment we investigate the relationship between the strength of the synapse connecting two neurons and the estimated mutual information between their spike-trains. The code implementing the procedure, described schematically in figure 9 in Chapter 2.2, can be seen in the Appendix. The most important experimental parameters are the ones involved in the estimator itself - the number of simulation trials per estimation - $T_{\#}$ - that is the size of the stimulus-response sample set, and the sizes of open balls h_1 and h_2. The former is easier to decide on - the bigger the sample set, the more representative it should be in general. The resolution of the model given by the region sizes requires some more attention.

The two assumptions that probability estimation in nearest-neighbour regions relies on need to be kept in mind. The regions must be large enough to provide meaningful statistics but small enough to avoid weakening too much the assumption that the probability is constant throughout their members. This problem is approached here only empirically - without trying to determine any optimal values. In order to attempt the last, some mechanism would need to be devised and put at work on every estimation sample set. Since the structure of the underlying metric spaces is hidden, this task is not straight-forward. One approach that could be adopted and has been used with some success in other spaces without coordinates such as Hilbert spaces, is to try to calculate the fractalisation coefficient of the underlying probability space. This is a measure of the roughness of a jagged edge, surface or other space that has emerged from the problem of measuring the lengths of real- world coastlines which appear to have different lengths when measured with metric units of different precision. The coefficient of fractalisation is determined by the slope of the linear relationship between the logarithm of the measured length and the magnification factor of the used unit. This would need to be adapted to use the relationship between the probability estimates and the size of the regions. However it is not certain whether the concept is directly transferable across to estimating the structuredness of the probability space corresponding to the metric space of our spike trains. This will require a more involved investigation and is therefore left out of the scope of this project since trial-and-error quickly turned out to produce satisfactory results.

The sizes of the two regions in the metric spaces of the two spike-train variables involved in mutual information estimation are set to be equal throughout. The experiment was initially run with $h_1 = h_2 = 10$ nearest-neighbour regions over sample sets of size 30 at ten estimation points, and straight away an increasing linear relationship between the strength of the connection and the MI estimates was recognisable. This early positive result determined the direction of the experimental process. Different values for the ball sizes ranging between 10 and 50 were experimented with, but this turned out to have little influence on the outcomes of the experiment. Since one of the requirements for the balls is to be rather small, most later trials were conducted with 12 or 14 samples per region. This aspect of the process is not analysed rigorously here because the estimator exhibited the desired behaviour immediately, without the need for a tedious search for the right parameters. It appears to deliver sensible results as long as the balls are not too large. These results might be systematically biased but there could be many different reasons for that, and apart from setting the optimal resolution the model may need additional corrections. Refining the estimator's requires a more in-depth analysis supplied with more diverse and voluminous experimental data. It would be easier to do this for a type of data such that the exact mutual information and the error of the estimates can be can be calculated directly analytically. This kind of tests could justify the estimator's application to a problem where such data is relevant and the aim is to avoid using the standard formula for efficiency or other reasons.

In the neuroscientific context however, the problem of calculating information in metric spaces without coordinates addressed by the model estimator is precisely that there is no direct analytic solution due to the complex nature of the space of neural spike trains. In this perspective any model can only be refined by putting more experimental data into context and trying to tune its convergence behaviour on growing sample data sets. If there was a good known method it could serve as a reference point, but in our case the other familiar approach is not at all guaranteed to lead to better results since it does not use the spike-train tailored metrics. In addition to this it is extremely inefficient and would hugely increase the computational cost of both experiment simulations and information estimation.

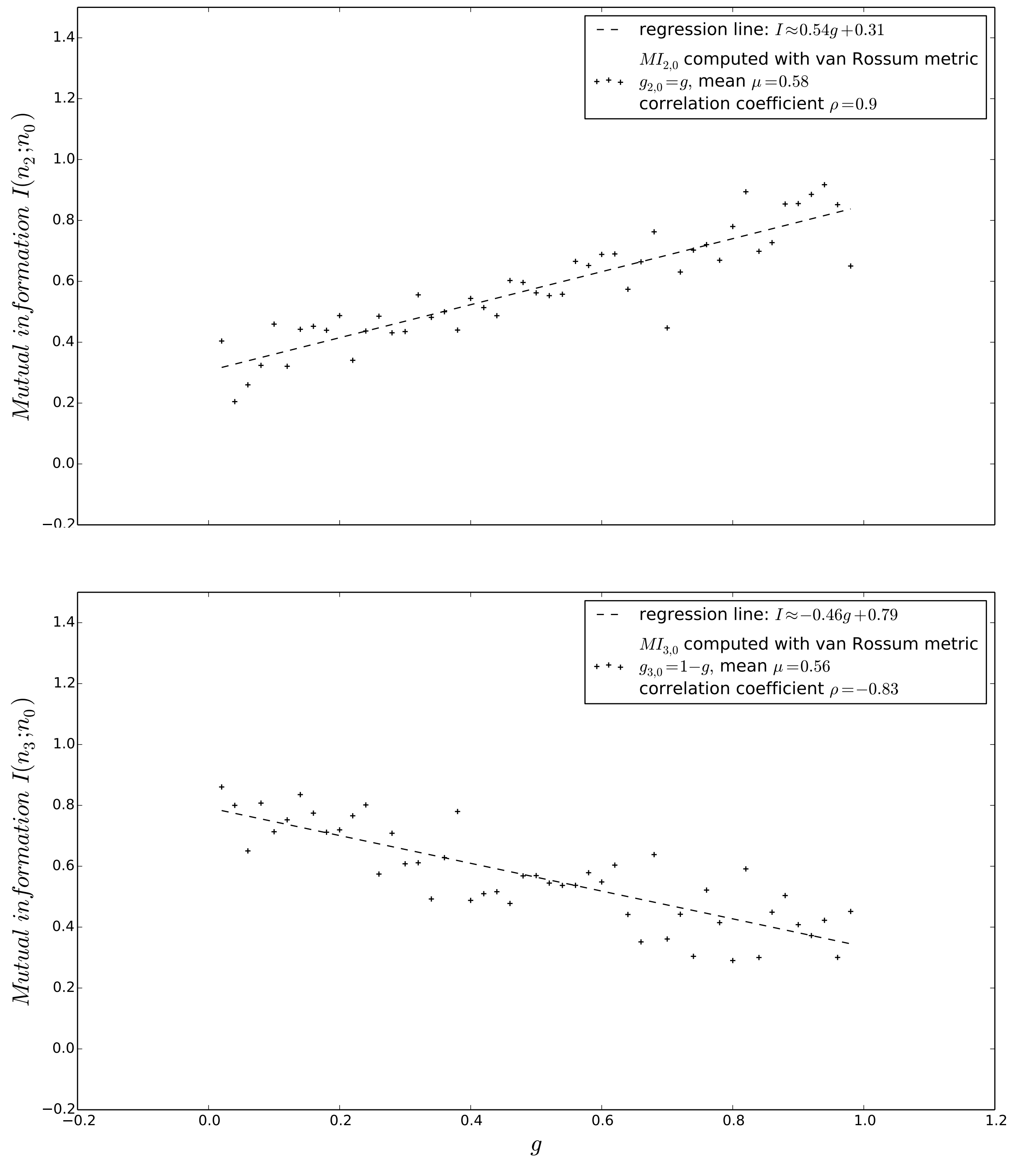


Figure 26: MI computed at 50 steps of g with $T_{\#} = 48$ sampling trials and ball-sizes $h_1 = h_2 = 12$. (Top) MI between neurons 2 and 0 - positive correlation with the connection strength g with Pearson coefficient $\rho = 0.87$ and regression slope 0.87. (Bottom) MI between neurons 3 and 0 - negative correlation as the strength of the synapse equals $1 - g$.

This essay aims to build on the dummy-data results from the clustering task performed with the initial version of the metric-space model, introduced in [23]. By demonstrating that the estimator performs well on a more elaborate task using data generated with a computational neural model we open the door for its future applications to neuroscientific and other problems. Figure 26 contains a plot of the estimated mutual information over growing connection strength between neurons indexed 2 and 0 in the experimental network. This result is obtained by running experiment 1 to compute mutual information at 50 steps of the synapse strength g between zero an one using the Victor-Purpura metric with sample sets of size $T_{\#} = 48$ and region sizes $h_1 = h_2 = 12$. There is a clear positive correlation between the two variables even using this relatively small sample size. The Pearson correlation coefficient of the estimates confirms this with a high positive value of $\rho \approx 0.87$. The regression line has a slope of 0.63 and although there is some deviation around it, it is consistently symmetric along the line and no distinct outliers are present. The relationship between the spike-trains of connected neurons is captured by the mutual information estimator which confirms our hypothesis. Very similar results were obtained using the van Rossum metric with an exponential kernel. The resolution of 50 sample points over the synaptic strength is preserved throughout the rest of the presented experiments. This value is tight enough to capture the tendency in the variation of the estimates and its importance to the overall accuracy is only symbolic. The parameter we are interested in varying from here on is the size of the spike-train sample sets.

The figure above also shows the MI estimates between neurons 3 and 0 where the synapse strength is $1-g$. Another way of verifying that the relationships between neurons are reflected in the estimates is by comparing the mutual information between spike-train responses and stimuli with an increasing connection strength between them, to the mutual information between the same responses to simultaneous stimuli with a weakening connection strength. The two connection strengths vary reciprocally due to the connectivity of our experimental network. We do this by first ordering the MI estimates between one pair of neurons and then plotting against them their corresponding estimates between the other pair. figure 27 gives an image of this plot. As expected, there is an inverse-proportional relationship between the two sets of mutual information estimates. This means that as the dependence of neuron 0's spiking on the pre-synaptic spiking of neuron increases 2 its dependence on the spiking of neuron 3 decreases and vice versa.

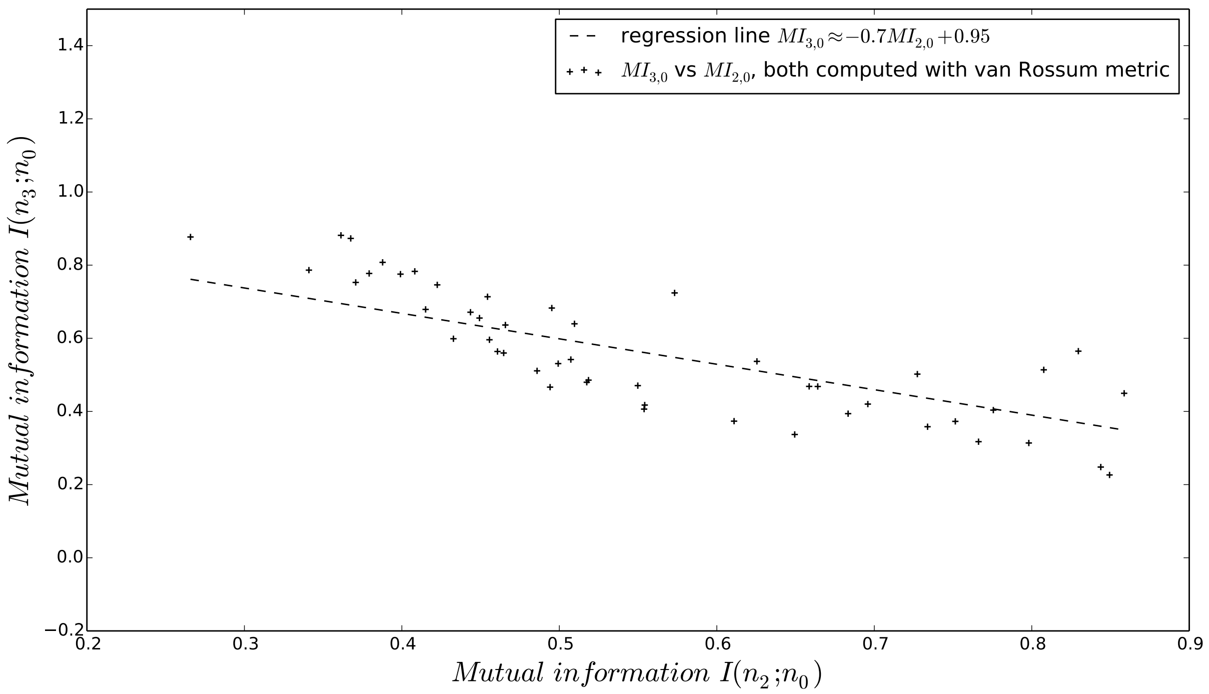


Figure 27: Mutual information between neurons 0 and 1, computed at 50 synapse strength steps with $t_{\#} = 48$ stimulus-response samples and ball-sizes $h_1 = h_2 = 12$. There is a clear linear relationship with correlation coefficient 0.87, modelled by a regression line of slope 0.87.

In order to investigate the mutual information estimator's response to growing sample sizes the experiment was run nine times starting with $T_{\#} = 16$ simulation trials per estimation and stepping up by 16 each time. The region sizes were fixed at $h_1 = h_2 = 14$ for every round. The outcomes of the experiment run with sample sizes 32, 64, 96 and 128 are displayed in figure 28, for estimates obtained with the van Rossum distance, and in figure 29 for the ones when the Victor-Purpura metric is used. As the number of spike trains in the sample sets increases the

the estimates slightly shift their values upwards and the slope of the linear relationship between them and the connection strength rises. Here again the mutual information between neurons 2 and 0 is taken as example, with the synapse between them strengthening.

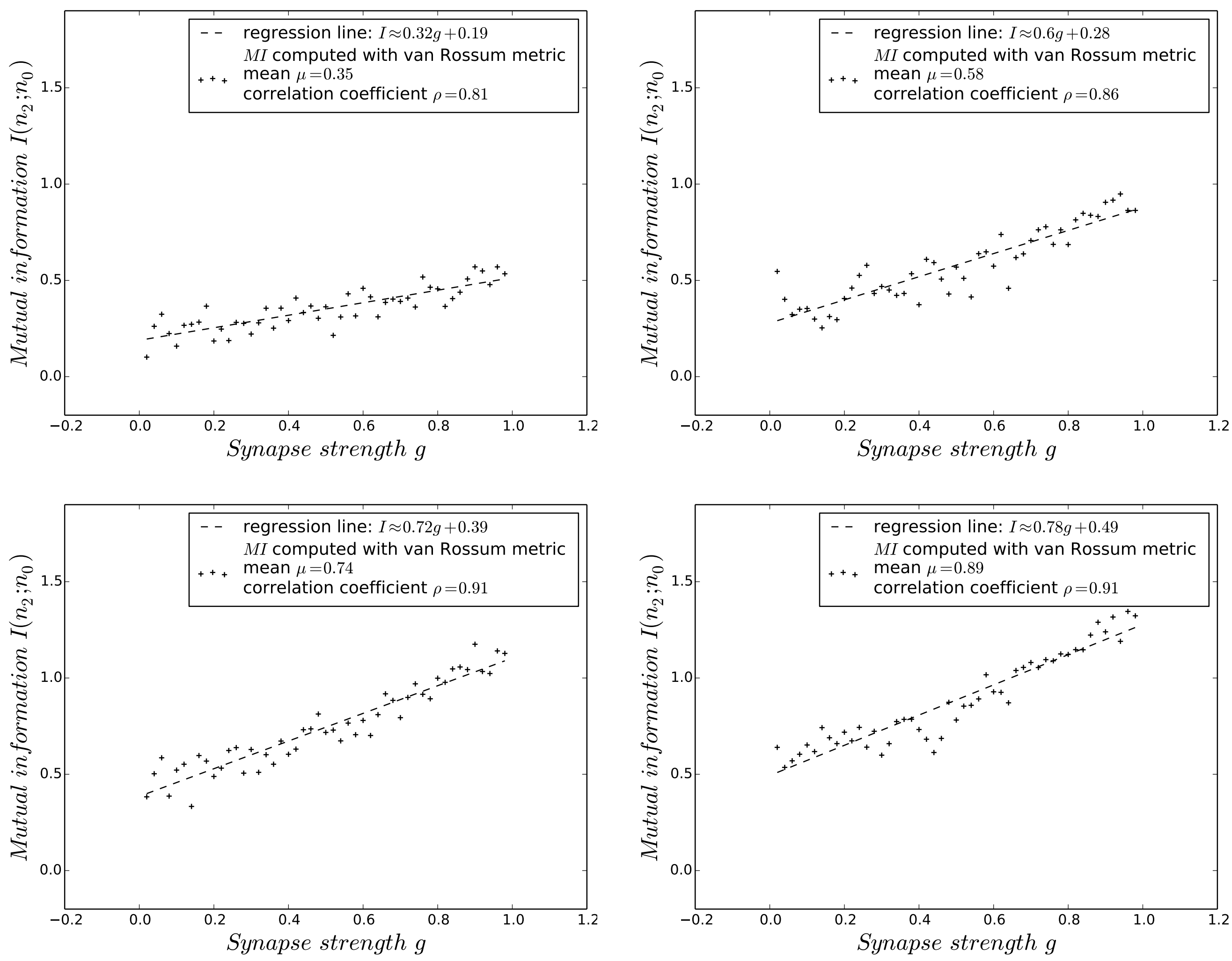


Figure 28: The results from running experiment 1 with the van Rossum spike-train metric and sample sizes $T_{\#} = 32$ (top-left), $T_{\#} = 64$ (top-right), $T_{\#} = 96$ (bottom-left) and $T_{\#} = 128$ (bottom-right). Region sizes are kept at $h_1 = h_2 = 14$ throughout.

From these results it can be seen that the information estimates vary as the sample data sets grow. Initially they increase faster as their relationship to connection strength takes its correct shape, but the results obtained with sample sets of size 64 (bottom-left) are already very similar to the ones produced with double the size - 128 (bottom right). The correlation coefficient in these two cases is almost exactly the same and the slopes of the regression lines are very similar. The positive offset along the MI axis appears to grow incrementally by about 0.1 with every next 32 samples added to the sets. The information is measured in bits reflecting the uncertainty associated with the structure of the spike trains expressed in binary coding length. Larger sample sets appear to enable the estimator to capture more of the mutual dependency between the structures of the two signals. This indicates that using the metric-space method, it would take more data to obtain optimal estimates, than to produce estimates that only capture the relative differences between the mutual information of weakly and strongly related variables. In order to confirm that the estimates will not keep increasing incrementally but will tend to converge to some optimal values provided enough data we need to examine the relationship between estimates for a fixed strength of the connection between two specific neurons and the size of the sample sets.

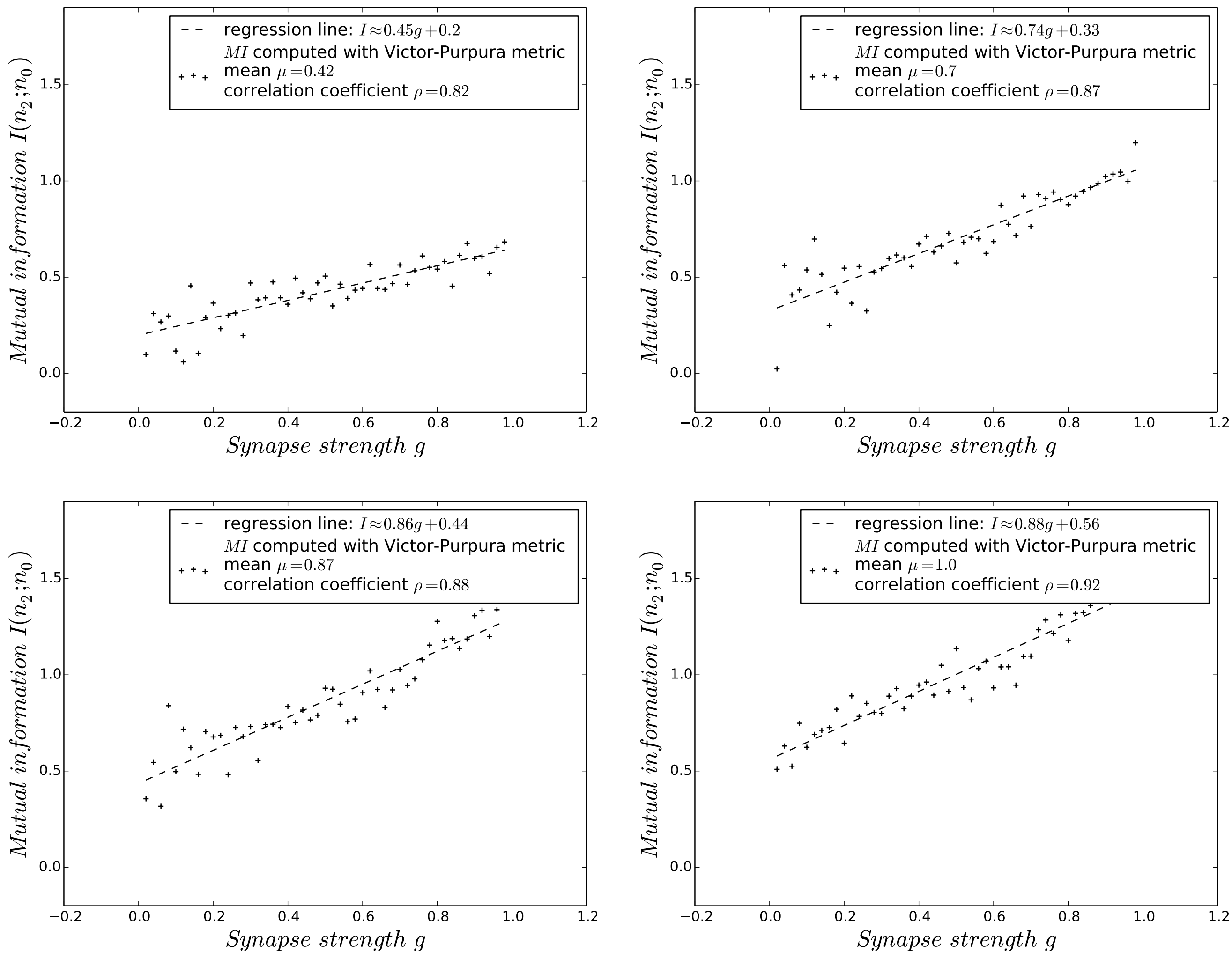


Figure 29: The results from running experiment 1 with the Victor-Purpura spike-train metric and sample sizes $T_{\#} = 32$ **(top-left)**, $T_{\#} = 64$ **(top-right)**, $T_{\#} = 96$ **(bottom-left)** and $T_{\#} = 128$ **(bottom-right)**. Region sizes are kept at $h_1 = h_2 = 14$ throughout.

We first plot the Pearson correlation coefficient against the sample size. As expected from the observed results this quantity converges very quickly to a value around 0.9, which is close to the maximum of 1. This is shown in figure 30 for the experimental cases of both spike train metrics - the van Rossum kernel-based method on the left, and the Victor-Purpura edit-length distance on the right. The two results are almost identical and reveal the same tendency. The results obtained with only 16 samples, 14 out of which are used for probability estimation are simply not representative.

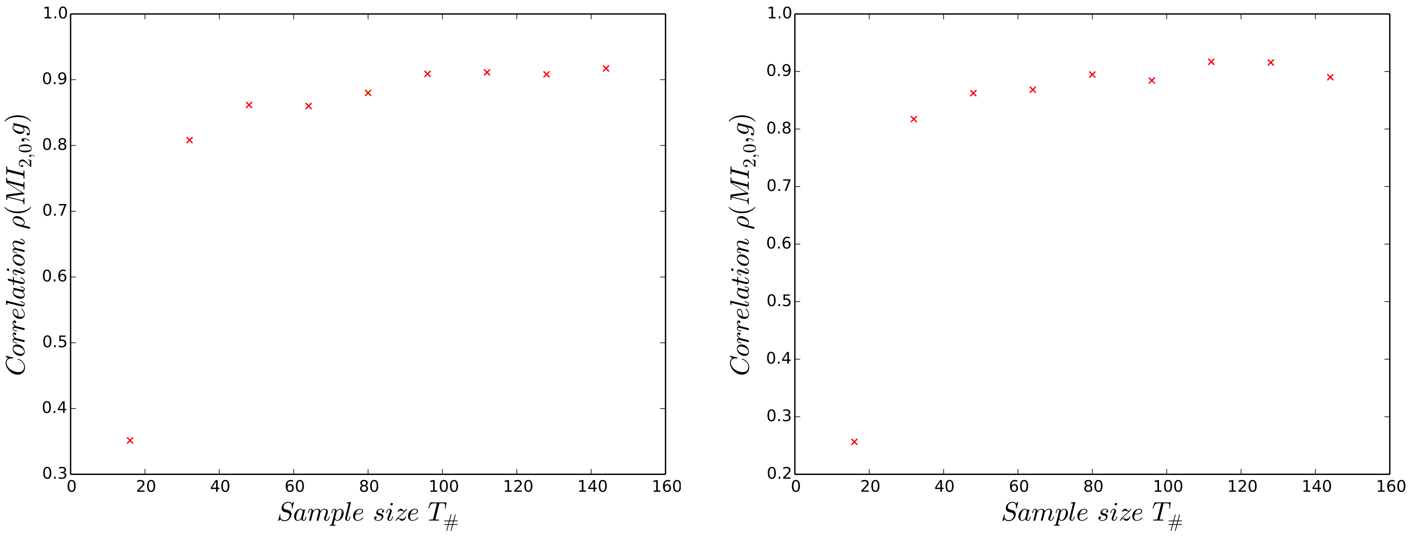


Figure 30: The convergence of the correlation coefficient ρ as the number of sample spike-trains used to estimate the MI between two neurons increases - with the van Rossum metric **(left)** and Victor-Purpura **(right)**.

We finally record the estimated mutual information of the two neurons when the connection between them is of maximum strength - $g = 1$, at every sample-size step. The resulting plots for the cases of both metrics can be seen in figure 31. In the fist one, where the kernel-based metric is used, the the slope of the estimates' growth appears to slowly decrease as they curve to the right the, larger $T_{\#}$ becomes. In the second one, with the edit-length distance this tendency is visible even more clearly as the amplitudes grow more rapidly and regularly through the first four estimates while for the remaining five they slow down and become more uncertain in their trend. From this final test we conclude that as the data processed by the estimator increases its estimates should vary less, approaching their optimal values. The convergence can be analysed further but the important news is that it is present.

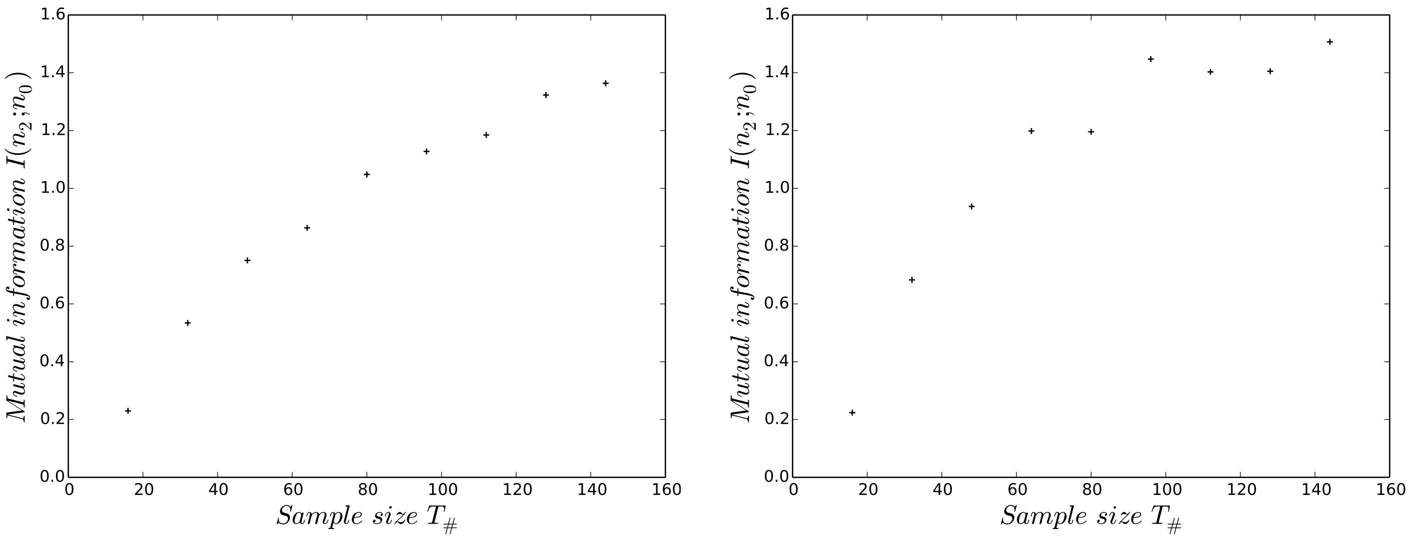


Figure 31: MI estimated between neurons with connection strength $g = 1$ as sample sizes grow. The growth of estimates slows down. The estimates obtained with the van Rossum distance **(left)** appear to curve to the right. The convergence is clearer in the case of the Victor-Purpura metric **(right)**.

4.2 Experiment 2

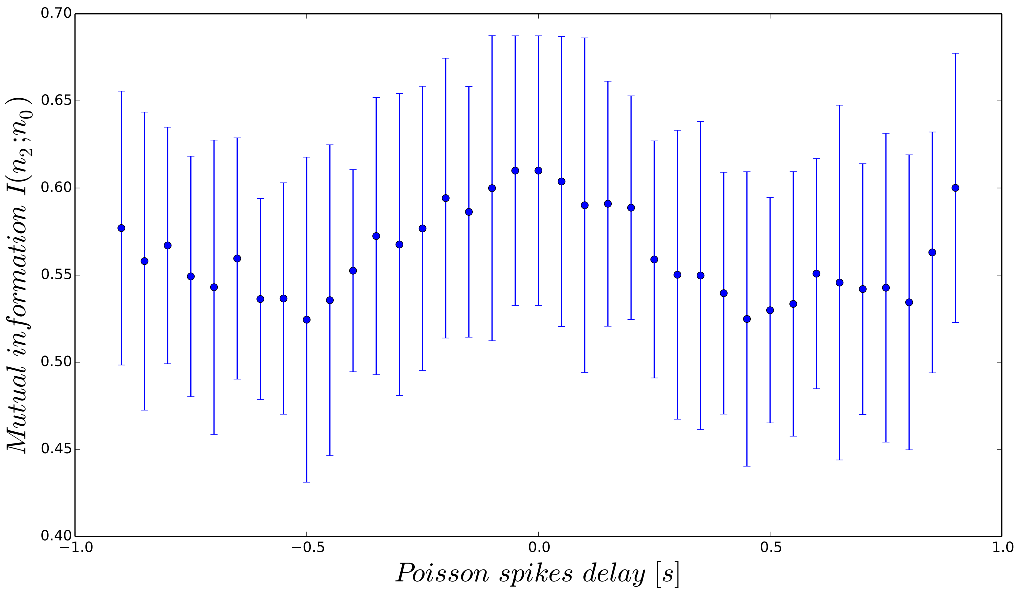


Figure 32: The results from experiment 2 obtained with the van Rossum spike-train metric for time delays in the range ±0.9 at step 0.05. Although the results meet our expectations in the range ±0.5 they interestingly exhibit the inverse behaviour for the rest of the range.

We now move on to the second experiment reflecting the mutual information estimator's sensitivity to time delays in one of the spike-train variables. The procedure for the experiment given in figure 10 in Chapter 2.4 is followed. The number of estimation rounds $R_\#$ is set to 23 with $T_\# = 48$ network simulation trials producing the spike-train sample sets for each of them. All the data is precomputed before time delays ranging in $[-0.9, +0.9]$ at step 0.05 $[s]$ are added to all the spike trains of Poisson neurons. The pairwise mutual information is estimated $R_\#$ times at each delay point and the mean and standard deviation are calculated for it. The plot for the MI between neurons 2 and 0 is displayed in figure 32 and 33 for distances computed with each of the two metrics.

For this experiment we consider only neurons 2 and 0 and the connection strength between them is set to 0.9. The time delay introduced in the spike trains of the stimuli has a principally different effect on their relationship to the responses, compared to the strength of the synapse between them. If we simply treat spike trains as signals, the delayed signals of the stimuli are still mutually dependent to the responses. The causality is just stretched through time and although it goes in both directions, the mutual information between two variables is a symmetric measure. However, because we cut off anything outside the 1s time window covered by the original spike train being delayed, the dependence should reduce as we leave some of the structure of the signal out. The spike train metrics on the other hand, consider the spike trains as a whole. In the case of the Victor-Purpura edit-length distance there are some replacements and moving stuff around to mould one structure into the other, whereas the van Rossum distance depends on the areas under the curves of the kernalised spike trains. But in fact, it does not really matter how they reflect the temporal shifts since they are computed to find neighbourhoods in the metric spaces of the stimuli and the responses separately. And since the same delay is added to all stimuli at each estimation step, similar responses to similar stimuli would still be grouped together in their respective sets. Therefore it is interesting to find out how if the estimates will reflect this kind of alteration in the signals and how.

Interestingly with the Victor-Purpura metric the estimator does not seem to capture the time shifts in the stimuli according to our expectations, especially when they are in the positive direction (fig. 33). This could possibly mean that structure-wise earlier parts of the stimulus spike train contain more information about the response than later parts. On the negative side the MI increases as the delay approaches zero, meaning that the residues of the stimuli shifted to the left, lacking some of the initial parts of the spike train are less similar to each other according to the edit-length distance, than residues lacking some of the later parts, shifted to the right. Possibly larger sample-set sizes for each estimation being more representative would improve the results. The van Rossum L_2 distance between kernel-filtered spike trains however, produces a different result at the same sampling resolution (fig. 32). In the range $[-0.5, +0.5]$ the MI estimates vary in the expected way reaching its peak around zero. When the delay is

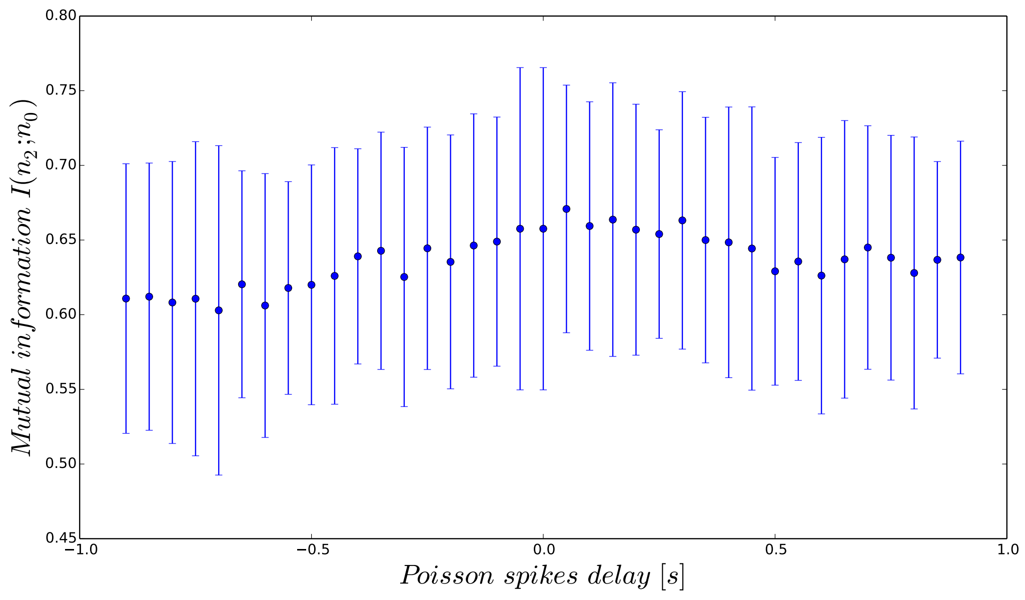


Figure 33: Experiment 2 with the Victor-Purpura edit-length distance. The estimates appear to be asymmetrical, giving higher estimates for positive delays than expected.

larger than ±0.5 the estimates become more noisy and appear to invert their trends reaching quite high values for delays around ±1. The nearest neighbour grouping under this metric appears to be better at capturing smaller delays. In this case again the relatively small sample-region size ratio may have its negative effect, but we will not investigate the effect of sample-set sizes on the experimental results here. We are satisfied with demonstrating that the estimator performs well on the task in experiment 1 and additionally appears to capture time the dynamics of time delays in the spike-train signals to some extent.

4.3 Connectivity inference

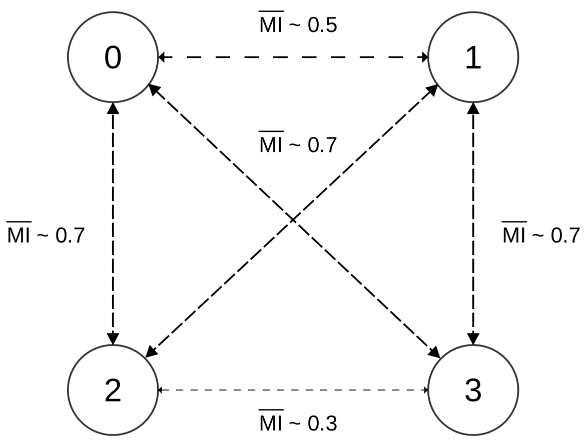

0
1
2
3
$\overline{\text{MI}}$ ~ 0.5
$\overline{\text{MI}}$ ~ 0.7
$\overline{\text{MI}}$ ~ 0.7
$\overline{\text{MI}}$ ~ 0.7
$\overline{\text{MI}}$ ~ 0.3

Figure 34: A graph of the hypothetical synaptic connections between the four neurons sampled in experiment 1. The edges are labelled with the means of the mutual information estimates for each pair.

The results from experiment 1 definitively establish a linear relationship between the strength of the synapse connecting two neurons in a network and the mutual information between their signals. The presence of this relationship validates the estimation model and suggests that we should be able to reconstruct some of the connectivity between the sampled neurons. Figure 35 gives all the pairwise results from experiment 1, omitting the doubling results for the relationships over synapses $\textcircled{3} \rightarrow \textcircled{1}$ and $\textcircled{3} \rightarrow \textcircled{0}$ which are analogous to the ones over $\textcircled{2} \rightarrow \textcircled{0}$ and $\textcircled{2} \rightarrow \textcircled{1}$ respectively. In a real experimental situation we are likely to have results from a network wired by synapses of specific strengths. Use our "insight" about how the MI estimates between different pairs of neurons vary together we can find some of the possible scenarios when the connections could be inferred correctly. We will only consider the subset of cases that were already simulated - that is when the inverse-proportional relationship between the strengths of the two pairs of synapses - represented by the vertical and diagonal edges in the graph respectively.

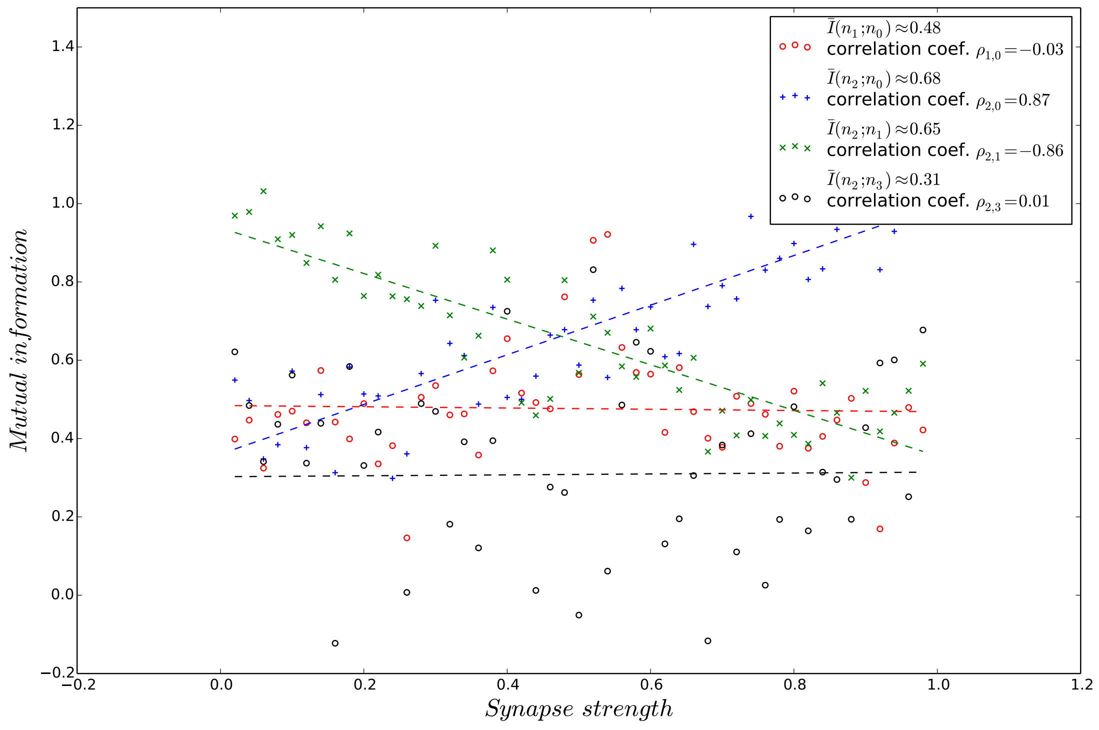


Figure 35: The combined results from experiment 1.

The the edges of the graph in figure 34 represent hypothetical synaptic connections between each pair of neurons and are labelled with the average MI estimates recorded between them in in experiment 1. These means are highest for the pairs of neurons that are truly connected. However, as can be seen from the plot in figure 35 there is no structure in the MI estimates between pairs of neurons of the same kind that are in fact not connected. Regardless of the synaptic strength between connected neurons these estimates appear to be somewhat uniformly distributed. Due to their stochastic nature in many cases they may not be very helpful if we are trying to infer the connectivity of the network using just the data processing inequality as discussed. Therefore we will exclude these cases and make the assumption that the results we obtained for these estimates are close to the average and are lower than the ones for pairs of neurons that are actually connected. That is we will consider the sub-cases when the MI estimates between pairs of connected neurons vary roughly between 0.5 and 0.9, keeping in mind their relationship, and the estimates for $I(n_1; n_0)$ and $I(n_2; n_3)$ gravitate around 0.5 and 0.3 respectively.

It turns out that in these cases, which are not quite unlikely within our simulated subset, the ARACNE strategy, described in [24, 26], is successful at identifying the true connections correctly. If the minimum-MI edge is removed from each possible triplet we are left only with the vertical and diagonal ones in the graph. The same results can be obtained by considering each triplet of neurons, thought of as spike-train variables, as a potential Markov chain and applying the DPI. Inferring the directions of the connections however, is a more involved task requiring some insight about the temporal structure of the signals apart from the mutual information between them. The important take-away point is that the estimates of the metric-space model for mutual information investigated here are valid and could be used by MI- based network inference algorithms reliably.

Conclusion

The aim of this project was to implement and test a model for estimating mutual information in metric spaces without coordinates in the context of computational neuroscience. The process of achieving this involved extensive research on the mathematical and information-theoretic grounds for its emergence, the metric-space framework for computing spike-train distances it relies on, and the computational methods for simulating neural signalling behaviour.

A neural simulation environment was developed in software to serve as the basis for constructing experiments designed to test the estimator. Two types of cutting-edge spike-train metrics have been implemented. It was demonstrated that probability densities can be successfully estimated using nearest-neighbour statistics in the metric spaces induced by these similarity measures. The implemented information estimator is simple and elegant and provides a novel direct way of estimating information-theoretic quantities depending on the probability distributions of more than one variables taking values in metric spaces.

The neuroscientific problem of calculating mutual information has no analytical solution to test the estimates against. The alternative estimation technique is effectively cumbersome and operates on a different principle risking to fit the space of spike-trains into the Procrustean bed of direct time discretisation. Therefore a strategy was devised for the verification of mutual information estimates based on criteria reflecting their implicit relationship to simulation parameters. Additionally, the nearest-neighbour resolution was investigated and the convergence of the estimates was tested on increasing amounts of input data under constant experimental conditions. Traditional correlation analysis has been used to quantify linear relationships along with the least-squares linear regression model. Both the Pearson correlation coefficient and the mutual information estimates were demonstrated to exhibit convergence behaviour.

The implications of the presented results to information-theoretic analysis in spaces with non-manifold geometry are yet to unfold. Here they served as a practical proof for the concept of using the marginal probability distribution of one variable as an integration measure in the metric space of another one to some approximation. We have seen this idea develop progressively through the work of Kozachenko and Leonenko [5], Kraskov, Stoegbauer and Grassberger [17], and Houghton and Tobin [27, 23].

The software and results produced during this year-long project will be used for its further development during the summer period. A version of the estimator for the case when one of the variables is coming from a discrete corpus has already been implemented and applied to real experimental data recording neural signals of a laboratory mouse running in a maze. Other, initially intended extensions have not been developed entirely to this point and are yet to be completed. The problem and its solution open up an array of possibilities for research and applications. And while this concludes the current work it will be interesting to continue exploring the topic.

Acknowledgements

I thank Conor Houghton for his continuous support throughout my studies at the University of Bristol. It has been a great privilege, during this final year, to take part in his research of cognition through the tools of mathematics and computer science. Understanding the process of neural communication is key to our exploration of the algorithms that nervous systems use to sustain life. It is in this kind of scientific endeavours that I see the true role of informatics in our endless hunt for purpose.

I thank Karl Henrik for the elaborations on various topics and for suggesting the fractalisation coefficient to me, Tilo Burghardt and Peter Flach for their teaching and guidance on some of the most intriguing aspects of computing, Dima, Bogdan and Ben for teaching me algorithms, Mike Fraser for introducing me into the world of programming, and everyone else in the department for being at the forefront of science and education.

I am forever grateful to my friends and colleagues Andrei Ilisei, Vince Polymenov, Ana Dumitras, Patrick Johnston, Milan Zolota and Nikolay Nikolov for their help and collaboration throughout my degree.

I thank my younger brother Alex, whom I owe a great deal of my competence as a computer scientist and in general.

Appendix

Experiment 1 - Implementation

```
from __future__ import division

from neuro import *

import numpy as np
import scipy as scp
import matplotlib as mp
import matplotlib.pyplot as plt

plt.switch_backend('QT4Agg')

# EXPERIMENTAL NETWORK CONNECTIVITY
#
# Integrate-and-fre neurons {0,1}
#
#       0     1
#       ^     ^
#       |\1-g/|
#       | \ / |
#     g |  x  | g        Synapse strength g in [0,1]
#       | / \ |
#       |/   \|
#       2     3
#
# Poisson neurons {2,3}

# Global parameters -------------------------------------------------------------
mili  = 0.001           # Scaling factor 10^-3

E_l   = -70*mili        # Standard reverse potential
V_th  = -54*mili        # Vth = -40 [mV]
V_res = -80*mili        # Reset membrane potential
R_m   = 1               # Rm = 1[M_ohm]
I_e   = 18*mili         # Ie = 3.1[nA]
t_M   = 30*mili         # tau_m = 10[ms] = C_m*Rm time constant of the membrane
t_Ref = 5*mili          # Refractory period = 5[ms]

dt    = 1*mili          # Time scale [ms]
T     = 1               # total simulation period [s]

t_S   = 10*mili         # Time scale of the synapse

IF = 2                  # Number of Integrate and Fire neurons
P  = 2                  # Number of Poisson neurons
N  = IF + P             # Total number of neurons

minRate = 10            # maximum average spike-rate for poisson neurons
maxRate = 50            # maximum average spike-rate for poisson neurons
rateRange = maxRate - minRate

samples = []            # object storing experimental sample set
data = []
trials  = 48            # number of simulation rounds in experiment

g_res = 50              # no. of synapse strength steps over [0,1] ~> resolution
g_bin = 1.0/g_res       # width of synapse strength step based on g_res
g_E = 0.5
g_sqrd_E = 0


h1 = 12                 # size of open balls in spike-trrain metric space
h2 = 12

tau_vR = 12*mili
vR_metric = (metric('vR', tau = tau_vR), 'van Rossum')
VP_metric = (metric('VP', q = 166), 'Victor-Purpura')

#metric = vR_metric
metric = vR_metric
```

```
MI_1_0 = []
MI_1_0_E = 0
MI_1_0_sqrd_E = 0
MI_1_0xG_E = 0 # mean of MI_1_0 x g

MI_2_0 = []
MI_2_0_E = 0
MI_2_0_sqrd_E = 0
MI_2_0xG_E = 0 # mean of MI_2_0 x g

MI_3_0 = []
MI_3_0_E = 0
MI_3_0_sqrd_E = 0
MI_3_0xG_E = 0 # mean of MI_3_0 x g

MI_2_3 = []
MI_2_3_E = 0
MI_2_3_sqrd_E = 0
MI_2_3xG_E = 0 # mean of MI_2_3 x g


# Experiment Simulation--------------------------------------------------------

# Convey experiment over a variation of synapse strengths
for g in range(1, g_res) :
  g *= g_bin
  sample = experiment()
  cMat = np.array([[  0,   0,  0,  0],
                   [  0,   0,  0,  0],
                   [  g, 1-g,  0,  0],
                   [1-g,   g,  0,  0]])

# simulate network a number of trials to generate sample data
  for r in range(trials) :
#   Initialise network parameters
    Vs = V_res + np.random.rand(IF)*(V_th - V_res)
    sTs = -t_M + np.random.rand(N)*t_M
    sRates = minRate + np.random.rand(P)*(rateRange)

    neurons = []
    for i in range(IF) :
      neurons.append(neuron(i, Vs[i], sTs[i], E_l, V_th, V_res, R_m, I_e, t_M, t_Ref))
    for i in range(IF, N) :
      neurons.append(pNeuron(i, sRates[i-IF], T, dt, sTs[i]))

#   Simulate network and save data
    sample.simulation += [netSim(neurons, cMat, T, dt)]
    Vs, Gs, raster = sample.simulation[-1].simulate()

    sample.population += [neurons]

# add experiment results to samples set
  samples += [sample]

# assemble together spike trains elicited from each neuron over trials
  neuro_var = []
  for n in range(N):
    s_trains = []
    for r in range(trials):
      s_trains += [samples[-1].population[r][n].sTrain]

    neuro_var += [s_trains]

  MI_1_0_g = computeMI(neuro_var[1], neuro_var[0], metric[0], h1, h2)
  MI_1_0.append(MI_1_0_g)
  MI_1_0_E      += MI_1_0_g
  MI_1_0_sqrd_E += MI_1_0_g**2
  MI_1_0xG_E    += MI_1_0_g*g

  MI_2_0_g = computeMI(neuro_var[2], neuro_var[0], metric[0], h1, h2)
  MI_2_0.append(MI_2_0_g)
  MI_2_0_E      += MI_2_0_g
  MI_2_0_sqrd_E += MI_2_0_g**2
  MI_2_0xG_E    += MI_2_0_g*g

  MI_3_0_g = computeMI(neuro_var[3], neuro_var[0], metric[0], h1, h2)
  MI_3_0.append(MI_3_0_g)
  MI_3_0_E      += MI_3_0_g
  MI_3_0_sqrd_E += MI_3_0_g**2
  MI_3_0xG_E    += MI_3_0_g*g

  MI_2_3_g = computeMI(neuro_var[2], neuro_var[3], metric[0], h1, h2)
  MI_2_3.append(MI_2_3_g)
  MI_2_3_E      += MI_2_3_g
  MI_2_3_sqrd_E += MI_2_3_g**2
  MI_2_3xG_E    += MI_2_3_g*g

  g_sqrd_E += g**2
```

```
MI_1_0_E      /= (g_res-1)
MI_1_0_sqrd_E /= (g_res-1)
MI_1_0xG_E    /= (g_res-1)

MI_2_0_E      /= (g_res-1)
MI_2_0_sqrd_E /= (g_res-1)
MI_2_0xG_E    /= (g_res-1)

MI_3_0_E      /= (g_res-1)
MI_3_0_sqrd_E /= (g_res-1)
MI_3_0xG_E    /= (g_res-1)

MI_2_3_E      /= (g_res-1)
MI_2_3_sqrd_E /= (g_res-1)
MI_2_3xG_E    /= (g_res-1)

g_sqrd_E /= (g_res-1)

pearson_1_0 = (MI_1_0xG_E - MI_1_0_E*g_E)/np.sqrt((MI_1_0_sqrd_E - MI_1_0_E**2)*(g_sqrd_E - g_E**2))
pearson_2_0 = (MI_2_0xG_E - MI_2_0_E*g_E)/np.sqrt((MI_2_0_sqrd_E - MI_2_0_E**2)*(g_sqrd_E - g_E**2))
pearson_3_0 = (MI_3_0xG_E - MI_3_0_E*g_E)/np.sqrt((MI_3_0_sqrd_E - MI_3_0_E**2)*(g_sqrd_E - g_E**2))
pearson_2_3 = (MI_2_3xG_E - MI_2_3_E*g_E)/np.sqrt((MI_2_3_sqrd_E - MI_2_3_E**2)*(g_sqrd_E - g_E**2))

print 'Mean MI(n1;n0)            = ' + str(MI_1_0_E)
print 'Pearson coef. MI(n1;n0) = ' + str(pearson_1_0)
print 'Mean MI(n2;n0)            = ' + str(MI_2_0_E)
print 'Pearson coef. MI(n2;n0) = ' + str(pearson_2_0)
print 'Average MI(n3;n0)        = ' + str(MI_3_0_E)
print 'Pearson coef. MI(n3;n0) = ' + str(pearson_3_0)
print 'Mean MI(n2;n3)            = ' + str(MI_2_3_E)
print 'Pearson coef. MI(n2;n3) = ' + str(pearson_2_3)


# Produce Graphs-----------------------------------------------------------------
# Plot Mutual Information

#set up x values (g - for conductance)
g_range  = [a*g_bin for a in range(1,g_res)]

# MI(Neuron_2; Neuron_0)
#perform linear regression on generated data
y = np.array(MI_2_0)
x = np.array(g_range)
A = np.vstack([x, np.ones(len(x))]).T
m20, c20 = np.linalg.lstsq(A, y)[0]

#scatter Mutual Information between neurons 2 & 0
#and plot fitted line
plt.figure(1, dpi=120)
plt.scatter(g_range, MI_2_0, marker = '+', color = 'black',
            label='$MI_{2,0}$ computed with '+metric[1]+' metric'+
            '\n$g_{2,0}=g,$ mean $\\mu='+str(round(MI_2_0_E,2))+
            '$\ncorrelation coefficient $\\rho='+
            str(round(pearson_2_0,2))+'$')
plt.plot(x, m20*x + c20, linestyle='--', color='black',
         label='regression line: $I\\approx'+str(round(m20,2))+
         'g+'+str(round(c20,2))+'$')
plt.ylabel('$Mutual$ $information$ $I(n_2;n_0)$', fontsize=20)
plt.xlabel('$g$', fontsize=20)
plt.ylim(-0.2,1.5)
plt.legend()
plt.show()


# MI(Neuron_2; Neuron_0)
#perform linear regression
y = np.array(MI_3_0)
m30, c30 = np.linalg.lstsq(A, y)[0]

plt.figure(2, dpi=120)
plt.scatter(g_range, MI_3_0, marker = '+', color = 'black',
            label='$MI_{3,0}$ computed with '+metric[1]+' metric'+
            '\n$g_{3,0}=1-g,$ mean $\\mu='+str(round(MI_3_0_E,2))+
            '$\ncorrelation coefficient $\\rho='
            +str(round(pearson_3_0,2))+'$')
plt.plot(x, m30*x + c30, linestyle='--', color='black',
         label='regression line: $I\\approx'+str(round(m30,2))+
         'g+'+str(round(c30,2))+'$')
plt.ylabel('$Mutual$ $information$ $I(n_3;n_0)$', fontsize=20)
plt.xlabel('$g$', fontsize=20)
plt.ylim(-0.2, 1.5)
plt.legend()
plt.show()
```

```

# MI(Neuron_3; Neuron_0) vs MI(Neuron_2; Neuron_0)
#perform linear regression
x = np.array(MI_2_0)
sortx = x.argsort()
x = np.array(x)[sortx]
A = np.vstack([x, np.ones(len(x))]).T
y = np.array(MI_3_0)
m3021, c3021 = np.linalg.lstsq(A, y)[0]

plt.figure(0, dpi=120)
plt.scatter(x, y, marker = '+', color = 'black',
            label='$MI_{3,0}$ vs $MI_{2,0}$, '+
            'both computed with '+metric[1]+' metric')
plt.plot(x, m3021*x + c3021, linestyle='--', color='black',
         label='regression line $MI_{3,0}\\approx'+str(round(m3021,2))+
         'MI_{2,0}+'+str(round(c3021,2))+'$')
plt.ylabel('$Mutual$ $information$ $I(n_3;n_0)$', fontsize=20)
plt.xlabel('$Mutual$ $information$ $I(n_2;n_0)$', fontsize=20)
plt.ylim(-0.2, 1.5)
plt.legend()
plt.show()


# MIXED PAIRWISE MI PLOT
#perform linear regressions
y = np.array(MI_1_0)
m10, c10 = np.linalg.lstsq(A, y)[0]
y = np.array(MI_2_3)
m23, c23 = np.linalg.lstsq(A, y)[0]

plt.figure(3, dpi=120)

plt.scatter(g_range, MI_1_0, facecolors='none', edgecolors='red',
            label='$\\bar{I}(n_1;n_0)\\approx'+str(round(MI_1_0_E,2))+
            '$\ncorrelation coef. $\\rho_{1,0}='+str(round(pearson_1_0,2))+'$')
plt.plot(x, m10*x + c10, linestyle='--', color='red')

plt.scatter(g_range, MI_2_0, marker = '+', color = 'blue',
            label='$\\bar{I}(n_2;n_0)\\approx'+str(round(MI_2_0_E,2))+
            ',$ $g_{2,0}=g$'+
            '\ncorrelation coef. $\\rho_{2,0}='+str(round(pearson_2_0,2))+'$')
plt.plot(x, m20*x + c20, linestyle='--', color='blue')

plt.scatter(g_range, MI_3_0, marker = 'x', color = 'green',
            label='$\\bar{I}(n_3;n_0)\\approx'+str(round(MI_3_0_E,2))+
            ',$ $g_{3,0}=1-g$'+
            '\ncorrelation coef. $\\rho_{3,0}='+str(round(pearson_3_0,2))+'$')
plt.plot(x, m30*x + c30, linestyle='--', color='green')

plt.scatter(g_range, MI_2_3, facecolors='none', edgecolors='black',
            label='$\\bar{I}(n_2;n_3)\\approx'+str(round(MI_2_3_E,2))+
            '$\ncorrelation coef. $\\rho_{2,3}='+str(round(pearson_2_3,2))+'$')
plt.plot(x, m23*x + c23, linestyle='--', color='black')

plt.ylabel('$Mutual$ $information$', fontsize=20)
plt.xlabel('$g$', fontsize=20)
plt.ylim(-0.2, 1.5)
plt.legend()
plt.show()
```

Figure 36: **neuroSim1.py**

Experiment 2 - Implementation

```
from __future__ import division

from neuro import *

from copy import deepcopy

import numpy as np
import scipy as scp
import matplotlib as mp
import matplotlib.pyplot as plt

plt.switch_backend('QT4Agg')

# EXPERIMENTAL NETWORK CONNECTIVITY
#
# Integrate-and-fre neurons {0,1}
#
#       0     1
#       ^     ^
#       |\1-g/|
#       | \ / |
#     g |  x  | g         Synapse strength g in [0,1]
#       | / \ |
#       |/   \|
#       2     3
#
# Poisson neurons {2,3}

# Global parameters -----------------------------------------------------------
mili  = 0.001           # Scaling factor 10^-3

E_l   = -70*mili        # Standard reverse potential
V_th  = -54*mili        # Vth = -40 [mV]
V_res = -80*mili        # Reset membrane potential
R_m   = 1               # Rm = 1[M_ohm]
I_e   = 18*mili         # Ie = 3.1[nA]
t_M   = 30*mili         # tau_m = 10[ms] = C_m*Rm time constant of the membrane
t_Ref = 5*mili          # Refractory period = 5[ms]

dt    = 1*mili          # Time scale [ms]
T     = 1               # total simulation period [s]

t_S   = 10*mili         # Time scale of the synapse

IF = 2                  # Number of Integrate and Fire neurons
P  = 2                  # Number of Poisson neurons
N  = IF + P             # Total number of neurons

minRate = 10            # maximum average spike-rate for poisson neurons
maxRate = 20            # maximum average spike-rate for poisson neurons
rateRange = maxRate - minRate

trials  = 48            # number of simulation rounds in each experiment
rounds  = 23

d_luft = 18
d_bin  = 0.05           # 10**(-len(str(d_luft)))

h1 = 12                 # size of open balls in spike-trrain metric space
h2 = 12

tau_vR = 12*mili
vR_metric = metric('vR', tau=tau_vR)
VP_metric = metric('VP', q=166)

metric = VP_metric

MI_2_0   = []

g = 0.9                # synapse 2-0 and 3-1 strength == 1 minus 2-1 or 3-0
# set up network topology

samples = []
cMat = np.array([[  0,   0,  0,  0],
                 [  0,   0,  0,  0],
                 [  g, 1-g,  0,  0],
                 [1-g,   g,  0,  0]])
```

```
# Experiment Simulation----------------------------------------------------------

for r in range(rounds):
  sample = experiment()
# simulate network a number of trials to generate sample data
  for t in range(trials):
#   initialise network parameters
    Vs      = V_res + np.random.rand(IF)*(V_th - V_res)
    sTs     = -t_M + np.random.rand(N)*t_M
    sRates  = minRate + np.random.rand(P)*(rateRange)
    neurons = []
    for i in range(IF):
      neurons.append(neuron(i, Vs[i], sTs[i], E_l, V_th, V_res, R_m, I_e, t_M, t_Ref))
    for i in range(IF, N):
      pN = pNeuron(i, sRates[i-IF], T, dt, sTs[i])
      pN.delay(0.05)
      neurons.append(pN)

#   simulate network and save data
    sample.simulation.append(netSim(neurons, cMat, T, dt))
    Vs, Gs, raster = sample.simulation[-1].simulate()
    sample.population.append(neurons)

  samples.append(sample)

#slide trhough the predefined range of delays
for d in range(-d_luft, d_luft+1) :
  d *= d_bin
# estimate information at each round and each delay
  MI_d = []
  for s in range(rounds):
  # group together spike trains from each IF neuron over trials
    neuro_var = []
    for n in range(IF):
      s_trains = []
      for t in range(trials):
        s_trains.append(samples[s].population[t][n].sTrain)
      neuro_var.append(s_trains)

  # delay and group together spike trains from each Poisson neuron over trials
    for n in range(IF, N):
      s_trains = []
      for t in range(trials):
        pN = deepcopy(samples[s].population[t][n])
        pN.delay(d)
        s_trains.append(pN.sTrain)
      neuro_var.append(s_trains)

#   compute mutual information between neurons 2 & 0 at current round
    MI_d.append(computeMI(neuro_var[2], neuro_var[0], metric, h1, h2))
# add list of mutual informations computted at current delay to array
  MI_2_0.append(MI_d)

#calculate mean MI and std.dev from samples at each delay point
MI_2_0_avg = []
MI_2_0_std = []
for d in range(len(MI_2_0)):
  MI_d_avg = 0
  for m in range(rounds):
    MI_d_avg += MI_2_0[d][m]
  MI_d_avg /= rounds
  MI_d_std = 0
  for m in range(rounds):
    MI_d_std += (MI_2_0[d][m] - MI_d_avg)**2
  MI_d_std = np.sqrt(MI_d_std/rounds)

  MI_2_0_avg.append(MI_d_avg)
  MI_2_0_std.append(MI_d_std)

# Produce Graphs----------------------------------------------------------------

#plot MI
d_lays = [d*d_bin for d in range(-d_luft,d_luft+1)]

plt.figure(2,dpi=120)
#plt.scatter(d_lays, MI_2_0_avg)
plt.errorbar(d_lays, MI_2_0_avg, yerr=MI_2_0_std, fmt='o')
plt.ylabel('$Mutual$ $information$ $I(n_2;n_0)$', fontsize=23)
plt.xlabel('$Poisson$ $spikes$ $delay$ $[s]$', fontsize=23)
plt.show()
```

Figure 37: **neuroSim2.py**

References

[1] Lapicque L. (1907). Recherches quantitatives sur lexcitation lectrique des nerfs traite comme une polarisation. *J. Physiol. Pathol. Gen*, 9:620635.

[2] Shannon CE. (1948) A mathematical theory of communication. *Bell Syst. Tech. J.*, 27: 379-423,623-656.

[3] Hodgkin AL and Huxley HF. (1952) Propagation of electrical signals along giant nerve fibres. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 140:177-183.

[4] Dobrushin RL. (1958) A simplified method for experimental estimate of the entropy of a stationary sequence *Theory Prob. Appl. 3*, 462

[5] Kozachenko LF, Leonenko NN. (1987) Sample estimate of the entropy of a random vector. *Probl. Pereda. Inf.*, 23: 916.

[6] Press WH, Teukolsky SA, Vetterling WT and Flannery BP. (1988) "Numerical recipes in C". *Cambridge University Press.*, p.710-714.

[7] Victor JD and Purpura KP. (1996) Metric-space analysis of spike trains: theory, algorithms and application. *Network: Computation in Neural Systems*, 8:127-164.

[8] Victor JD and Purpura KP. (1996) Nature and precision of temporal coding in visual cortex: a metric-space analysis. *Journal of Neurophysiology*, 76:1310-1326

[9] Reich DS, Victor JD, Knight BW, Ozaki T, Kaplan E. (1997) Response variability and timing precision of neuronal spike trains in vivo. *Journal Neurophysiology*, 77:2836-2841.

[10] Samoilov M. (1997) Reconstruction and functional analysis of general chemical reactions and reaction networks. *Ph.D. thesis*, Stanford University.

[11] Strong SP, Koberle R, de Ruyter van Steveninck RR, Bialek W. (1998) Entropy and information in neural spike trains *Phys. Rev. Lett. 80*, 197-200

[12] Rubner Y, Tomasi C, Guibas LJ. (2000) The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40:99121.

[13] Dayan P and Abbott L. (2001) "Theoretical Neuroscience" *MIT Press*, Chapter 1.4.

[14] Samoilov M, Arkin A, Ross J. (2001) On the deduction of chemical reaction pathways from measurements of time series of concentrations. *Chaos*, 11: 108114.

[15] van Rossum M. (2001) A novel spike distance. *Neural Computation*, 12:751-763.

[16] Victor JD. (2002) Binless strategies for estimation of information from neural data. *Phys. Rev. E*, 66, 051903.

[17] Kraskov A, Stögbauer H, Grassberger P. (2004) Estimating mutual information. *Phys. Rev. E*, 69, 066138.

[18] Cover TM, Thomas JA. (2006) "Elements of information theory" -2nd ed. *A Wiley-Interscience publication.*, p.13-57 ISBN-13 978-0-471-24195-9.

[19] Margolin A, Wang K, Lim W, Kustagi M, Nemenman I, et al. (2006) Reverse engineering cellular networks. *Nat Protoc*, 1: 662671.

[20] Margolin A, Nemenman I, Basso K, Wiggins C, Stolovitzky G, et al. (2006) Aracne: an algorithm for the reconstruction of gene regulatory networks in a mammalian cellular context. *BMC Bioinform*, 7: S7.

[21] Houghton CJ, Victor JD. (2012) Measuring representational distances – the spike-train metrics approach. *"Visual population codes: toward a common multivariate framework for cell recording and functional imaging". (eds N Kriegeskorte, G Kreiman) Cambridge, MA: MIT Press*, p.391-416.

[22] Houghton CJ and Kreuz T. (2012) On the efficient calculation of van Rossum distances. *Network computation in neural systems*, 23(1-2):48-58.

[23] Tobin RJ, Houghton CJ. (2013) A kernel-based calculation of spike train information. *Entropy 15*, 45404552.

[24] Villaverde AF, Ross J, Banga JR. (2013) Reverse engineering cellular networks with information theoretic methods. *Cells 2013*, 2, 306-329; doi:10.3390/cells2020306 ISSN 2073-4409.

[25] Kinney JB and Atwal GS. (2014) Equitability, mutual information, and the maximal information coefficient. *Proc. Natl. Acad. Sci. U S A*, 111: 33549.

[26] Villaverde AF, Ross J, Mora n F, Banga JR. (2014) MIDER: Network Inference with Mutual Information Distance and Entropy Reduction. PLoS ONE 9(5): e96732. doi:10.1371/journal.pone.0096732

[27] Houghton CJ. (2015) Calculating mutual information for spike trains and other data with distances but no coordinates. *R. Soc. open sci.*, 2: 140391.