JIDT: An information-theoretic toolkit for studying the dynamics of complex systems

Joseph T. Lizier[1, 2, *]

[1]*Max Planck Institute for Mathematics in the Sciences, Inselstraße 22, 04103 Leipzig, Germany*
[2]*CSIRO Digital Productivity and Services, Marsfield, NSW 2122, Australia*

(Dated: August 14, 2014)

Complex systems are increasingly being viewed as distributed information processing systems, particularly in the domains of computational neuroscience, bioinformatics and Artificial Life. This trend has resulted in a strong uptake in the use of (Shannon) information-theoretic measures to analyse the dynamics of complex systems in these fields. We introduce the Java Information Dynamics Toolkit (JIDT): a Google code project which provides a standalone, (GNU GPL v3 licensed) open-source code implementation for empirical estimation of information-theoretic measures from time-series data. While the toolkit provides classic information-theoretic measures (e.g. entropy, mutual information, conditional mutual information), it ultimately focusses on implementing higher-level measures for information dynamics. That is, JIDT focusses on quantifying information storage, transfer and modification, and the dynamics of these operations in space and time. For this purpose, it includes implementations of the transfer entropy and active information storage, their multivariate extensions and local variants. JIDT provides implementations for both discrete and continuous-valued data for each measure, including various types of estimator for continuous data (e.g. Gaussian, box-kernel and Kraskov-Stögbauer-Grassberger) which can be swapped at run-time due to Java's object-oriented polymorphism. Furthermore, while written in Java, the toolkit can be used directly in MATLAB, GNU Octave and Python. We present the principles behind the code design, and provide several examples to guide users.

PACS numbers: 89.70.Cf, 05.65.+b, 89.75.Fb, 87.19.lo
Keywords: information transfer, information storage, intrinsic computation, complex networks, information theory, transfer entropy, Java, MATLAB, Octave, Python

[*]joseph.lizier@gmail.com

I. INTRODUCTION

Information theory was originally introduced by Shannon [1] to quantify fundamental limits on signal processing operations and reliable communication of data [2, 3]. More recently, it is increasingly being utilised for the design and analysis of complex self-organized systems [4]. *Complex systems* science [5] is the study of large collections of (generally simple) entities, where the global system behaviour is a non-trivial result of the local interactions of the individual elements; e.g. emergent consciousness from neurons, emergent cell behaviour from gene regulatory networks and flocks determining their collective heading. The application of information theory to complex systems can be traced to the increasingly-popular perspective that commonalities between complex systems may be found "in the way they handle information" [6]. Certainly, there have been many interesting insights gained from the application of traditional information-theoretic measures such as entropy and mutual information to study complex systems, for example: proposals of candidate complexity measures [7, 8], characterising order-chaos phase transitions [9–12], and measures of network structure [13, 14].

More specifically though, researchers are increasingly viewing the global behaviour of complex systems as emerging from the *distributed information processing*, or *distributed computation*, between the individual elements of the system [15–19], e.g. collective information processing by neurons [20]. Computation in complex systems is examined in terms of: how information is transferred in the interaction between elements, how it is stored by elements, and how these information sources are non-trivially combined. We refer to the study of these operations of *information storage, transfer and modification*, and in particular how they unfold in space and time, as *information dynamics* [15, 21].

Information theory is the natural domain to quantify these operations of information processing, and we have seen a number of measures recently introduced for this purpose, including the well-known transfer entropy [22], as well as active information storage [23] and predictive information [24, 25]. Natural affinity aside, information theory offers several distinct advantages as a measure of information processing in dynamics,[1] including: its model-free nature (requiring only access to probability distributions of the dynamics), ability to handle stochastic dynamics and capture non-linear relationships, its abstract nature, generality, and mathematical soundness.

In particular, this type of information-theoretic analysis has gained a strong following in computational neuroscience, where the transfer entropy has been widely applied [27–39], (for example for effective network inference), and measures of information storage are gaining traction [40–42]. Similarly, such information-theoretic analysis is popular in studies of canonical complex systems [19, 23, 43–46], dynamics of complex networks [47–52], social media [53–55], and in Artificial Life and Modular Robotics both for analysis [56–64] and design [65–70] of embodied cognitive systems (in particular see the "Guided Self-Organization" series of workshops, e.g. [71]).

This paper introduces **JIDT** – the **Java Information Dynamics Toolkit** – which provides a standalone implementation of information-theoretic measures of dynamics of complex systems. JIDT is open-source, licensed under GNU General Public License v3, and available for download via Google code at http://code.google.com/p/information-dynamics-toolkit/. JIDT is designed to facilitate *general purpose* empirical estimation of information-theoretic measures from time-series data, by providing easy to use, portable implementations of measures of information transfer, storage, shared information and entropy.

We begin by describing the various information-theoretic measures which are implemented in JIDT in Section II, including the basic entropy and (conditional) mutual information [2, 3], as well as the active information storage [23], the transfer entropy [22] and its conditional/multivariate forms [45, 46]. We also describe how one can compute *local* or *pointwise* values of these information-theoretic measures at specific observations of time-series processes, so as to construct their *dynamics* in time. We continue to then describe the various estimator types which are implemented for each of these measures in Section III (i.e. for discrete or binned data, and Gaussian, box-kernel and Kraskov-Stögbauer-Grassberger estimators). Readers familiar with these measures and their estimation may wish to skip these sections. We also summarise the capabilities of similar information-theoretic toolkits in Section III C (focussing on those implementing the transfer entropy).

We then turn our attention to providing a detailed introduction of JIDT in Section IV, focussing on the current version 1.0 distribution. We begin by highlighting the unique features of JIDT in comparison to related toolkits, in particular in: providing *local* information-theoretic measurements of dynamics; implementing conditional and other multivariate transfer entropy measures; and including implementations of other related measures including the active information storage. We describe the (almost zero) installation process for JIDT in Section IV A: JIDT is standalone software, requiring no prior installation of other software (except a Java Virtual Machine), and no explicit compiling or building. We describe the contents of the JIDT distribution in Section IV B, and then in Section IV C outline which estimators are implemented for each information theoretic measure. We then describe the principles behind the

[1] Further commentary on links between information-theoretic analysis and traditional dynamical systems approaches are discussed in [26].

design of the toolkit in Section IV D, including our object-oriented approach in defining interfaces for each measure, then providing multiple implementations (one for each estimator type). Section IV E-Section IV G then describe how the code has been tested, how the user can (re-)build it, and what extra documentation is available (principally the project wiki and Javadocs).

Finally and most importantly, Section V outlines several demonstrative examples supplied with the toolkit, which are intended to guide the user through how to use JIDT in their code. We begin with simple Java examples in Section V A, which includes a description of the general pattern of usage in instantiating a measure and making calculations, and walks the user through differences in calculators for discrete and continuous data, and multivariate calculations. We also describe how to take advantage of the polymorphism in JIDT's object-oriented design to facilitate run-time swapping of the estimator type for a given measure. Other demonstration sets from the distribution are presented also, including: basic examples using the toolkit in MATLAB, GNU Octave and Python (Section V B and Section V C); reproduction of Schreiber's original transfer entropy examples from [22] (Section V D); and local information profiles for cellular automata (Section V E).

II. INFORMATION-THEORETIC MEASURES

In this section, we give a brief overview of the information-theoretic measures which will are implemented in JIDT. We begin by describing basic information-theoretic measures such as entropy and mutual information in Section II A, then go on to describe in Section II B the more contemporary measures which are being used to quantify the information dynamics of distributed computation. The latter are the real focus of the toolkit. We also describe in Section II C how one can measure local or pointwise information-theoretic measures (to assign information values to specific observations or outcomes of variables and their interactions), the extension of the measures to continuous variables in Section II D, and in Section II E how one can evaluate the statistical signficance of the interaction between variables. All features discussed are available in JIDT unless otherwise noted.

A. Basic information-theoretic measures

We first outline basic information-theoretic measures [2, 3] implemented in JIDT.

The fundamental quantity of information theory is the **Shannon entropy**, which represents the average uncertainty associated with any measurement x of a random variable X:[2]

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

with a probabilities distribution function p defined over the alphabet α_x of possible outcomes for x (where $\alpha_x = \{0, \ldots, M_X - 1\}$ without loss of generality for some M_X discrete symbols). Note that unless otherwise stated, logarithms are taken by convention in base 2, giving units in bits.

The Shannon entropy was originally derived following an axiomatic approach, being derived as the unique formulation (up to the base of the logarithm) satisfying a certain set of properties or axioms (see further details in [1]). The uncertainty $H(X)$ associated with a measurement of X is equal to the average information required to predict it (see self-information below). $H(X)$ for a measurement x of X can also be interpreted as the minimal average number of bits required to encode or describe its value without losing information [2, 3].

The **joint entropy** of two random variables X and Y is a generalization to quantify the uncertainty of their joint distribution:

$$H(X,Y) = -\sum_{x \in \alpha_x} \sum_{y \in \alpha_y} p(x,y) \log_2 p(x,y). \tag{2}$$

We can of course write the above equation for multivariate $\mathbf{Z} = \{X, Y\}$, and then generalise to $H(\mathbf{X})$ for $\mathbf{X} = \{X_1, X_2, \ldots, X_G\}$. Such expressions for entropies of multivariates allows us to expand *all* of the following quantities for multivariate $\mathbf{X}$, $\mathbf{Y}$ etc.

[2] Notation for all quantities is summarised in Table II.

The **conditional entropy** of X given Y is the average uncertainty that remains about x when y is known:

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

The conditional entropy for a measurement x of X can be interpreted as the minimal average number of bits required to encode or describe its value without losing information, given that the receiver of the encoding already knows the value y of Y. The previous quantities are related by the following *chain rule*:

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

The **mutual information** (MI) between X and Y measures the average reduction in uncertainty about x that results from learning the value of y, or vice versa:

$$\begin{aligned} I(X;Y) &= \sum_{x\in\alpha_x}\sum_{y\in\alpha_y} p(x,y)\log_2 \frac{p(x \mid y)}{p(x)} && (5)\\ &= H(X) - H(X \mid Y). && (6)\end{aligned}$$

The MI is symmetric in the variables X and Y. The mutual information for measurements x and y of X and Y can be interpreted as the average number of bits *saved* in encoding or describing x given that the receiver of the encoding already knows the value of y, in comparison to the encoding of x without the knowledge of y. These descriptions of x with and without the value of y are both minimal without losing information. Note that one can compute the *self-information* $I(X;X) = H(X)$. Finally, one may define a generalization of the MI to a set of more than two variables $\mathbf{X} = \{X_1, X_2, \ldots, X_G\}$, known as the **multi-information** or **integration** [7]:

$$\begin{aligned} I(\mathbf{X}) &= I(X_1; X_2; \ldots; X_G)\\ &= \left(\sum_{g=1}^{G} H(X_g)\right) - H(X_1, X_2, \ldots, X_G). && (7)\end{aligned}$$

Equivalently we can split the set into two parts, $\mathbf{X} = \{\mathbf{Y}, \mathbf{Z}\}$, and express this quantity iteratively in terms of the multi-information of its components individually and the mutual information between those components:

$$I(\mathbf{X}) = I(\mathbf{Y}) + I(\mathbf{Z}) + I(\mathbf{Y};\mathbf{Z}). \tag{8}$$

The **conditional mutual information** between X and Y given Z is the mutual information between X and Y when Z is known:

$$\begin{aligned} I(X;Y \mid Z) &= \sum_{x\in\alpha_x}\sum_{y\in\alpha_y}\sum_{z\in\alpha_z} p(x,y,z)\log_2 \frac{p(x \mid y,z)}{p(x \mid z)} && (9)\\ &= \sum_{x\in\alpha_x}\sum_{y\in\alpha_y}\sum_{z\in\alpha_z} p(x,y,z)\log_2 \frac{p(x,y,z)p(z)}{p(x,z)p(y,z)} && (10)\\ &= H(X \mid Z) - H(X \mid Y,Z). && (11)\end{aligned}$$

Note that a conditional MI $I(X;Y \mid Z)$ may be either larger or smaller than the related unconditioned MI $I(X;Y)$ [3]. Such conditioning removes redundant information in Y and Z about X, but adds synergistic information which can only be decoded with knowledge of both Y and Z (see further description regarding "partial information decomposition", which refers to attempts to tease these components apart [72–76]).

One can consider the MI from two variables Y_1, Y_2 jointly to another variable X, $I(X; Y_1, Y_2)$, and using Eq. (4), Eq. (6) and Eq. (11) decompose this into the information carried by the first variable plus that carried by the second conditioned on the first:

$$I(X;Y_1,Y_2) = I(X;Y_1) + I(X;Y_2 \mid Y_1). \tag{12}$$

Of course, this *chain rule* generalises to multivariate $\mathbf{Y}$ of dimension greater than two.

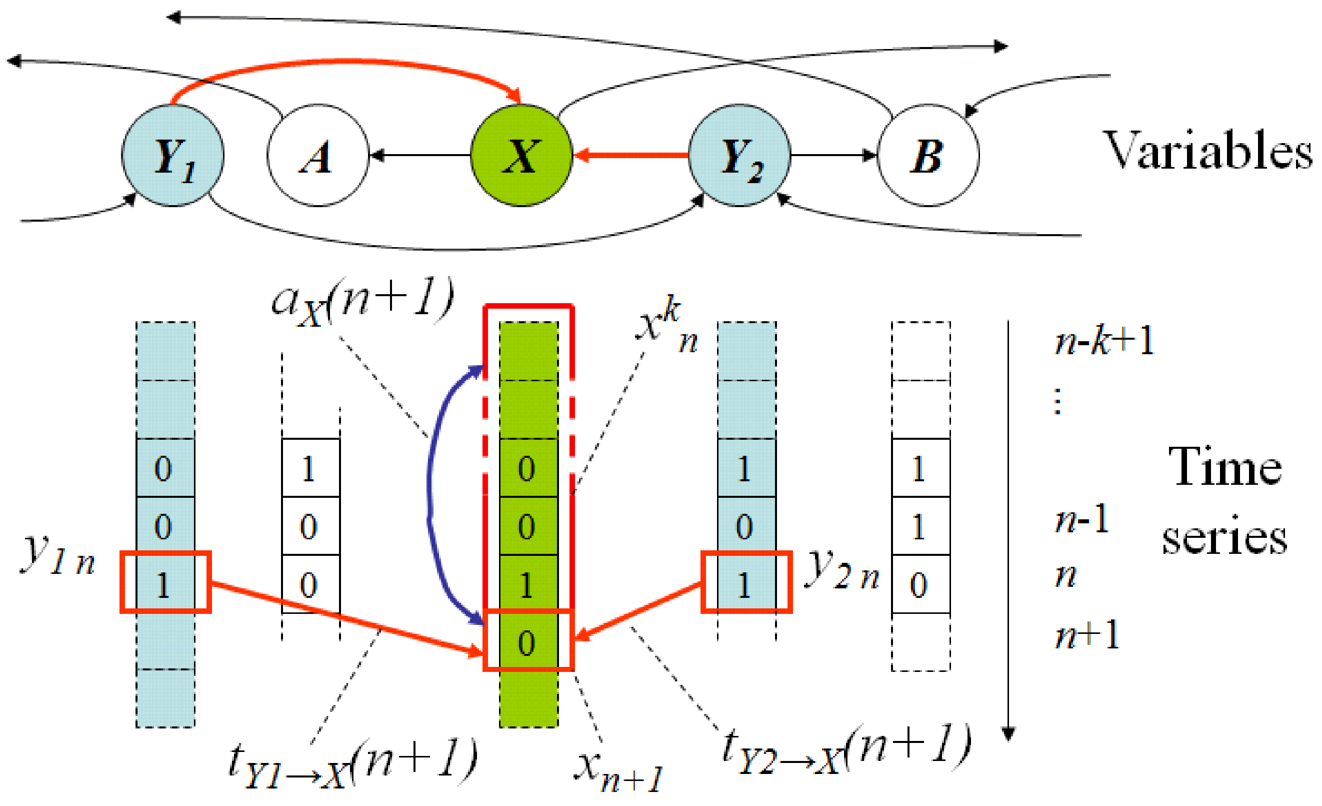

FIG. 1. Measures of information dynamics with respect to a destination variable X. We address the information content in a measurement x_{n+1} of X at time $n+1$ with respect to the active information storage $a_X(n+1,k)$, and local transfer entropies $t_{Y_1\rightarrow X}(n+1,k)$ and $t_{Y_2\rightarrow X}(n+1,k)$ from variables Y_1 and Y_2.

B. Measures of information dynamics

Next, we build on the basic measures of information theory to present measures of the dynamics of information processing. We focus on measures of information in *time-series processes* X of the random variables $\{\ldots X_{n-1}, X_n, X_{n+1}\ldots\}$ with process realisations $\{\ldots x_{n-1}, x_n, x_{n+1}\ldots\}$ for countable time indices n.

We briefly review the framework for *information dynamics* which was recently introduced in [15, 21, 23, 45, 46, 77]. This framework considers how the information in variable X_{n+1} is related to previous variables, e.g. X_n, of the process or other processes, addressing the fundamental question: *"where does the information in a random variable X_{n+1} in a time series come from?"*. As indicated in Fig. 1, this question is addressed in terms of information from the past of process X (i.e. the information *storage*), information contributed from other source processes Y (i.e. the information *transfer*), and how these sources combine (information *modification*). The goal is to decompose the information in the next observation of X, X_{n+1}, in terms of these information sources.

The **entropy rate** is defined by [2]:

$$H'_{\mu X} = \lim_{n\rightarrow\infty} \frac{1}{n} H(X_1, X_2, \ldots, X_n) \tag{13}$$

$$= \lim_{n\rightarrow\infty} \frac{1}{n} H(\mathbf{X}^{(n)}_n), \tag{14}$$

(where the limit exists) where we have used $\mathbf{X}^{(k)}_n = \{X_{n-k+1}, \ldots, X_{n-1}, X_n\}$ to denote the k consecutive variables of X up to and including time step n, which has realizations $\mathbf{x}^{(k)}_n = \{x_{n-k+1}, \ldots, x_{n-1}, x_n\}$. This quantity describes the limiting rate at which the entropy of n consecutive measurements of X grow with n. A related definition for a

(conditional) entropy rate is given by:[3]

$$H_{\mu X} = \lim_{n\to\infty} H(X_n \mid X_1, X_2, \dots, X_{n-1}) \tag{15}$$

$$= \lim_{n\to\infty} H(X_n \mid \mathbf{X}_{n-1}^{(n-1)}). \tag{16}$$

For stationary processes X, the limits for the two quantities $H'_{\mu X}$ and $H_{\mu X}$ exist (i.e. the average entropy rate converges) and are equal [2].

For our purposes in considering information dynamics, we are interested in the conditional formulation $H_{\mu X}$, since it explicitly describes how one random variable X_n is related to the previous instances $\mathbf{X}_{n-1}^{(n-1)}$. For practical usage, we are particularly interested in estimation of $H_{\mu X}$ with finite-lengths k, and in estimating it regarding the information at different time indices n. That is to say, we use the notation $H_{\mu X_{n+1}}(k)$ to describe finite-k estimates of the conditional entropy rate in X_{n+1} given $\mathbf{X}_n^{(k)}$:

$$H_{\mu X_{n+1}}(k) = H(X_{n+1} \mid \mathbf{X}_n^{(k)}). \tag{17}$$

Assuming stationarity we define:

$$H_{\mu X}(k) = H_{\mu X_{n+1}}(k) \tag{18}$$

for any n, and of course letting $k = n$ and joining Eq. (16) and Eq. (17) we have $\lim_{n\to\infty} H_{\mu X_{n+1}}(k) = H_{\mu X}$.

Next, the **effective measure complexity** [78] or **excess entropy** [25] quantifies the total amount of structure or memory in the process X, and is computed in terms of the slowness of the approach of the conditional entropy rate estimates to their limiting value:

$$E_X = \sum_{k=0}^{\infty} (H_{\mu X}(k) - H_{\mu X}). \tag{19}$$

When the process X is stationary we may represent the excess entropy as the mutual information between the semi-infinite past and semi-infinite future of the process:

$$E_X = \lim_{k\to\infty} E_X(k), \tag{20}$$

$$E_X(k) = I(X_n^{(k)}; \mathbf{X}_{n+1}^{(k^+)}), \tag{21}$$

where $\mathbf{X}_{n+1}^{(k^+)}$ refers to the next k values $\{X_{n+1}, X_{n+2}, \dots, X_{n+k}\}$ with realizations $\mathbf{x}_{n+1}^{(k^+)} = \{x_{n+1}, x_{n+2}, \dots, x_{n+k}\}$, and $E_X(k)$ are finite-k estimates of E_X. This formulation is known as the **predictive information** [24], as it highlights that the excess entropy captures the information in a system's past which can also be found in its future. It is the most appropriate formulation for our purposes, since it provides a clear interpretation as information storage. That is, the excess entropy can be viewed in this formulation as measuring information from the past of the process that is stored – potentially in a distributed fashion in external variables – and is used at some point in the future of the process [23]. This contrasts with the statistical complexity [79, 80], an upper bound to the excess entropy, which measures *all* information which is *relevant* to the prediction of the future of the process states; i.e. the stored information which *may be used* in the future [23].

In contrast again, the **active information storage** (AIS) was introduced [23] to measure how much of the information from the past of the process X is observed to be *in use* in computing its *next observation*. This measure of information storage more directly addresses our key question of determining the sources of the information in the next observation X_{n+1}. The active information storage is the average mutual information between realizations $\mathbf{x}_n^{(k)}$ of the past state $\mathbf{X}_n^{(k)}$ (as $k \to \infty$) and the corresponding realizations x_{n+1} of the next value X_{n+1} of process X:

$$A_X = \lim_{k\to\infty} A_X(k), \tag{22}$$

$$A_X(k) = I(\mathbf{X}_n^{(k)}; X_{n+1}). \tag{23}$$

[3] Note that we have reversed the use of the primes in the notation from [2], in line with [25].

We note that $\mathbf{x}_n^{(k)}$ are Takens' *embedding vectors* [81] with *embedding dimension* k, which capture the underlying *state* of the process X for Markov processes of order k.[4] As such, one needs to at least take k at the Markovian order of X in order to capture all relevant information in the past of X, otherwise (for non-Markovian processes) the limit $k \to \infty$ is theoretically required in general [23]. We also note that since:

$$A_X = H(X) - H_{\mu X}, \tag{24}$$

then the limit in Eq. (22) exists for stationary processes (i.e. $A(X)$ converges with $k \to \infty$) [23].

Arguably the most important measure in this toolkit is Schreiber's **transfer entropy** (TE) measure [22]. TE captures the concept of information transfer, as the amount of information that a source process provides about a destination (or target) process' next state in the context of the destination's past. Quantitatively, this is the average mutual information from realizations $\mathbf{y}_n^{(l)}$ of the state $\mathbf{Y}_n^{(l)}$ of a source process Y to the corresponding realizations x_{n+1} of the next value X_{n+1} of the destination process X, conditioned on realizations $\mathbf{x}_n^{(k)}$ of its previous state $\mathbf{X}_n^{(k)}$:

$$T_{Y \to X}(l) = \lim_{k \to \infty} T_{Y \to X}(k, l), \tag{25}$$

$$T_{Y \to X}(k, l) = I(\mathbf{Y}_n^{(l)}; X_{n+1} \mid \mathbf{X}_n^{(k)}). \tag{26}$$

TE has become a very popular tool in complex systems in general (e.g. [45, 51, 57, 62, 70, 82, 83]) and in computational neuroscience in particular (e.g. [31, 33, 34, 37, 84]). For multivariate Gaussians, the TE is equivalent (up to a factor of 2) to the Granger causality [85].

There are a number of important considerations regarding the use of this measure (see further discussion in [28, 29, 45, 86, 87]). First, for the embedding vectors $\mathbf{x}_n^{(k)}$ one needs to at least take k larger than the Markovian order of X in order to eliminate any AIS from being redundantly measured in the TE.[5] Then, one may need to extend k to capture synergies generated in x_{n+1} between the source $\mathbf{y}_n^{(l)}$ and earlier values in X. For non-Markovian processes X (or non-Markovian processes when considered jointly with the source), one should theoretically take the limit as $k \to \infty$ [45]. Setting k in this manner gives the perspective to separate information storage and transfer in the distributed computation in process X, and allows one to interpret the transfer entropy as properly representing information transfer [45, 88].

Also, note that the transfer entropy can be defined for an arbitrary source-destination delay u [89]:

$$T_{Y \to X}(k, l, u) = I(\mathbf{Y}_{n+1-u}^{(l)}; X_{n+1} \mid \mathbf{X}_n^{(k)}), \tag{27}$$

and indeed that this should be done for the appropriate causal delay $u > 0$. For ease of presentation here, we describe the measures for $u = 1$ only, though all are straightforward to generalise and are implemented with generic u in JIDT.

Furthermore, considering the source *state* $\mathbf{y}_n^{(l)}$ rather than a scalar y_n is most appropriate where the observations y mask a hidden Markov process which is causal to X (as is the case in analysis of brain imaging data), or where multiple past values of Y in addition to y_n are causal to x_{n+1}. Otherwise, where y_n is directly causal to x_{n+1}, and where it is the only direct causal source in Y, we use only $l = 1$ [45, 88].

Finally, for proper interpretation as information transfer, Y is constrained among the causal information contributors to X [88]. With that said, the concepts of information transfer and causality are complementary but distinct, and TE should not be thought of as measuring causal effect [88, 90, 91]. We have also provided a thermodynamic interpretation of transfer entropy in [92, 93], as being proportional to external entropy production, possibly due to irreversibility.

Now, the transfer entropy may also be conditioned on other possible sources Z to account for their effects on the destination. The **conditional transfer entropy**[6] was introduced for this purpose [45, 46]:

$$T_{Y \to X|Z}(l) = \lim_{k \to \infty} T_{Y \to X|Z}(k, l), \tag{28}$$

$$T_{Y \to X|Z}(k, l) = I(\mathbf{Y}_n^{(l)}; X_{n+1} \mid \mathbf{X}_n^{(k)}, Z_n), \tag{29}$$

[4] We can use an embedding delay τ to give $\mathbf{x}_n^{(k)} = \{x_{n-(k-1)\tau}, \ldots, x_{n-\tau}, x_n\}$, where this helps to better empirically capture the state from a finite sample size. Non-uniform embeddings (i.e. with irregular delays) may also be useful [32] (not implemented in JIDT at this stage).

[5] The destination's embedding dimension should be increased before that of the source, for this same reason.

[6] This is sometimes known as "multivariate" TE, though this term can be confused with TE applied to multivariate source and destination variables (i.e. the collective TE).

Note that Z_n may represent an embedded state of another variable, or be explicitly multivariate, and may of course be from time index/indices other than n. Transfer entropies conditioned on other variables have been used in several biophysical and neuroscience applications, e.g. [31, 32, 94]. We typically describe TE measurements which are not conditioned on any other variables (as in Eq. (25)) as **pairwise** or **apparent transfer entropy**, and measurements conditioned on *all* other causal contributors to X_{n+1} as **complete transfer entropy** [45]. Further, one can consider multivariate sources $\mathbf{Y}$, in which case we refer to the measure $T_{\mathbf{Y}\rightarrow X}(k,l)$ as a **collective transfer entropy** [46].

Finally, while how to measure information modification remains an open problem (see [75]), JIDT contains an implementation of an early attempt at capturing this concept in the **separable information** [46]:

$$S_X = \lim_{k\rightarrow\infty} S_X(k), \tag{30}$$

$$S_X(k) = A_X(k) + \sum_{Y\in\mathbf{V}_X\setminus X} T_{Y\rightarrow X}(k, l_Y). \tag{31}$$

Here, $\mathbf{V}_X$ represents the set of causal information sources $\mathbf{V}_X$ to X, while l_Y is the embedding dimension for source Y.

C. Local information-theoretic measures

Local information-theoretic measures (also known as **pointwise information-theoretic measures**) characterise the information attributed with *specific* measurements x, y and z of variables X, Y and Z [86], rather than the traditional average information measures associated with these variables introduced in Section II A and Section II B. Although they are deeply ingrained in the fabric of information theory, and heavily used in some areas (e.g. in natural language processing [95]), until recently [23, 45, 46, 77, 80, 96, 97] local information-theoretic measures were rarely applied to complex systems.

That these local measures are now being applied to complex systems is important, because they provide a direct, model-free, mechanism to analyse the *dynamics* of how information processing unfolds in time. In other words: traditional, average information-theoretic measures would return one value to characterise, for example, the transfer entropy between Y and X. Local transfer entropy on the other hand, returns a time-series of values to characterise the information transfer from Y to X as a function of time, so as to directly reveal the *dynamics* of their interaction. Indeed, it is well-known that local values (within a global average) provide important insights into the dynamics of nonlinear systems [98].

A more complete description of local information-theoretic measurements is provided in [86]. Here we provide a brief overview of the local values of the measures previously introduced.

The most illustrative local measure is of course the **local entropy** or **Shannon information content**. The Shannon information content of an outcome x of measurement of the variable X is [3]:

$$h(x) = -\log_2 p(x). \tag{32}$$

Note that by convention we use lower-case symbols to denote local information-theoretic measures. The Shannon information content was shown to be the unique formulation for a local entropy (up to the base of the logarithm) satisfying required properties corresponding to those of the average Shannon entropy (see [99] for details). Now, the quantity $h(x)$ is simply the information content attributed to the specific symbol x, or the information required to predict or uniquely specify that specific value. Less probable outcomes x have higher information content than more probable outcomes, and we have $h(x) \geq 0$. The Shannon information content of a given symbol x is the *code-length* for that symbol in an optimal encoding scheme for the measurements X, i.e. one that produces the minimal expected code length.

We can form all traditional information-theoretic measures as the *average* or *expectation value* of their corresponding local measure, e.g.:

$$H(X) = \sum_{x\in\alpha_x} p(x)h(x), \tag{33}$$

$$= \langle h(x) \rangle . \tag{34}$$

While the above represents this as an average over the relevant ensemble, we can write the same average over all of the N samples x_n (with each sample given an index n) used to generate the probability distribution function (PDF)

$p(x)$ [45, 86], e.g.:

$$H(X) = \frac{1}{N}\sum_{n=1}^{N} h(x_n), \tag{35}$$

$$= \langle h(x_n)\rangle_n \,. \tag{36}$$

Next, we have the **conditional Shannon information content** (or **local conditional entropy**) [3]:

$$h(x \mid y) = -\log_2 p(x \mid y), \tag{37}$$

$$h(y,) = h(y) + h(x \mid y), \tag{38}$$

$$H(X \mid Y) = \langle h(x \mid y)\rangle \,. \tag{39}$$

As above, local quantities satisfy corresponding chain rules to those of their averaged quantities.

The **local mutual information** is defined (uniquely, see [100, ch. 2]) as "the amount of information provided by the occurrence of the event represented by y_i about the occurrence of the event represented by x_i", i.e.:

$$i(x;y) = \log_2 \frac{p(x \mid y)}{p(x)}, \tag{40}$$

$$= h(x) - h(x \mid y), \tag{41}$$

$$I(X;Y) = \langle i(x;y)\rangle \,. \tag{42}$$

$i(x;y)$ is symmetric in x and y, as is the case for $I(x;y)$. The local mutual information is the difference in code lengths between coding the value x in isolation (under the optimal encoding scheme for X), or coding the value x given y (under the optimal encoding scheme for X given Y). In other words, this quantity captures the coding "cost" for x in not being aware of the value y.

Of course this "cost" averages to be non-negative, however the local mutual information may be either positive or negative for a specific pair x, y. Positive values are fairly intuitive to understand: $i(x;y)$ is positive where $p(x \mid y) > p(x)$, i.e. knowing the value of y *increased* our expectation of (or positively informed us about) the value of the measurement x. Negative values simply occur in Eq. (40) where $p(x \mid y) < p(x)$. That is, knowing the value of y changed our belief $p(x)$ about the probability of occurrence of the outcome x to a smaller value $p(x \mid y)$, and hence we considered it less likely that x would occur when knowing y than when not knowing y, in a case were x nevertheless occurred. Consider the following example from [86], of the probability that it will rain today, $p(\texttt{rain} = 1)$, and the probability that it will rain given that the weather forecast said it would not, $p(\texttt{rain} = 1 \mid \texttt{rain\_forecast} = 0)$. We could have $p(\texttt{rain} = 1 \mid \texttt{rain\_forecast} = 0) < p(\texttt{rain} = 1)$, so we would have $i(\texttt{rain} = 1; \texttt{rain\_forecast} = 0) < 0$, because we considered it less likely that rain would occur today when hearing the forecast than without the forecast, in a case where rain nevertheless occurred. Such negative values of MI are actually quite meaningful, and can be interpreted as there being negative information in the value of y about x. We could also interpret the value y as being *misleading* or *misinformative* about the value of x, because it *lowered* our expectation of observing x prior to that observation being made in this instance. In the above example, the weather forecast was misinformative about the rain today.

Note that the local mutual information $i(x;y)$ measure above is distinct from *partial* localization expressions, i.e. the partial mutual information or specific information $I(x;Y)$ [101], which consider information contained in specific values x of one variable X about the other (unknown) variable Y. While there are two valid approaches to measuring partial mutual information, as above there is only one valid approach for the fully local mutual information $i(x;y)$ [100, ch. 2].

The **local conditional mutual information** is similarly defined [100, ch. 2]:

$$i(x;y \mid z) = \log_2 \frac{p(x \mid y,z)}{p(x \mid z)}, \tag{43}$$

$$= h(x \mid z) - h(x \mid y,z), \tag{44}$$

$$I(X;Y \mid Z) = \langle i(x;y \mid z)\rangle \,. \tag{45}$$

$I(X;Y \mid Z)$ is the difference in code lengths (or coding cost) between coding the value x given z (under the optimal encoding scheme for X given Z), or coding the value x given both y and z (under the optimal encoding scheme for X given Y and Z). As per I(X ; Y), the local conditional MI is symmetric in x and y, and may take positive or negative values.

Local measures of information dynamics are formed via the local definitions of the basic information-theoretic measures above. Here, the local measures pertain to realisations x_n, $\mathbf{x}_n^{(k)}$, $\mathbf{y}_n^{(l)}$, etc, of the processes at specific time index n. The PDFs may be estimated either from multiple realisations of the process for time index n, or from multiple observations over time from one (or several) full time-series realisation(s) where the process is stationary (see comments in [86]).

We have the **local entropy rate**:[7]

$$h_{\mu X}(n+1,k) = h(x_{n+1} \mid \mathbf{x}_n^{(k)}), \tag{46}$$
$$H_{\mu X}(k) = \langle h_{\mu X}(n,k) \rangle . \tag{47}$$

Next, the **local excess entropy** is defined as (via the predictive information formulation from Eq. (21)) [80]:

$$e_X(n+1,k) = i(\mathbf{x}_n^{(k)}; \mathbf{x}_{n+1}^{(k^+)}), \tag{48}$$
$$E_X(k) = \langle e_X(n,k) \rangle . \tag{49}$$

We then have the **local active information storage** $a_X(n+1)$ [23]:

$$a_X(n+1,k) = i(\mathbf{x}_n^{(k)}; x_{n+1}), \tag{50}$$
$$A_X(k) = \langle a_X(n+1,k) \rangle . \tag{51}$$

The local values of active information storage measure the dynamics of information storage at different time points within a system, revealing to us how the use of memory fluctuates during a process. As described for the local MI, $a_X(n+1,k)$ may be positive or negative, meaning the past history of the process can either positively inform us or actually *misinform* us about its next value [23]. Fig. 1 indicates a local active information storage measurement for time-series process X.

The **local transfer entropy** is [45]:

$$t_{Y\to X}(n+1,k,l) = i(\mathbf{y}_n^{(l)}; x_{n+1} \mid \mathbf{x}_n^{(k)}), \tag{52}$$
$$T_{Y\to X}k,l = \langle t_{Y\to X}(n+1,k,l) \rangle . \tag{53}$$

These local information transfer values measure the dynamics of transfer in time between a given pair of time-series processes, revealing to us how information is transferred in time and space. Fig. 1 indicates a local transfer entropy measurement for a pair of processes $Y \to X$.

Finally, we have the **local conditional transfer entropy** [45, 46]:

$$t_{Y\to X|Z}(n+1,k,l) = i(\mathbf{y}_n^{(l)}; x_{n+1} \mid \mathbf{x}_n^{(k)}, z_n), \tag{54}$$
$$T_{Y\to X|Z}(n+1,k,l) = \left\langle t_{Y\to X|Z}(n+1,k,l) \right\rangle . \tag{55}$$

D. Differential entropy

Note that all of the information-theoretic measures above considered a discrete alphabet of symbols $alpha_x$ for a given variable X. When X in fact is a continuous-valued variable, we shift to consider **differential entropy** measurements (see [2, ch. 9]). We briefly discuss differential entropy, since some of our estimators discussed in Section III evaluate these quantities for continuous-valued variables rather than strictly Shannon entropies.

The differential entropy of a continuous variable X with probability density function $f(x)$ is defined as [2, ch. 9]:

$$H_D(X) = -\int_{S_X} f(x) \log f(x) dx, \tag{56}$$

where S_X is the set where $f(x) > 0$. The differential entropy is strongly related to the Shannon entropy, but has important differences to what the Shannon entropy would return on discretizing the same variables. Primary amongst these differences is that $H_D(X)$ changes with scaling of the variable X, and that it can be negative.

[7] For the local measures of information dynamics, while formal definitions may be provided by taking the limit as $k \to \infty$, we will state only the formulae for their finite-k estimates.

Joint and conditional ($H_D(X \mid Y)$) differential entropies may be evaluated from Eq. (56) expressions using the same chain rules from the Shannon measures. Similarly, the differential mutual information may be defined as [2, ch. 9]:

$$I_D(X;Y) = H_D(X) - H_D(X \mid Y), \tag{57}$$

$$= \int_{S_X,S_Y} f(x,y) \log \frac{f(x,y)}{f(x)f(y)} \, dx \, dy. \tag{58}$$

Crucially, the properties of $I_D(X;Y)$ are the same as for discrete variables, and indeed $I_D(X;Y)$ is equal to the discrete MI $I(X^\Delta;Y^\Delta)$ for discretizations X^Δ and Y^Δ with bin size Δ, in the limit $\Delta \to 0$ [2, ch. 9]. Conditional MI and other derived measures (e.g. transfer entropy) follow.

E. Statistical significance testing

In *theory*, the MI between two unrelated variables Y and X is equal to 0. The same goes for the TE between two variables Y and X with no directed relationship, or the conditional MI between Y and X given Z where there is no conditional relationship. In *practice*, where the MI, conditional MI or TE are empirically measured from a finite number of samples N, a bias of a non-zero measurement is likely to result even where there is no such (directed) relationship. A common question is then whether a given empirical measurement is statistically different from 0, and therefore represents sufficient evidence for a (directed) relationship between the variables.

This question is addressed in the following manner [29, 33, 34, 83, 84, 102, 103]. We form a *null hypothesis* H_0 that there is no such relationship, and then make a test of statistical significance of evidence (our original measurement) in support of that hypothesis. To perform such a test, we need to know what the *distribution* for our measurement would look like if H_0 was true, and then evaluate a p-value for sampling our actual measurement from this distribution. If the test fails, we accept the alternate hypothesis that there is a (directed) relationship.

For example, for an MI measurement $I(Y;X)$, we generate the distribution of *surrogate* measurements $I(Y^s;X)$ under the assumption of H_0. Here, Y^s represents *surrogate* variables for Y generated under H_0, which have the same statistical properties as Y, but any potential correlation with X is destroyed. Specifically, this means that $p(x \mid y)$ in Eq. (6) is distributed as $p(x)$ (with $p(y)$ retained also).

In some situations, we can compute the distribution of $I(Y^s;X)$ analytically. For example, for linearly-coupled Gaussian multivariates $\mathbf{X}$ and $\mathbf{Y}$, $I(\mathbf{Y}^s;\mathbf{X})$ measured in *nats* follows a chi-square distribution, specifically $\chi^2_{|\mathbf{X}||\mathbf{Y}|}/2N$ with $|\mathbf{X}||\mathbf{Y}|$ degrees of freedom, where $|\mathbf{X}|$ ($|\mathbf{Y}|$) is the number of Gaussian variables in vector $\mathbf{X}$ ($\mathbf{Y}$) [104, 105]. Also, for discrete variables X and Y with alphabet sizes M_X and M_Y, $I(Y^s;X)$ measured in *bits* follows a chi-square distribution, specifically $\chi^2_{(M_X-1)(M_Y-1)}/(2N \log 2)$ [105, 106]. Note that these distributions are followed *asymptotically* with the number of samples N, and the approach is much slower for discrete variables with skewed distributions [107], *which reduces the utility of this analytic result in practice.*[8] Barnett and Bossomaier [83] generalise these results to state that a model-based null distribution (in *nats*) will follow $\chi^2_d/2N$, where d is the "difference between the number of parameters" in a full model (capturing $p(x \mid y)$ in Eq. (6)) and a null model (capturing $p(x)$ only).

Where no analytic distribution is known, the distribution of $I(Y^s;X)$ must be computed empirically. This is done by a bootstrapping method [29, 33, 34, 84, 102, 103], creating a large number of surrogate time-series pairs $\{Y^s, X\}$ by shuffling the samples of Y (so as to retain $p(x)$ and $p(y)$ but not $p(x \mid y)$), and computing a population of $I(Y^s;X)$ values.

Now, for a conditional MI, we generate the distribution of $I(Y^s;X \mid Z)$ under H_0, which means that $p(x \mid y,z)$ in Eq. (9) is distributed as $p(x \mid z)$ (with $p(y)$ retained also).[9] The asymptotic distribution may be formed analytically for linearly-coupled Gaussian multivariates defined above [83, 104] (in *nats*) as $\chi^2_{|\mathbf{X}||\mathbf{Y}|}/2N$ with $|\mathbf{X}||\mathbf{Y}|$ degrees of freedom – interestingly, this does *not* depend on the Z variable. Similarly, for discrete variables the asymptotic distribution (in *bits*) is $\chi^2_{(M_X-1)(M_Y-1)M_Z}/(2N \log 2)$ [106]. Again, the distribution of $I(Y^s;X \mid Z)$ is otherwise computed by bootstrapping, this time by creating surrogate time-series $\{Y^s, X, Z\}$ by shuffling the samples of Y (retaining $p(x \mid z)$ and $p(y)$ but not $p(x \mid y,z)$), and computing a population of $I(Y^s;X \mid Z)$ values.

---

[8] See Section V F for an investigation of this.

[9] Clearly, this approach specifically makes a *directional* hypothesis test of Eq. (9) rather than a non-directional test of Eq. (10). Asymptotically these will be the same anyway (as is clear for the analytic cases discussed here). In practice, we favour this somewhat directional approach since in most cases we are indeed interested in the directional question of whether Y adds information to X in the context of Z.

Statistical significance testing for the transfer entropy can be handled as a special case of the conditional MI. For linear-coupled Gaussian multivariates $\mathbf{X}$ and $\mathbf{Y}$, the null $T_{Y^s \rightarrow X}(k,l)$ (in *nats*) is asymptotically $\chi^2/2N$ distributed with $l|\mathbf{X}||\mathbf{Y}|$ degrees of freedom [83, 104, 107], while for discrete X and Y, $T_{Y^s \rightarrow X}(k,l)$ (in *bits*) is asymptotically $\chi^2/(2N \log 2)$ distributed with $(M_X - 1)(M_Y^l - 1)M_X^k$ degrees of freedom [83]. Again, the distribution of $T_{Y^s \rightarrow X}(k,l)$ is otherwise computed by bootstrapping [29, 33, 34, 84, 102, 103], under which surrogates must preserve $p(x_{n+1} \mid x_n^{(k)})$ but not $p(x_{n+1} \mid \mathbf{x}_n^{(k)}, \mathbf{y}_n^{(l)})$. Directly shuffling the series Y to create the Y^s is *not* a valid approach, since it destroys $\mathbf{y}_n^{(l)}$ vectors (unless $l = 1$). Valid approaches include: shuffling the $\mathbf{y}_n^{(l)}$ amongst the set of $\{x_{n+1}, \mathbf{x}_n^{(k)}, \mathbf{y}_n^{(l)}\}$ tuples;[10] rotating the Y time-series (where we have stationarity); or swapping sample time series Y_i between different trials i in an ensemble approach [29, 34, 84, 108]. Conditional TE may be handled similarly as a special case of a conditional MI.

Finally, we note that such assessment of statistical significance is often used in the application of effective network inference from multivariate time-series data (e.g. [29, 33, 34, 84]). In this and other situations where multiple hypothesis tests are considered together, one should correct for multiple comparisons using family-wise error rates (e.g. Bonferroni correction) or false discovery rates.

III. ESTIMATION TECHNIQUES

While the mathematical formulation of the quantities in Section II are relatively straightforward, empirically estimating them in practice from a finite number N of samples of time-series data can be a complex process, and is dependent on the type of data you have and its properties. All estimators are subject to bias and variance due to finite sample size.

In this section, we introduce the various types of estimators which are included in JIDT. Such estimators are discussed in some depth in [87], for the transfer entropy in particular.

A. Discrete-valued variables

For discrete variables X, Y, Z etc, the definitions in Section II may be used directly, by counting the matching configurations in the available data to obtain the relevant plug-in probability estimates (e.g. $\hat{p}(x \mid y)$ and $\hat{p}(x)$ for MI). This approach may be taken for both local and average measures. Several bias correction techniques are available, e.g. [109, 110], though not yet implemented in JIDT.

B. Continuous-valued variables

For continuous variables X, Y, Z, one could simply discretise or bin the data and apply the discrete estimators above. Alternatively, we can use an estimator that harnesses the continuous nature of the variables, dealing with the differential entropy and probability density functions. The latter is more complicated but yields a more accurate result. We discuss several such estimators in the following.

1. *Gaussian-distribution model*

The simplest estimator uses a *multivariate Gaussian model* for the relevant variables, assuming linear interactions between them. Under this model, for $\mathbf{X}$ (of d dimensions) the average entropy has the form [2]:

$$H(\mathbf{X}) = \frac{1}{2} \ln \left((2\pi e)^d \mid \Omega_{\mathbf{X}} \mid \right), \tag{59}$$

(in *nats*) where $\mid \Omega_{\mathbf{X}} \mid$ is the determinant of the $d \times d$ covariance matrix $\Omega_{\mathbf{X}} = \overline{\mathbf{X}\mathbf{X}^T}$, and the overbar "represents an average over the statistical ensemble" [111]. Any standard information-theoretic measure in Section II can then be obtained from sums and differences of these joint entropies. For example, Kaiser and Schreiber [112] demonstrated

[10] This is the approach taken in JIDT.

how to compute transfer entropy in this fashion. These estimators are fast ($\mathrm{O}\left(Nd^2\right)$) and parameter-free, but subject to the linear-model assumption.

Since PDFs were effectively bypassed in Eq. (59), the local entropies (and by sums and differences, other local measures) can be obtained by first reconstructing the probability of a given observation $\mathbf{x}$ in a multivariate process with covariance matrix $\Omega_{\mathbf{X}}$:

$$p(\mathbf{x}) = \frac{1}{(\sqrt{2\pi})^d \mid \Omega_{\mathbf{X}} \mid^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T \Omega_{\mathbf{X}}^{-1}(\mathbf{x}-\mu)\right), \tag{60}$$

(where μ is the vector of expectation values of $\mathbf{x}$), then using these values directly in the equation for the given local quantity as a plug-in estimate [86].[11]

2. *Kernel estimation*

Using *kernel-estimators* (e.g. see [22, 113]), the relevant joint PDFs (e.g. $\hat{p}(x,y)$ and $\hat{p}(x)$ for MI) are estimated with a *kernel function* Θ, which measures "similarity" between pairs of samples $\{x_n, y_n\}$ and $\{x_{n'}, y_{n'}\}$ using a resolution or *kernel width* r. For example, we can estimate:

$$\hat{p}_r(x_n, y_n) = \frac{1}{N}\sum_{n'=1}^{N} \Theta\left(\left|\begin{pmatrix} x_n - x_{n'} \\ y_n - y_{n'} \end{pmatrix}\right| - r\right). \tag{61}$$

By default Θ is the step kernel ($\Theta(x > 0) = 0$, $\Theta(x \leq 0) = 1$), and the norm $|\cdot|$ is the maximum distance. This combination – a *box kernel* – is what is implemented in JIDT. It results in $\hat{p}_r(x_n, y_n)$ being the proportion of the N values which fall within r of $\{x_n, y_n\}$ in both dimensions X and Y. Different resolutions r may be used for the different variables, whilst if using the same r then prior normalisation of the variables is sensible. Other choices for the kernel Θ and the norm $|\cdot|$ are possible. Conditional probabilities may be defined in terms of their component joint probabilities. These plug-in estimates for the PDFs are then used directly in evaluating a local measure for each sample $n \in [1, N]$ and averaging these over all samples, i.e. via Eq. (36) for $H(X)$ rather than via Eq. (1) (e.g. see [112] for transfer entropy). Note that methods for bias-correction here are available for individual entropy estimates (e.g. as proposed by Grassberger for the box kernel [114]), but when combined for sums of entropies (as in MI, TE, etc.) "this approach is not viable ... since the finite sample fluctuations ... are not independent and we cannot correct their bias separately" [112].[12] Such issues are addressed by the Kraskov-Stögbauer-Grassberger estimator in the next section.

Kernel estimation can measure non-linear relationships and is model-free (unlike Gaussian estimators), though is sensitive to the parameter choice for resolution r [22, 112] (see below), is biased and is less time-efficient (with naive algorithms requiring $\mathrm{O}\left(N^2\right)$ time, but efficient neighbour searching can reduce this to $\mathrm{O}\left(N \log N\right)$ or box-assisted methods $\mathrm{O}\left(N\right)$ [113]).[13] Box-assisted methods are used in JIDT.

Selecting a value for r can be difficult, with too small a value yielding undersampling effects (e.g. MI the values diverge [22]) whilst too large a value ignores subtleties in the data. One can heuristically determine a lower bound for r to avoid undersampling. Assuming all data are normalised (such that r then refers to a number of standard deviations) and spread somewhat evenly, the values for each variable roughly span 6 standard deviations and a given sample has $\sim N/(6/2r)$ coincident samples in any given dimension or $\sim N/(6/2r)^d$ in the full joint space of d dimensions. Requiring some number K of coincident samples on average within r (Lungarella et al. [115] suggest $K \geq 3$ though at least 10 is more common), we then solve for $K \leq N/(6/2r)^d$.[14] Even within these extremes however, the choice of r can have a very large influence on the comparative results of the measure, see [22, 112].

3. *Kraskov-Stögbauer-Grassberger (KSG) technique*

Kraskov, Stögbauer and Grassberger (KSG) [117, 118] improved on (box) kernel estimation for MI by combining several specific enhancements designed to reduce errors when handling a small number of observations. These include:

[11] This method can produce a local or pointwise Granger causality, as a local transfer entropy using a Gaussian model estimator.
[12] As such, these are not implemented in JIDT, except for one method available for testing with the kernel estimator for TE.
[13] These quoted time complexities ignore the dependency on dimension d of the data, but will require a multiplier of at least d to determine norms, with larger multipliers perhaps required for more complicated box-assisted algorithms.
[14] More formally, one can consider the average number of coincidences for the *typical set*, see [2, 116].

the use of Kozachenko-Leonenko estimators (see [119]) of log-probabilities via nearest-neighbour counting; bias correction; and a fixed number K of nearest-neighbours in the full X-Y joint space. The latter effectively means using a dynamically altered kernel width r to adjust to the density of samples in the vicinity of any given observation, which smooths out errors in the PDF estimation. For each sample $\{x, y\}$, one finds the Kth nearest neighbour in the full $\{x, y\}$ space (using max norms to compare x and y distances), and sets kernel widths r_x and r_y from it. The authors then propose two different algorithms for determining r_x and r_y from the Kth nearest neighbour.

For algorithm 1, r_x and r_y are set to the maximum of the x and y distances to the Kth nearest neighbour, and one then counts the number of neighbours n_x and n_y strictly within these widths in each marginal space. Then the averages of n_x and n_y over all samples are used to compute:

$$I^{(1)}(X;Y) = \psi(K) - \langle \psi(n_x+1) + \psi(n_y+1) \rangle + \psi(N), \tag{62}$$

(in *nats*) where ψ denotes the digamma function.

For algorithm 2, r_x and r_y are set separately to the x and y distances to the Kth nearest neighbour, and one then counts the number of neighbours n_x and n_y within and on these widths in each marginal space. Again one uses the averages of n_x and n_y over all samples to compute (in *nats*):

$$I^{(2)}(X;Y) = \psi(K) - \frac{1}{K} - \langle \psi(n_x) + \psi(n_y) \rangle + \psi(N). \tag{63}$$

Crucially, the estimator is bias corrected, and is demonstrated to be quite robust to variations in K (from $K = 4$ upwards, as variance in the estimate decreases with K) [117]. Of the two algorithms: algorithm 1 (Eq. (62)) is more accurate for smaller numbers of samples but is more biased, while algorithm 2 (Eq. (63)) is more accurate for very large sample sizes.

Kraskov originally proposed that TE could be computed [118] as the difference between two MIs (with each estimated using the aforementioned technique). However, the KSG estimation technique has since been properly extended to conditional MI [120] and transfer entropy (originally in [121] and later for algorithm 2 in [29]) with single estimators. Here for $I(X;Y \mid Z)$, for each sample $\{x, y, z\}$, one finds the Kth nearest neighbour in the full $\{x, y, z\}$ space (using max norms to compare x, y and z distances), and sets kernel widths r_x, r_y and r_z from it. Following algorithm 1, r_z and $\{r_{xz}, r_{yz}\}$ are set to the maximum of the marginal distances to the Kth nearest neighbour, and one then counts $\{n_z, n_{xz}, n_{yz}\}$ strictly within this width (where n_{xz} and n_{yz} refer to counts in the joint $\{x, z\}$ and $\{y, z\}$ joint spaces) to obtain [120, 121]:

$$I^{(1)}(X;Y \mid Z) = \psi(K) + \langle \psi(n_z+1) - \psi(n_{xz}+1) - \psi(n_{yz}+1) \rangle . \tag{64}$$

While following algorithm 2, $\{r_x, r_y, r_z\}$ are set separately to the marginal distances to the Kth nearest neighbour, and one then counts $\{n_z, n_{xz}, n_{yz}\}$ within or on these widths to obtain [29]:

$$I^{(2)}(X;Y \mid Z) = \psi(K) - \frac{2}{K} + \left\langle \psi(n_z) - \psi(n_{xz}) + \frac{1}{n_{xz}} - \psi(n_{yz}) + \frac{1}{n_{yz}} \right\rangle . \tag{65}$$

Local values for these estimators can be extracted by unrolling the expectation values and computing the nearest neighbour counts only at the given observation $\{x, y\}$, e.g. for algorithm 1 [86]:

$$i^{(1)}(x;y) = \psi(K) - \psi(n_x+1) - \psi(n_y+1) + \psi(N), \tag{66}$$

$$i^{(1)}(x;y \mid z) = \psi(K) + \psi(n_z+1) - \psi(n_{xz}+1) - \psi(n_{yz}+1). \tag{67}$$

This approach has been used to estimate local transfer entropy in [122] and [53].

KSG estimation builds on the non-linear and model-free capabilities of kernel estimation to add bias correction, better data efficiency and accuracy, and being effectively parameter-free (being relatively stable to choice of K). As such, it is widely-used as best of breed solution for MI, conditional MI and TE for continuous data (see e.g. [29, 87]). On the downside, it can be computationally expensive with naive algorithms requiring $\mathrm{O}\left(N^2\right)$ time (again ignoring the dimensionality of the data) though fast nearest neighbour search techniques can reduce this to $\mathrm{O}\left(N \log N\right)$. At the time of writing, JIDT only implements a naive algorithm here, though fast nearest neighbour search is a high-priority feature to add.

4. Permutation entropy and symbolic TE

Permutation entropy approaches [123] estimate the relevant PDFs based on the relative ordinal structure of the joint vectors (this is not suitable for PDFs of single dimensional variables). That is, for a joint variable $\mathbf{X}$ of d

dimensions, a sample $\mathbf{x}$ with components x_i ($i \in \{0 \dots d-1\}$) is replaced by an ordinal vector $\mathbf{o}$ with components $o_i \in \{0 \dots d-1\}$, where the value of $o_i = r$ assigned for x_i being the r-th largest component in $\mathbf{x}$. The PDF $p(\mathbf{x})$ is replaced by computation of $\hat{p}(\mathbf{o})$ for the corresponding ordinal vector, and these are used as plug-in estimates for the relevant average or local information-theoretic measure.

Permutation entropy has for example been adapted to estimate TE as the *symbolic transfer entropy* [124], with local symbolic transfer entropy also defined [58, 59].

Permutation approaches are computationally fast, since they effectively compute a discrete entropy after the ordinal symbolisation ($\mathrm{O}(N)$). They are a model-based approach however, assuming that all relevant information is in the ordinal relationship between the variables. This is not necessarily the case, and can lead to misleading results, as demonstrated in [89].

C. Related open-source information-theoretic toolkits

We next consider other existing open-source information-theoretic toolkits for computing the aforementioned measures empirically from time-series data. In particular we consider those which provide implementations of the transfer entropy. For each toolkit, we describe its purpose, the type of data it handles, and which measures and estimators are implemented.

TRENTOOL (http://www.trentool.de, GPL v3 license) [84] is a MATLAB toolbox which is arguably the most mature open-source toolkit for computing TE. It is not intended for general-purpose use, but designed from the ground up for transfer entropy analysis of (continuous) neural data, using the data format of the FieldTrip toolbox [125] for EEG, MEG and LFP recordings. In particular, it is designed for performing effective connectivity analysis between the input variables (see [34, 126]), including statistical significance testing of TE results (as outlined in Section II E) and processing steps to deal with volume conduction and identify cascade or common-driver effects in the inferred network. Conditional/multivariate TE is not yet available, but planned. TRENTOOL automates selection of parameters for embedding input time-series data and for source-target delays, and implements KSG estimation (see Section III B 3), harnessing fast nearest neighbour search, parallel computation and GPU-based algorithms [108].

The MuTE toolbox (available via figshare, CC-BY license) [127] provides MATLAB code for TE estimation. In particular, MuTE is capable of computing conditional TE, includes a number of estimator types (Gaussian, KSG, and discrete or binned), and adds non-uniform embedding (see [32]). It also adds code to assist with embedding parameter selection, and incorporates statistical significance testing.

Transfer entropy toolbox (TET, http://code.google.com/p/transfer-entropy-toolbox/, BSD license) [37] provides C-code callable from MATLAB for TE analysis of spiking data. TET is limited to binary (discrete) data only. Users can specify embedding dimension and source-target delay parameters.

MILCA (Mutual Information Least-dependent Component Analysis http://www.ucl.ac.uk/ion/departments/sobell/-Research/RLemon/MILCA/MILCA, GPL v3 license) provides C-code (callable from MATLAB) for mutual information calculations on continuous data [117, 128, 129]. MILCA's purpose is to use the MI calculations as part of Independent Component Analysis (ICA), but they can be accessed in a general-purpose fashion. MILCA implements KSG estimators with fast neighbour search; indeed, MILCA was co-written by the authors of this technique. It also handles multidimensional variables.

TIM (http://www.cs.tut.fi/%7etimhome/tim/tim.htm, GNU Lesser GPL license) [130] provides C++ code (callable from MATLAB) for general-purpose calculation of a wide range of information-theoretic measures on continuous-valued time-series, including for multidimensional variables. The measures implemented include entropy (Shannon, Renyi and Tsallis variants), Kullback-Leibler divergence, MI, conditional MI, TE and conditional TE. TIM includes various estimators for these, including Kozachenko-Leonenko (see Section III B 3), Nilsson-Kleijn [131] and Stowell-Plumbley [132] estimators for (differential) entropy, and KSG estimation for MI and conditional MI.

The MVGC (multivariate Granger causality toolbox, http://www.sussex.ac.uk/sackler/mvgc/, GPL v3 license) [133] provides a MATLAB implementation for general-purpose calculation of the Granger causality (i.e. TE with a linear-Gaussian model, see Section II B) on continuous data. MVGC also requires the MATLAB Statistics, Signal Processing and Control System Toolboxes.

There is a clear gap for a general-purpose information-theoretic toolkit, which can run in multiple code environments, implementing all of the measures in Section II A and Section II B, with various types of estimators, and with implementation of local values, measures of statistical significance etc. In the next section we introduce JIDT, and outline how it addresses this gap. Users should make a judicious choice of which toolkit suits their requirements, taking into account data types, estimators and application domain. For example, TRENTOOL is built from the ground up for effective network inference in neural imaging data, and is certainly the best tool for that application in comparison to a general-purpose toolkit.

TABLE I. Relevant web/wiki pages on JIDT website.

Name	URL
Project home	http://code.google.com/p/information-dynamics-toolkit/
Installation	http://code.google.com/p/information-dynamics-toolkit/wiki/Installation
Downloads	http://code.google.com/p/information-dynamics-toolkit/wiki/Downloads
MATLAB/Octave use	http://code.google.com/p/information-dynamics-toolkit/wiki/UseInOctaveMatlab
Octave-Java array conversion	http://code.google.com/p/information-dynamics-toolkit/wiki/OctaveJavaArrayConversion
Python use	http://code.google.com/p/information-dynamics-toolkit/wiki/UseInPython
JUnit test cases	http://code.google.com/p/information-dynamics-toolkit/wiki/JUnitTestCases
Demos	http://code.google.com/p/information-dynamics-toolkit/wiki/Demos
Simple Java Examples	http://code.google.com/p/information-dynamics-toolkit/wiki/SimpleJavaExamples
Octave/MATLAB Examples	http://code.google.com/p/information-dynamics-toolkit/wiki/OctaveMatlabExamples
Python Examples	http://code.google.com/p/information-dynamics-toolkit/wiki/PythonExamples
Cellular Automata demos	http://code.google.com/p/information-dynamics-toolkit/wiki/CellularAutomataDemos
Schreiber TE Demos	http://code.google.com/p/information-dynamics-toolkit/wiki/SchreiberTeDemos
jidt-discuss group	http://groups.google.com/group/jidt-discuss
SVN URL	http://information-dynamics-toolkit.googlecode.com/svn/trunk/

IV. JIDT INSTALLATION, DESIGN AND DEMONSTRATIONS

JIDT (Java Information Dynamics Toolkit, http://code.google.com/p/information-dynamics-toolkit/, GPL v3 license) is unique as a general-purpose information-theoretic toolkit which provides all of the following features in one package:

- Implementation of a large array of measures, including all conditional/multivariate forms of the transfer entropy, complementary measures such as active information storage, and allows full specification of relevant embedding parameters;
- Implementation a wide variety of estimator types and applicability to both discrete and continuous data;
- Implementation of local measurement for all estimators;
- Inclusion of statistical significance calculations for MI, TE, etc. and their conditional variants;
- No dependencies on other installations (except Java).

Furthermore, JIDT is written in Java, taking advantage of the following features:

- The code becomes platform agnostic, requiring only an installation of the Java Virtual Machine (JVM) to run;
- The code is object-oriented, with common code shared and an intuitive hierarchical design using interfaces; this provides flexibility and allows different estimators of same measure can be swapped dynamically using polymorphism;
- The code can be called directly from MATLAB, GNU Octave and Python, but runs faster than native code in those languages (still slower but comparable to C/C++ [134]); and
- Automatic Javadoc generation for each class.

In the following, we describe the (minimal) installation process in Section IV A, and contents of the version 1.0 JIDT distribution in Section IV B. We then describe which estimators are implemented for each measure in Section IV C, and architecture of the source code in Section IV D. We also outline how the code has been tested in Section IV E, how to build it (if required) in Section IV F and point to other sources of documentation in Section IV G.

A. Installation and dependencies

There is **little to no installation** of JIDT required beyond downloading the software. The software can be run on any platform which supports a standard edition Java Runtime Environment (i.e. Windows, Mac, Linux, Solaris).

Material pertaining to installation is described in full at the "Installation" wiki page for the project (*see Table I for all relevant project URLs*); summarised as follows:

1. Download a code release package from the "Downloads" wiki page. Full distribution is recommended (described in Section IV B) so as to obtain e.g. access to the examples described in Section V, though a "Jar only" distribution provides just the JIDT library `infodynamics.jar` in Java archive file format.

2. Unzip the distribution to the location of your choice, and/or move the `infodynamics.jar` file to a relevant location. Ensure that `infodynamics.jar` is on the Java classpath when your code attempts to access it (see Section V).

3. To update to a new version, simply copy the new distribution over the top of the previous one.

As an alternative, advanced users can take an SVN checkout of the source tree from the SVN URL (see Table I) and build the `infodynamics.jar` file using `ant` scripts (see Section IV F).

In general, there are no dependencies that a user would need to download in order to run the code. Some exceptions are as follows:

1. Java must be installed on your system in order to run JIDT; most systems will have Java already installed. To simply run JIDT, you will only need a Java Runtime Environment (JRE, also known as Java Virtual Machine or JVM), whereas to modify and/or build to software, or write your own Java code to access it, you will need the full Java Development Kit (JDK), standard edition (SE). Download it from http://java.com/. For using JIDT via MATLAB, a JVM is included in MATLAB already.

2. If you wish to build the project using the `build.xml` script – this requires `ant` (see Section IV F).

3. If you wish to run the unit test cases (see Section IV E) - this requires the `JUnit` framework: http://www.junit.org/ - for how to run `JUnit` with our ant script see "JUnit test cases" wiki page.

4. Additional preparation may be required to use JIDT in GNU Octave or Python. Octave users must install the `octave-java` package from the `Octave-forge` project – see description of these steps at "MATLAB/Octave use" wiki page. Python users must install a relevant Python-Java extension – see description at "Python use" wiki page. Both cases will depend on a JVM on the system (as per point 1 above), though the aforementioned extensions may install this for you.

Note that JIDT does *adapt* code from a number of sources in accordance with their open-source license terms, including: Apache Commons Math v3.3 (http://commons.apache.org/proper/commons-math/), the JAMA project (http://math.nist.gov/javanumerics/jama/), and the octave-java package from the Octave-Forge project (http://octave.-sourceforge.net/java/). Relevant notices are supplied in the `notices` folder of the distribution. Such code is included in JIDT however and does not need to be installed separately.

B. Contents of distribution

The contents of the current (version 1.0) JIDT (full) distribution are as follows:

- The top level folder contains the `infodynamics.jar` library file, a GNU GPL v3 license, a `readme.txt` file and an ant `build.xml` script for (re-)building the code (see Section IV F);

- The `java` folder contains source code for the library in the `source` subfolder (described in Section IV C), and unit tests in the `unittests` subfolder (see Section IV E).

- The `javadocs` folder contains automatically generated Javadocs from the source code, as discussed in Section IV G.

- The `demos` folder contains several example applications of the software, described in Section V, sorted into folders to indicate which environment they are intended to run in, i.e. `java`, `octave` (which is compatible with `MATLAB`) and `python`. There is also a `data` folder here containing sample data sets for these demos and unit tests.

- The `notices` folder contains notices and licenses pertaining to derivations of other open source code used in this project.

TABLE II. An outline of which estimation techniques are implemented for each relevant information-theoretic measure. The ✓symbol indicates that the measure is implemented for the given estimator and is applicable to *both* univariate and multivariate arguments (e.g. collective transfer entropy, where the source is multivariate), while the addition of superscript 'u' (i.e. $\checkmark^u$) indicates the measure is implemented for univariate arguments only. The equation numbers refer to the definitions for the average and local values (where provided) for each measure. Also, while the KSG estimator is not applicable for entropy, the ⋆ symbol there indicates the implementation of Kozachenko-Leonenko estimator [119] for entropy (which the KSG technique is based on for MI; see Section III B 3).

Measure			Discrete	Continuous estimators			
			estimator	Gaussian	Box-Kernel	Kraskov *et al.*(KSG)	Permutation
Name	Notation	Defined at	§III A	§III B 1	§III B 2	§III B 3	§III B 4
Entropy	$H(X)$	Eq. (1,32)	✓	✓	✓	⋆	
Entropy rate	$H_{\mu X}$	Eq. (14,46)	✓	*Use two multivariate entropy calculators*			
Mutual information (MI)	$I(X;Y)$	Eq. (6,40)	✓	✓	✓	✓	
Conditional MI	$I(X;Y \mid Z)$	Eq. (11,43)	✓	✓		✓	
Multi-information	$I(\mathbf{X})$	Eq. (7)	✓		$\checkmark^u$	$\checkmark^u$	
Transfer entropy (TE)	$T_{Y \to X}$	Eq. (25,52)	✓	✓	✓	✓	$\checkmark^u$
Conditional TE	$T_{Y \to X \mid Z}$	Eq. (29,54)	✓	$\checkmark^u$		$\checkmark^u$	
Active information storage	A_X	Eq. (23,50)	✓	$\checkmark^u$	$\checkmark^u$	$\checkmark^u$	
Predictive information	E_X	Eq. (21,48)	✓	*Use a multivariate MI calculator*			
Separable information	S_X	Eq. (31)	✓				

C. Source code and estimators implemented

The Java source code for the JIDT library contained in the `java/source` folder is organised into the following Java *packages* (which map directly to subdirectories):

- `infodynamics.measures` contains all of the classes implementing the information-theoretic measures, split into:
  - `infodynamics.measures.discrete` containing all of the measures for discrete data;
  - `infodynamics.measures.continuous` which at the top level contains Java *interfaces* for each of the measures as applied to continuous data, then a set of sub-packages (`gaussian`, `kernel`, `kozachenko`, `kraskov` and `symbolic`) which map to each estimator type in Section III and contain *implementations* of such estimators for the interfaces defined for each measure (Section IV D describes the object-oriented design used here). Table II identifies which estimators are measured for each estimator type;
  - `infodynamics.measures.mixed` includes *experimental* discrete-to-continuous MI calculators, though these are not discussed in detail here.
- `infodynamics.utils` contains classes providing a large number of utility functions for the measures (e.g. matrix manipulation, file reading/writing including in Octave text format);
- `infodynamics.networkinference` contains implementations of higher-level algorithms which use the information-theoretic calculators to infer an effective network structure from time-series data (see Section V F).

As outlined above, Table II describes which estimators are implemented for each measure. This effectively maps the definitions of the measures in Section II to the estimators in Section III. All estimators provide the corresponding *local* information-theoretic measures (as introduced in Section II C). Also, for the most part, the estimators include a generalisation to multivariate $\mathbf{X}$, $\mathbf{Y}$, etc, as identified in the table.

D. JIDT architecture

The measures for continuous data have been organised in a strongly *object-oriented* fashion.[15] Fig. IV D provides a sample (partial) Unified Modeling Language (UML) class diagram of the implementations of the conditional mutual

[15] This is also the case for the measures for discrete data, though to a lesser degree and without multiple estimator types, so this is not focussed on here.

information (Eq. (11)) and transfer entropy (Eq. (25)) measures using KSG estimators (Section III B 3). This diagram shows the typical object-oriented hierarchical structure of the implementations of various estimators for each measure. The class hierarchy is organised as follows.

Interfaces at the top layer define the available *methods* for each measure. At the top of this figure we see the `ConditionalMutualInfoCalculatorMultiVariate` and `TransferEntropyCalculator` interfaces which define the methods each estimator *class* for a given measure must implement. Such interfaces are defined for each information-theoretic measure in the `infodynamics.measures.continuous` package.

Abstract classes[16] at the intermediate layer provide basic functionality for each measure. Here, we have abstract classes `ConditionalMutualInfoMultiVariateCommon` and `TransferEntropyCalculatorViaCondMutualInfo` which *implement* the above interfaces, providing common code bases for the given measures that various child classes can build on to specialise themselves to a particular estimator type. For instance, the `TransferEntropyCalculatorViaCond-MutualInfo` class provides code which abstractly *uses* a `ConditionalMutualInfoCalculatorMultiVariate` interface in order to make transfer entropy calculations, but does not concretely specify which type of conditional MI estimator to use, nor fully set its parameters.

Child classes at the lower layers add specialised functionality for each estimator type for each measure. These child classes *inherit* from the above parent classes, building on the common code base to add specialisation code for the given estimator type. Here that is the KSG estimator type. The child classes at the bottom of the hierarchy have no remaining abstract functionality, and can thus be used to make the appropriate information-theoretic calculation. We see that `ConditionalMutualInfoCalculatorMultiVariateKraskov` begins to *specialise* `ConditionalMutualInfoMultiVariateCommon` for KSG estimation, with further specialisation by its child class `ConditionalMutualInfoCalculatorMultiVariateKraskov1` which implements the KSG algorithm 1 (Eq. (62)). Not shown here is `ConditionalMutualInfoCalculatorMultiVariateKraskov2` which implements the KSG algorithm 2 (Eq. (63)) and has similar class relationships. We also see that `TransferEntropyCalculatorKraskov` specialises `TransferEntropyCalculatorViaCondMutualInfo` for KSG estimation, by using `ConditionalMutualInfoCalculator-MultiVariateKraskov1` (or `ConditionalMutualInfoCalculatorMultiVariateKraskov2`, not shown) as the specific implementation of `ConditionalMutualInfoCalculatorMultiVariate`. The implementations of these interfaces for other estimator types (e.g. `TransferEntropyCalculatorGaussian`) sit at the same level here inheriting from the common abstract classes above.

This type of object-oriented hierarchical structure delivers two important benefits: i. the *decoupling* of common code away from specific estimator types and into common parent classes allows code re-use and simpler maintenance, and ii. the use of interfaces delivers *subtype polymorphism* allowing *dynamic dispatch*, meaning that one can write code to compute a given measure using the methods on its interface and only specify the estimator type at runtime (see demo Section V A 8).

E. Validation

The calculators in JIDT are validated using a set of *unit tests* (distributed in the `java/unittests` folder). Unit testing is a method of testing software by the use of a set of small test cases which call parts of the code and check the output against expected values, flagging errors if they arise. The unit tests in JIDT are implemented via the `JUnit` framework version 3 (http://www.junit.org/). They can be run via the ant script (see Section IV F).

At a high level, the unit tests include validation of the results of information-theoretic calculations applied to the sample data in `demos/data` against measurements from various other existing toolkits, e.g.:

- The KSG estimator (Section III B 3) for MI is validated against values produced from the *MILCA* toolkit [117, 128, 129];
- The KSG estimator for conditional MI and TE is validated against values produced from scripts within *TRENTOOL* [84];
- The discrete and box-kernel estimators for TE are validated against the plots in Schreiber's original paper [22] on TE (see Section V D);
- The Gaussian estimator for TE (Section III B 1) is verified against values produced from (a modified version of) the *computeGranger.m* script of the *ChaLearn Connectomics Challenge Sample Code* [135].

[16] Abstract classes provide implementations of some but not all methods required for a class, so they cannot be directly instantiated themselves but child classes which provide implementations for the missing methods and may be instantiated.

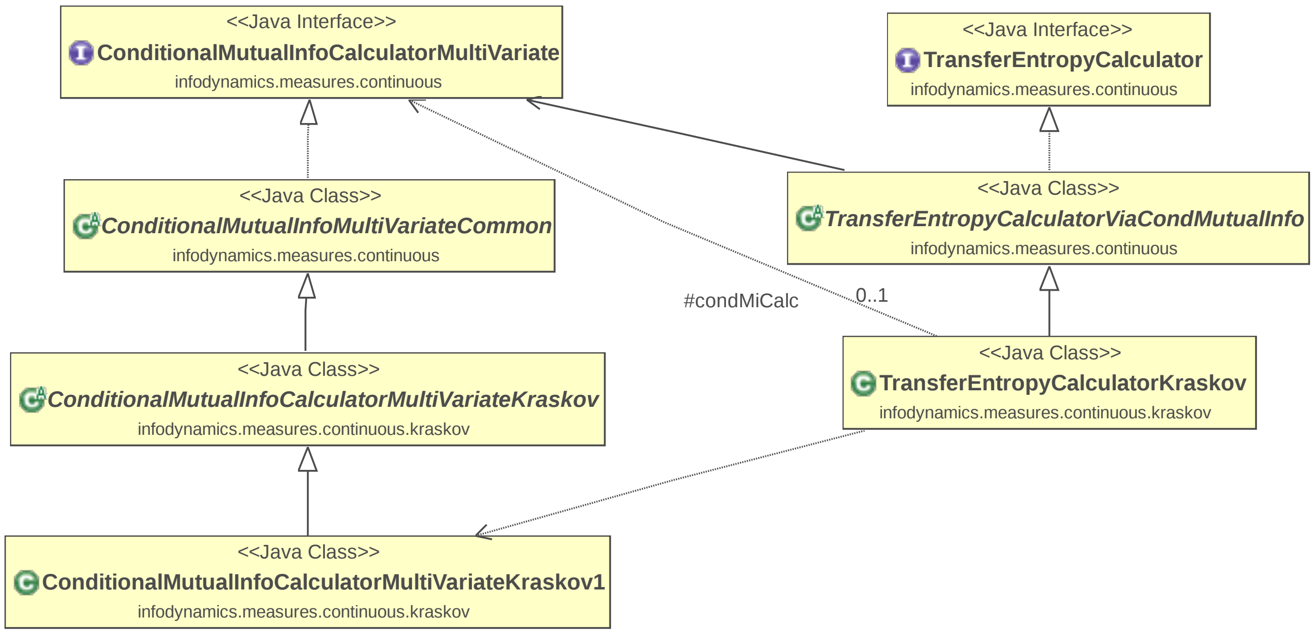
<<Java Interface>>
ConditionalMutualInfoCalculatorMultiVariate
infodynamics.measures.continuous
<<Java Interface>>
TransferEntropyCalculator
infodynamics.measures.continuous
<<Java Class>>
ConditionalMutualInfoMultiVariateCommon
infodynamics.measures.continuous
<<Java Class>>
TransferEntropyCalculatorViaCondMutualInfo
infodynamics.measures.continuous
#condMiCalc
0..1
<<Java Class>>
ConditionalMutualInfoCalculatorMultiVariateKraskov
infodynamics.measures.continuous.kraskov
<<Java Class>>
TransferEntropyCalculatorKraskov
infodynamics.measures.continuous.kraskov
<<Java Class>>
ConditionalMutualInfoCalculatorMultiVariateKraskov1
infodynamics.measures.continuous.kraskov

FIG. 2. Partial UML class diagram of the implementations of the conditional mutual information (Eq. (11)) and transfer entropy (Eq. (25)) measures using KSG estimators. As explained in the main text, this diagram shows the typical *object-oriented* structure of the implementations of various estimators for each measure. The relationships indicated on the class diagram are as follows: dotted lines with hollow triangular arrow heads indicate the realisation or *implementation* of an interface by a class; solid lines with hollow triangular arrow heads indicate the generalisation or *inheritance* of a child or subtype from a parent or superclass; lines with plain arrow heads indicate that one class *uses* another (with the solid line indicating direct usage and dotted line indicating indirect usage via the superclass).

Further code coverage by the unit tests is planned in future work.

F. (Re-)building the code

Users may wish to build the code, perhaps if they are directly accessing the source files via `svn` or modifying the files. The source code may be compiled manually of course, or in your favourite IDE (Integrated Development Environment), however JIDT provides an `ant` build script, `build.xml`, to streamline this process. Apache `ant` – see http://ant.apache.org/ – is a command-line tool to build various interdependent targets in software projects, much like the older style `Makefile` for C/C++.

To build any of the following targets using `build.xml`, run `ant <targetName>` in the top-level directory of the distribution, where `<targetName>` may be any of the following:

- `build` or `jar` (this is the default if no `<targetName>` is supplied) – creates a jar file for the JIDT library;
- `compile` – compiles the JIDT library and unit tests;
- `junit` – runs the JIDT library and unit tests;
- `javadocs` – generates automated Javadocs from the formatted comments in the source code;
- `jardist` – packages the JIDT jar file in a distributable form, as per the jar-only distributions of the project;
- `dist` – runs unit tests, and packages the JIDT jar file, Javadocs, demos, etc in a distributable form, as per the full distributions of the project;
- `clean` – delete all compiled code etc built by the above commands.

G. Documentation and support

Documentation to guide users of JIDT is composed of:

1. This manuscript!
2. The Javadocs contained in the `javadocs` folder of the distribution (main page is `index.html`). Javadocs are html formatted documentation for each package, class and interface in the library, which are automatically created from formatted comments in the source code. The Javadocs are very useful tools for users, since they provide specific details about each class and their methods, in more depth than we are able to do here; for example which properties may be set for each class. The Javadocs can be (re-)generated using ant as described in Section IV F.
3. The demos; as described further in Section V, on the demos wiki page (see Table I), and the individual wiki page for each demo;
4. The project wiki pages (accessed from the project home page, see Table I) provide additional information on various features, e.g. how to use JIDT in MATLAB or Octave and Python;
5. The unit tests (as described in Section IV E provide additional examples on how to run the code.

You can also join our email discussion group `jidt-discuss` on Google Groups (see URL in Table I) or browse past messages, for announcements, asking questions, etc.

V. CODE DEMONSTRATIONS

In this section, we describe some simple demonstrations on how to use the JIDT library. Several sets of demonstrations are included in the JIDT distribution, some of which are described here. More detail is provided for each demo on its wiki page, accessible from the main Demos wiki page (see Table I). We begin with the main set of *Simple Java Demos*, focussing in particular on a detailed walk-through of using a KSG estimator to compute transfer entropy since the calling pattern here is typical of all estimators for continuous data. Subsequently, we provide more brief overviews of other examples available in the distribution, including how to run the code in MATLAB, GNU Octave and Python, implementing the transfer entropy examples from Schreiber's original paper [22], and computing spatiotemporal profiles of information dynamics in Cellular Automata.
; for

A. Simple Java Demos

The primary set of demos is the "Simple Java Demos" set at `demos/java` in the distribution. This set contains eight standalone Java programs to demonstrate simple use of various aspects of the toolkit. This set is described further at the `SimpleJavaExamples` wiki page (see Table I).

The Java source code for each program is located at `demos/java/infodynamics/demos` in the JIDT distribution, and shell scripts to run each program are found at `demos/java/`. The shell scripts demonstrate how to compile and run the programs from command line, e.g. `example1TeBinaryData.sh` contains the following commands:

```
# Make sure the latest source file is compiled.
javac -classpath "../../infodynamics.jar" "infodynamics/demos/Example1TeBinaryData.java"
# Run the example:
java -classpath ".:../../infodynamics.jar" infodynamics.demos.Example1TeBinaryData
```

Listing 1. Shell script example1TeBinaryData.sh.

The examples focus on various transfer entropy estimators (though similar calling paradigms can be applied to all estimators), including:

1. computing transfer entropy on *binary (discrete) data*;
2. computing transfer entropy for specific channels within *multidimensional* binary data;
3. computing transfer entropy on *continuous* data using *kernel estimation*;

4. computing transfer entropy on *continuous* data using *KSG estimation*;
5. computing *multivariate* transfer entropy on *multidimensional* binary data;
6. computing *mutual information* calculator on continuous data, using *dynamic dispatch* or *late-binding* to a particular estimator;
7. computing transfer entropy from an ensemble of time-series samples;
8. computing transfer entropy on *continuous* data using *binning* then discrete calculation.

In the following, we explore selected salient examples in this set. We begin with `Example1TeBinaryData.java` and `Example4TeContinuousDataKraskov.java` as typical calling patterns to use estimators for discrete and continuous data respectively, then add extensions for how to compute local measures and statistical significance, use ensembles of samples, handle multivariate data and measures, and dynamic dispatch.

1. *Typical calling pattern for an information-theoretic measure on discrete data*

`Example1TeBinaryData.java` (see Listing 2) provides a typical calling pattern for calculators for **discrete data**, using the `infodynamics.measures.discrete.TransferEntropyCalculatorDiscrete` class. *While the specifics of some methods may be slightly different, the general calling paradigm is the same for all discrete calculators.*

```
1  int arrayLengths = 100;
2  RandomGenerator rg = new RandomGenerator();
3  // Generate some random binary data:
4  int[] sourceArray = rg.generateRandomInts(arrayLengths, 2);
5  int[] destArray = new int[arrayLengths];
6  destArray[0] = 0;
7  System.arraycopy(sourceArray, 0, destArray, 1, arrayLengths - 1);
8  // Create a TE calculator and run it:
9  TransferEntropyCalculatorDiscrete teCalc = new TransferEntropyCalculatorDiscrete(2, 1);
10 teCalc.initialise();
11 teCalc.addObservations(sourceArray, destArray);
12 double result = teCalc.computeAverageLocalOfObservations();
```

Listing 2. Estimation of TE from discrete data; source code adapted from `Example1TeBinaryData.java`.

The data type used for all discrete data are `int[]` time-series arrays (indexed by time). Here we are computing TE for **univariate time series** data, so `sourceArray` and `destArray` at line 4 and line 5 are single dimensional `int[]` arrays. Multidimensional time series are discussed in Section V A 6.

The first step in using any of the estimators is to construct an instance of them, as per line 9 above. Parameters/properties for calculators for discrete data are *only* supplied in the constructor at line 9 (this is not the case for continuous estimators, see Section V A 2). See the Javadocs for each calculator for descriptions of which parameters can be supplied in their constructor. The arguments for the TE constructor here include: the number of discrete values ($M = 2$), which means the data can take values $\{0, 1\}$ (the allowable values are always enumerated $0, \ldots, M - 1$); and the embedded history length $k = 1$. Note that for measures such as TE and AIS which require embeddings of time-series variables, the user must provide the embedding parameters here.

All calculators must be initialised before use or re-use on new data, as per the call to `initialise()` at line 10. This call clears any PDFs held inside the class, The `initialise()` method provides a mechanism by which the same object instance may be used to make separate calculations on multiple data sets, by calling it in between each application (i.e. looping from line 12 back to line 10 for a different data set – see the full code for `Example1TeBinaryData.java` for an example).

The user then supplies the data to construct the PDFs with which the information-theoretic calculation is to be made. Here, this occurs at line 11 by calling the `addObservations()` method to supply the source and destination time series values. This method can be called multiple times to add multiple sample time-series before the calculation is made (see further commentary for handling ensembles of samples in Section V A 5).

Finally, with all observations supplied to the estimator, the resulting transfer entropy may be computed via `computeAverageLocalOfObservations()` at line 12. The information-theoretic *measurement is returned in bits* for all discrete calculators. In this example, since the destination copies the previous value of the (randomised) source, then `result` should approach 1 bit.

2. *Typical calling pattern for an information-theoretic measure on continuous data*

Before outlining how to use the continuous estimators, we note that the discrete estimators above may be applied to continuous `double[]` data sets by first *binning* them to convert them to `int[]` arrays, using either `MatrixUtils.discretise(double data[], int numBins)` for even bin sizes or `MatrixUtils.discretiseMaxEntropy( double data[], int numBins)` for maximum entropy binning (see `Example8TeContinuousDataByBinning`). This is very efficient, however as per Section III B it is more accurate to use an estimator which utilises the continuous nature of the data.

As such, we now review the use of a KSG estimator (Section III B 3) to compute transfer entropy (Eq. (25)), as a *standard calling pattern* for all estimators applied to *continuous data*. The sample code in Listing 3 is adapted from `Example4TeContinuousDataKraskov.java`. (That this is a standard calling pattern can easily be seen by comparing to `Example3TeContinuousDataKernel.java`, which uses a box-kernel estimator but has very similar method calls, except for which parameters are passed in).

```
1 double[] sourceArray, destArray;
2 // ...
3 // Import values into sourceArray and destArray
4 // ...
5 TransferEntropyCalculatorKraskov teCalc = new TransferEntropyCalculatorKraskov();
6 teCalc.setProperty("k", "4");
7 teCalc.initialise(1);
8 teCalc.setObservations(sourceArray, destArray);
9 double result = teCalc.computeAverageLocalOfObservations();
```

Listing 3. Use of KSG estimator to compute transfer entropy; adapted from `Example4TeContinuousDataKraskov.java`

Notice that the calling pattern here is almost the same as that for discrete calculators, as seen in Listing 2, with some minor differences outlined below.

Of course, for continuous data we now use `double[]` arrays (indexed by time) for the univariate time-series data here at line 1. Multidimensional time series are discussed in Section V A 7.

As per discrete calculators, we begin by constructing an instance of the calculator, as per line 5 above. Here however, parameters for the operation of the estimator are not only supplied via the constructor (see below). As such, all classes offer a constructor with no arguments, while only some implement constructors which accept certain parameters for the operation of the estimator.

Next, almost all relevant properties or parameters of the estimators can be supplied by passing key-value pairs of `String` objects to the `setProperty(String, String)` method at line 6. The key values for properties which may be set for any given calculator are described in the Javadocs for the `setProperty` method for each calculator. Properties for the estimator may be set by calling `setProperty` at any time; in most cases the new property value will take effect immediately, though it is only guaranteed to hold after the next initialisation (see below). At line 6, we see that property `"k"` (shorthand for `ConditionalMutualInfoCalculatorMultiVariateKraskov.PROP_K`) is set to the value `"4"`. As described in the Javadocs for `TransferEntropyCalculatorKraskov.setProperty`, this sets the number of nearest neighbours K to use in the KSG algorithm in the full joint space. Properties can also easily be extracted and set from a file, see `Example6LateBindingMutualInfo.java`.

As per the discrete calculators, all continuous calculators must be initialised before use or re-use on new data (see line 7). This clears any PDFs held inside the class, but additionally finalises any property settings here. Also, the `initialise()` method for continuous estimators may accept some parameters for the calculator – here it accepts a setting for the k embedded history length parameter for the transfer entropy (see Eq. (25)). Indeed, there may be several *overloaded* forms of `initialise()` for a given class, each accepting different sets of parameters. For example, the `TransferEntropyCalculatorKraskov` used above offers an `initialise(k, tau_k, l, tau_l, u)` method taking arguments for both source and target embedding lengths k and l, embedding delays τ_k and τ_l (see Section II B), and source-target delay u (see Eq. (27)). Note that currently such embedding parameters must be supplied by the user, although we intend to implement automated embedding parameter selection in the future. Where a parameter is not supplied, the value given for it in a previous call to `initialise()` or `setProperty()` (or otherwise its default value) is used.

The supply of samples is also subtly different for continuous estimators. Primarily, all estimators offer the `setObservations()` method (line 8) for supplying a single time-series of samples (which can only be done once). See Section V A 5 for how to use multiple time-series realisations to construct the PDFs via an `addObservations()` method.

Finally, the information-theoretic measurement (line 9) is returned in *either bits or nats* as per the standard definition for this type of estimator in Section III (i.e. bits for discrete, kernel and permutation estimators; nats for Gaussian and KSG estimators).

At this point (before or after line 9) once all observations have been supplied, there are other quantities that the user may compute. These are described in the next two subsections.

3. Local information-theoretic measures

Listing 4 computes the *local* transfer entropy (Eq. (52)) for the observations supplied earlier in Listing 3:

```
10 double[] localTE = teCalc.computeLocalOfPreviousObservations();
```

Listing 4. Computing local measures after Listing 3; adapted from `Example4TeContinuousDataKraskov.java`.

Each calculator (discrete or continuous) provides a `computeLocalOfPreviousObservations()` method to compute the relevant local quantities for the given measure (see Section II C). This method returns a `double[]` array of the local values (local TE here) at every time step n for the supplied time-series observations. For TE estimators, note that the first k values (history embedding length) will have value zero, since local TE is not defined without the requisite history being available.[17]

4. Null distribution and statistical significance

For the observations supplied earlier in Listing 3, Listing 5 computes a distribution of surrogate TE values obtained via bootstrapping under the null hypothesis that `sourceArray` and `destArray` have no temporal relationship (as described in Section II E).

```
11 EmpiricalMeasurementDistribution dist = teCalc.computeSignificance(1000);
```

Listing 5. Computing null distribution after Listing 3; adapted from `Example3TeContinuousDataKernel.java`.

The method `computeSignificance()` is implemented for all MI and conditional MI based measures (including TE), for both discrete and continuous estimators. It returns an `EmpiricalMeasurementDistribution` object, which contains a `double[]` array `distribution` of an empirical distribution of values obtained under the null hypothesis (the sample size for this distribution is specified by the argument to `computeSignificance()`). The user can access the mean and standard deviation of the distribution, a p-value of whether these surrogate measurements were greater than the actual TE value for the supplied source, and a corresponding t-score (which assumes a Gaussian distribution of surrogate scores) via method calls on this object (see Javadocs for details).

Some calculators (discrete and Gaussian) overload the method `computeSignificance()` (without an input argument) to return an object encapsulating an *analytically* determined p-value of surrogate distribution where this is possible for the given estimation type (see Section II E). The availability of this method is indicated when the calculator implements the `AnalyticNullDistributionComputer` interface.

5. Ensemble approach: using multiple trials or realisations to construct PDFs

Now, the use of `setObservations()` for continuous estimators implies that the PDFs are computed from a single *stationary* time-series realisation. One may supply multiple time-series realisations (e.g. as multiple stationary trials from a brain-imaging experiment) via the following alternative calling pattern to line 8 in Listing 3:

```
8.a teCalc.startAddObservations();
8.b teCalc.addObservations(sourceArray1, destArray1);
8.c teCalc.addObservations(sourceArray2, destArray2);
8.d teCalc.addObservations(sourceArray3, destArray3);
8.e // ...
8.f teCalc.finaliseAddObservations();
```

Listing 6. Supply of multiple time-series realisations as observations for the PDFs; an alternative to line 8 in Listing 3. Code is adapted from `Example7EnsembleMethodTeContinuousDataKraskov.java`

[17] The return format is more complicated if the user has supplied observations via several `addObservations()` calls rather than `setObservations()`; see the Javadocs for `computeLocalOfPreviousObservations()` for details.

Computations on the PDFs constructed from this data can then follow as before. Note that other variants of `addObservations()` exist, e.g. which pull out sub-sequences from the time series arguments; see the Javadocs for each calculator to see the options available. Also, for the discrete estimators, `addObservations()` may be called multiple times directly without the use of a `startAddObservations()` or `finaliseAddObservations()` method. This type of calling pattern may be used to realise an *ensemble approach* to constructing the PDFs (see [29, 84, 108, 121]), in particular by supplying only short corresponding (*stationary*) parts of each trial to generate the PDFs for that section of an experiment.

6. *Joint-variable measures on multivariate discrete data*

For calculations involving **joint variables** from **multivariate discrete data** time-series (e.g. collective transfer entropy, see Section II B), we use the *same* discrete calculators (unlike the case for continuous-valued data in Section V A 7). This is achieved with one simple pre-processing step, as demonstrated by `Example5TeBinaryMultivarTransfer.-java`:

```
1  int[][] source, dest;
2  // ...
3  // Import binary values into the arrays,
4  //  with two columns each.
5  // ...
6  TransferEntropyCalculatorDiscrete teCalc = new TransferEntropyCalculatorDiscrete(4, 1);
7  teCalc.initialise();
8  teCalc.addObservations(
9     MatrixUtils.computeCombinedValues(source, 2),
10    MatrixUtils.computeCombinedValues(dest, 2));
11 double result = teCalc.computeAverageLocalOfObservations();
```

Listing 7. Java source code adapted from `Example5TeBinaryMultivarTransfer.java`.

We see that the multivariate discrete data is represented using two-dimensional `int[][]` arrays at line 1, where the first array index (row) is time and the second (column) is variable number.

The important pre-processing at line 9 and line 10 involves combining the joint vector of discrete values for each variable at each time step into a single discrete number; i.e. if our joint vector `source[t]` at time t has v variables, each with M possible discrete values, then we can consider the joint vector as a v-digit base-M number, and directly convert this into its decimal equivalent. The `computeCombinedValues()` utility in `infodynamics.utils.MatrixUtils` performs this task for us at each time step, taking the `int[][]` array and the number of possible discrete values for each variable $M = 2$ as arguments. Note also that when the calculator was constructed at line 6, we need to account for the total number of possible combined discrete values, being $M^v = 4$ here.

7. *Joint-variable measures on multivariate continuous data*

For calculations involving **joint variables** from **multivariate continuous data** time-series, JIDT provides *separate* calculators to be used. `Example6LateBindingMutualInfo.java` demonstrates this for calculators implementing the `MutualInfoCalculatorMultiVariate` interface:[18]

```
double[][] variable1, variable2;
MutualInfoCalculatorMultiVariate miCalc;
// ...
// Import continuous values into the arrays
//  and instantiate miCalc
// ...
miCalc.initialise(2, 2);
miCalc.setObservations(variable1, variable2);
double miValue = miCalc.computeAverageLocalOfObservations();
```

Listing 8. Java source code adapted from `Example6LateBindingMutualInfo.java`.

[18] In fact, for MI, JIDT does not actually define a calculator for univariates – the multivariate calculator `MutualInfoCalculatorMultiVariate` provides interfaces to supply univariate `double[]` data where each variable is univariate.

First, we see that the multivariate continuous data is represented using two-dimensional `double[][]` arrays at line 1, where (as per Section V A 6) the first array index (row) is time and the second (column) is variable number. The instantiating of a class implementing the `MutualInfoCalculatorMultiVariate` interface to make the calculations is not shown here (but is discussed separately in Section V A 8).

Now, a crucial step in using the multivariate calculators is specifying in the arguments to `initialise()` the number of dimensions (i.e. the number of variables or columns) for each variable involved in the calculation. At line 7 we see that each variable in the MI calculation has two dimensions (i.e. there will be two columns in each of `variable1` and `variable2`).

Otherwise, other interactions with these multivariate calculators follow the same form as for the univariate calculators.

8. Coding to interfaces; or dynamic dispatch

Listing 8 (`Example6LateBindingMutualInfo.java`) also demonstrates the manner in which a user can code to the interfaces defined in `infodynamics.measures.continuous`, and dynamically alter the instantiated class implementing this interface at runtime. This is known as *dynamic dispatch*, enabled by the *polymorphism* provided by the interface (described at Section IV D). This is a useful feature in object-oriented programming where, here, a user wishes to write code which requires a particular measure, and dynamically switch-in different estimators for that measure at runtime. For example, in Listing 8 we may normally use a KSG estimator, but switch-in a linear-Gaussian estimator if we happen to know our data is Gaussian.

To use dynamic dispatch with JIDT:

1. Write code to use an interface for a calculator (e.g. `MutualInfoCalculatorMultiVariate` in Listing 8), rather than to directly use a particular implementing class (e.g. `MutualInfoCalculatorMultiVariateKraskov`);

2. Instantiate the calculator object by *dynamically* specifying the implementing class (compare to the static instantiation at line 5 of Listing 3), e.g. using a variable name for the class as shown in Listing 9:

```
String implementingClass;
// Load the name of the class to be used into the
//   variable implementingClass
miCalc = (MutualInfoCalculatorMultiVariate) Class.forName(implementingClass).newInstance();
```

Listing 9. Dynamic instantiation of a mutual information calculator, belonging at line 5 in Listing 8. Adapted from `Example6LateBindingMutualInfo.java`.

Of course, to be truly dynamic, the value of `implementingClass` should not be hard-coded but must be somehow set by the user. For example, in the full `Example6LateBindingMutualInfo.java` it is set from a properties file.

B. MATLAB / Octave Demos

The "Octave/MATLAB code examples" set at `demos/octave` in the distribution provide a basic set of demonstration scripts for using the toolkit in GNU Octave or MATLAB. The set is described in some detail at the `OctaveMatlabExamples` wiki page (Table I). See Section IV A regarding installation requirements for running the toolkit in Octave, with more details at the `UseInOctaveMatlab` wiki page (see Table I).

The scripts in this set mirror the Java code in the "Simple Java Demos" set (Section V A), to demonstrate that *anything which JIDT can do in a Java environment can also be done in MATLAB/Octave.* The user is referred to the distribution or the `OctaveMatlabExamples` wiki page for more details on the examples. An illustrative example is provided in Listing 10, which converts Listing 2 into MATLAB/Octave:

```
javaaddpath('../../infodynamics.jar');
sourceArray=(rand(100,1)>0.5)*1;
destArray = [0; sourceArray(1:99)];
teCalc=javaObject('infodynamics.measures.discrete.TransferEntropyCalculatorDiscrete', 2, 1);
teCalc.initialise();
teCalc.addObservations(
    octaveToJavaIntArray(sourceArray),
    octaveToJavaIntArray(destArray));
result = teCalc.computeAverageLocalOfObservations()
```

Listing 10. Estimation of TE from discrete data in MATLAB/Octave; adapted from `example1TeBinaryData.m`

This example illustrates several important steps for using JIDT from a MATLAB/Octave environment:

1. Specify the classpath (i.e. the location of the `infodynamics.jar` library) before using JIDT with the function `javaaddpath(samplePath)` (at line 1);

2. Construct classes using the `javaObject()` function (see line 4);

3. Use of objects is otherwise almost the same as in Java itself, however:

   (a) In Octave, conversion between native *array* data types and Java arrays is not straightforward; we recommend using the supplied functions for such conversion in `demos/octave`, e.g. `octaveToJavaIntArray.m`. These are described on the `OctaveJavaArrayConversion` wiki page (Table I), and see example use in line 7 here, and in `example2TeMultidimBinaryData.m` and `example5TeBinaryMultivarTransfer.m`.

   (b) In Java arrays are indexed from 0, whereas in Octave or MATLAB these are indexed from 1. So when you call a method on a Java object such as `MatrixUtils.select(double data, int fromIndex, int length)` – even from within MATLAB/Octave – you must be aware that `fromIndex` will be indexed from 0 inside the toolkit, not 1!

C. Python Demos

Similarly, the "Python code examples" set at `demos/python` in the distribution provide a basic set of demonstration scripts for using the toolkit in Python. The set is described in some detail at the `PythonExamples` wiki page (Table I). See Section IV A regarding installation requirements for running the toolkit in Python, with more details at the `UseInPython` wiki page (Table I).

Again, the scripts in this set mirror the Java code in the "Simple Java Demos" set (Section V A), to demonstrate that *anything which JIDT can do in a Java environment can also be done in Python.*

Note that this set uses the `JPype` library (http://jpype.sourceforge.net/) to create the Python-Java interface, and the examples would need to be altered if you wish to use a different interface. The user is referred to the distribution or the `PythonExamples` wiki page for more details on the examples.

An illustrative example is provided in Listing 11, which converts Listing 2 into Python:

```
1  from jpype import *
2  import random
3  startJVM(getDefaultJVMPath(), "-ea", "-Djava.class.path=../../infodynamics.jar")
4  sourceArray = [random.randint(0,1) for r in xrange(100)]
5  destArray = [0] + sourceArray[0:99];
6  teCalcClass = JPackage("infodynamics.measures.discrete").TransferEntropyCalculatorDiscrete
7  teCalc = teCalcClass(2,1)
8  teCalc.initialise()
9  teCalc.addObservations(sourceArray, destArray)
10 result = teCalc.computeAverageLocalOfObservations()
11 shutdownJVM()
```

Listing 11. Estimation of TE from discrete data in Python; adapted from `example1TeBinaryData.py`

This example illustrates several important steps for using JIDT from Python via JPype:

1. Import the relevant packages from JPype (line 1);

2. Start the JVM and specify the classpath (i.e. the location of the `infodynamics.jar` library) before using JIDT with the function `startJVM()` (at line 3);

3. Construct classes using a reference to their package (see line 6 and line 7);

4. Use of objects is otherwise almost the same as in Java itself, however conversion between native *array* data types and Java arrays can be tricky – see comments on the `UseInPython` wiki page (see Table I).

5. Shutdown the JVM when finished (line 11).

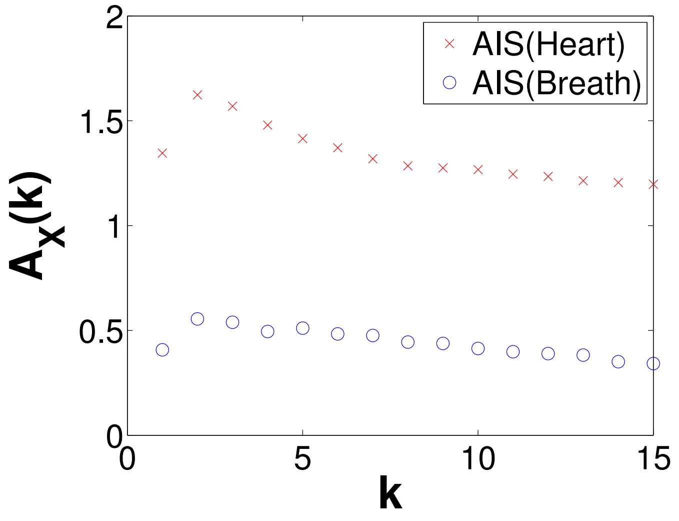

FIG. 3. Active information storage computed by the KSG estimator ($K = 4$ nearest neighbours) as a function of embedded history length for the heart and breath rate time-series data.

D. Schreiber's Transfer Entropy Demos

The "Schreiber Transfer Entropy Demos" set at `demos/octave/SchreiberTransferEntropyExamples` in the distribution recreates the original examples from Schreiber's paper introducing transfer entropy [22]. The set is described in some detail at the `SchreiberTeDemos` wiki page (see Table I). The demo can be run in MATLAB or Octave.

The set includes computing TE with a discrete estimator for data from a Tent Map simulation, with a box-kernel estimator for data from a Ulam Map simulation, and again with a box-kernel estimator for heart and breath rate data from a sleep apnea patient.[19] Importantly, the demo shows correct values for important parameter settings (e.g. use of bias correction) which were not made clear in the original paper.

We also revisit the heart-breath rate analysis using a KSG estimator, demonstrating how to select embedding dimensions k and l. As an example, we show in Fig. 3 a calculation of AIS (Eq. (23)) for the heart and breath rate data, using a KSG estimator with $K = 4$ nearest neighbours, as a function of embedding length k. This plot is produced by calling the MATLAB function: `activeInfoStorageHeartBreathRatesKraskov(1:15, 4)`. Since the KSG estimator is bias corrected, then we can use the peak to suggest that an embedded history of $k = 2$ for both heart and breath time-series is appropriate to capture all relevant information from the past without adding more spurious than relevant information as k increases. (The result is stable with the number of nearest neighbours K.) We continue on to use those embedding lengths for further investigation with the TE in the demonstration.

E. Cellular Automata Demos

The "Cellular Automata Demos" set at `demos/octave/CellularAutomata` in the distribution provide a standalone demonstration of the utility of *local* information dynamics profiles. The scripts allow the user to reproduce the key results from [15, 21, 23, 45, 46, 137] etc., i.e. plotting local information dynamics measures at every point in space-time in the cellular automata (CA). These results confirmed the long-held conjectures that gliders are the dominant information transfer entities in CAs, while blinkers and background domains are the dominant information storage components, and glider/particle collisions are the dominant information modification events.

The set is described in some detail at the `CellularAutomataDemos` wiki page (see Table I). The demo can be run in MATLAB or Octave. The main file for the demo is `plotLocalInfoMeasureForCA.m`, which can be used to specify a CA type to run and which measure to plot an information profile for. Several higher-level scripts are available

[19] This data set was made available via the Santa Fe Institute time series contest held in 1991 [136] and redistributed with JIDT with kind permission from Andreas Weigend.

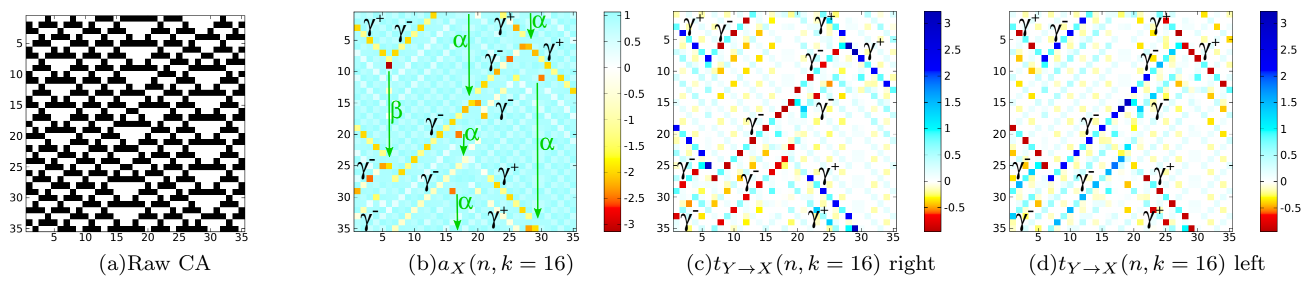

FIG. 4. Local information dynamics in ECA **rule 54** for the raw values in (a) (black for "1", white for "0"). 35 time steps are displayed for 35 cells, and time increases down the page for all CA plots. All units are in bits, as per scales on the right-hand sides. (b) Local active information storage; Local apparent transfer entropy: (c) one cell to the right, and (d) one cell to the left per time step. NB: Reprinted with kind permission of Springer Science+Business Media from [21]: J. T. Lizier, in *Directed Information Measures in Neuroscience*, Understanding Complex Systems, edited by M. Wibral, R. Vicente, and J. T. Lizier (Springer, Berlin/Heidelberg, 2014) pp. 161–193.

to demonstrate how to call this, including `DirectedMeasuresChapterDemo2013.m` which was used to generate the figures in [86] (reproduced in Fig. 4).

F. Other Demos

The toolkit contains a number of other demonstrations, which we briefly mention here:

- The "Interregional Transfer demo" set at `demos/java/interregionalTransfer/` is a higher-level example of computing information transfer between two regions of variables (e.g. brain regions in fMRI data), using multivariate extensions to the transfer entropy, to infer effective connections between the regions. This demonstration implements the method originally described in [33]. Further documentation is provided via the `Demos` wiki page (see Table I).
- The "Detecting interaction lags" demo set at `demos/octave/DetectingInteractionLags` shows how to use the transfer entropy calculators to investigate a source-destination lag that is different to 1 (the default). In particular, this demo was used to make the comparisons of using transfer entropy (TE) and momentary information transfer (MIT) [138] to investigate source-destination lags in [89] (see Test cases Ia and Ib therein). In particular, the results show that TE is most suitable for investigating source-destination lags as MIT can be deceived by source memory, and also that symbolic TE (Section III B 4) can miss important components of information in an interaction. Further documentation is provided via the `Demos` wiki page (see Table I).
- The "Null distribution" demo set at `demos/octave/NullDistributions` explores the match between analytic and bootstrapped distributions of MI, conditional MI and TE under null hypotheses of no relationship between the data sets (see Section II E). Further documentation is provided via the `Demos` wiki page (see Table I).

VI. CONCLUSION

We have described the Java Information Dynamics Toolkit (JIDT), open-source toolkit available on a Google code, which implements information-theoretic measures of dynamics via several different estimators. We have described the architecture behind the toolkit and how to use it, providing several detailed code demonstrations.

In comparison to related toolkit, JIDT provides implementations for a wider array of information-theoretic measures, with a wider variety of estimators implemented, adds implementations of local measures and statistical significance, and is standalone software. Furthermore, being implemented in Java, JIDT is platform agnostic and requires little to no installation, is fast, exhibits an intuitive object-oriented design, and can be used in MATLAB, Octave and Python.

JIDT has been used to produce results in publications by both this author and others [19, 21, 28, 40, 42, 69, 86, 89, 137, 139].

It may be complemented by the Java Partial Information Decomposition (JPID) toolkit [75, 140], which implements early attempts [72] to separately measure redundant and synergistic components of the conditional mutual information (see Section II A).

We are planning the extension or addition of several important components in the future. Of highest priority are our plans to add fast nearest neighbour searches to the KSG estimators, exploring the use of multi-threading and GPU computing, and automated parameter selection for time-series embedding. We will add implementations for "missing" estimators to complete Table II, and aim for larger code coverage by our unit tests. Most importantly however, we seek collaboration on the project from other developers in order to expand the capabilities of JIDT, and we will welcome volunteers who wish to contribute to the project.

ACKNOWLEDGMENTS

I am grateful to many collaborators and contacts who have tested JIDT in their applications, provided test data, found bugs and provided feedback, etc. These include primarily: X. Rosalind Wang, Michael Wibral, Oliver Cliff, Siddharth Pritam, Rommel Ceguerra, and Oliver Obst; as well as: Sakyasingha Dasgupta, Heni Ben Amor, Mizuki Oka, Ipek Özdemir, Christoph Hartmann and Michael Harré.

[1] C. E. Shannon, Bell System Technical Journal **27**, 379 (1948).
[2] T. M. Cover and J. A. Thomas, *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*, 99th ed. (Wiley-Interscience, New York, 1991).
[3] D. J. C. MacKay, *Information Theory, Inference, and Learning Algorithms* (Cambridge University Press, Cambridge, 2003).
[4] M. Prokopenko, F. Boschietti, and A. J. Ryan, Complexity **15**, 11 (2009).
[5] M. Mitchell, *Complexity: A guided tour* (Oxford University Press, New York, 2009).
[6] M. Gell-Mann, *The Quark and the Jaguar* (W.H. Freeman, New York, 1994).
[7] G. Tononi, O. Sporns, and G. M. Edelman, Proceedings of the National Academy of Sciences **91**, 5033 (1994).
[8] C. Adami, BioEssays : news and reviews in molecular, cellular and developmental biology **24**, 1085 (2002).
[9] M. Prokopenko, P. Wang, P. Valencia, D. Price, M. Foreman, and A. Farmer, Artificial Life **11**, 407 (2005).
[10] O. Miramontes, Complexity **1**, 56 (1995).
[11] R. V. Solé and S. Valverde, Physica A **289**, 595 (2001).
[12] M. Prokopenko, J. T. Lizier, O. Obst, and X. R. Wang, Physical Review E **84**, 041116+ (2011).
[13] R. V. Solé and S. Valverde, in *Complex Networks*, Lecture Notes in Physics, Vol. 650, edited by E. Ben-Naim, H. Frauenfelder, and Z. Toroczkai (Springer, Berlin / Heidelberg, 2004) pp. 189–207.
[14] M. Piraveenan, M. Prokopenko, and A. Y. Zomaya, European Physical Journal B **67**, 291 (2009).
[15] J. T. Lizier, *The Local Information Dynamics of Distributed Computation in Complex Systems*, Springer Theses (Springer, Berlin / Heidelberg, 2013).
[16] C. G. Langton, Physica D: Nonlinear Phenomena **42**, 12 (1990).
[17] P. Fernández and R. V. Solé, in *Power Laws, Scale-Free Networks and Genome Biology*, Molecular Biology Intelligence Unit, edited by E. V. Koonin, Y. I. Wolf, and G. P. Karev (Springer US, 2006) pp. 206–225.
[18] M. Mitchell, in *Non-Standard Computation*, edited by T. Gramß, S. Bornholdt, M. Groß, M. Mitchell, and T. Pellizzari (Wiley-VCH Verlag GmbH & Co. KGaA, Weinheim, 1998) pp. 95–140.
[19] X. R. Wang, J. M. Miller, J. T. Lizier, M. Prokopenko, and L. F. Rossi, PLoS ONE **7**, e40084+ (2012).
[20] P. Gong and C. van Leeuwen, PLoS Computational Biology **5** (2009).
[21] J. T. Lizier, M. Prokopenko, and A. Y. Zomaya, in *Guided Self-Organization: Inception*, Emergence, Complexity and Computation, Vol. 9, edited by M. Prokopenko (Springer Berlin Heidelberg, 2014) pp. 115–158.
[22] T. Schreiber, Physical Review Letters **85**, 461 (2000).
[23] J. T. Lizier, M. Prokopenko, and A. Y. Zomaya, Information Sciences **208**, 39 (2012).
[24] W. Bialek, I. Nemenman, and N. Tishby, Physica A: Statistical Mechanics and its Applications **302**, 89 (2001), arXiv:physics/0103076.
[25] J. P. Crutchfield and D. P. Feldman, Chaos **13**, 25 (2003).
[26] R. D. Beer and P. L. Williams, Cognitive Science(2014), "Information Processing and Dynamics in Minimally Cognitive Agents", in press.
[27] C. J. Honey, R. Kötter, M. Breakspear, and O. Sporns, Proceedings of the National Academy of Sciences of the United States of America **104**, 10240 (2007).
[28] *Directed Information Measures in Neuroscience*, edited by M. Wibral, R. Vicente, and J. T. Lizier (Springer, Berlin, Heidelberg, 2014).
[29] M. Wibral, R. Vicente, and M. Lindner, in *Directed Information Measures in Neuroscience*, Understanding Complex Systems, edited by M. Wibral, R. Vicente, and J. T. Lizier (Springer, Berlin/Heidelberg, 2014) pp. 3–36.
[30] D. Marinazzo, G. Wu, M. Pellicoro, L. Angelini, and S. Stramaglia, PLoS ONE **7**, e45026+ (2012).
[31] S. Stramaglia, G.-R. Wu, M. Pellicoro, and D. Marinazzo, in *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (IEEE, 2012) pp. 3668–3671, arXiv:1203.3037.

[32] L. Faes, G. Nollo, and A. Porta, Physical Review E **83**, 051112+ (2011).
[33] J. T. Lizier, J. Heinzle, A. Horstmann, J.-D. Haynes, and M. Prokopenko, Journal of Computational Neuroscience **30**, 85 (2011).
[34] R. Vicente, M. Wibral, M. Lindner, and G. Pipa, Journal of Computational Neuroscience **30**, 45 (2011).
[35] O. Stetter, D. Battaglia, J. Soriano, and T. Geisel, PLoS Computational Biology **8**, e1002653+ (2012).
[36] V. A. Vakorin, O. A. Krakovska, and A. R. McIntosh, Journal of Neuroscience Methods **184**, 152 (2009).
[37] S. Ito, M. E. Hansen, R. Heiland, A. Lumsdaine, A. M. Litke, and J. M. Beggs, PLoS ONE **6**, e27431+ (2011).
[38] V. Mäki-Marttunen, I. Diez, J. M. Cortes, D. R. Chialvo, and M. Villarreal, Frontiers in Neuroinformatics **7**, 24+ (2013), arXiv:1310.3217.
[39] W. Liao, J. Ding, D. Marinazzo, Q. Xu, Z. Wang, C. Yuan, Z. Zhang, G. Lu, and H. Chen, NeuroImage **54**, 2683 (2011).
[40] M. Wibral, J. T. Lizier, S. Vögler, V. Priesemann, and R. Galuske, Frontiers in Neuroinformatics **8**, 1+ (2014).
[41] L. Faes and A. Porta, in *Directed Information Measures in Neuroscience*, Understanding Complex Systems, edited by M. Wibral, R. Vicente, and J. T. Lizier (Springer Berlin Heidelberg, 2014) pp. 61–86.
[42] C. Gómez, J. T. Lizier, M. Schaum, P. Wollstadt, C. Grützner, P. Uhlhaas, C. M. Freitag, S. Schlitt, S. Bölte, R. Hornero, and M. Wibral, Frontiers in Neuroinformatics **8**, 9+ (2014).
[43] J. R. Mahoney, C. J. Ellison, R. G. James, and J. P. Crutchfield, Chaos **21**, 037112+ (2011).
[44] L. Barnett, J. T. Lizier, M. Harré, A. K. Seth, and T. Bossomaier, Physical Review Letters **111**, 177203+ (2013).
[45] J. T. Lizier, M. Prokopenko, and A. Y. Zomaya, Physical Review E **77**, 026110+ (2008).
[46] J. T. Lizier, M. Prokopenko, and A. Y. Zomaya, Chaos **20**, 037109+ (2010).
[47] J. T. Lizier, M. Prokopenko, and A. Y. Zomaya, in *Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems (ALife XI), Winchester, UK*, edited by S. Bullock, J. Noble, R. Watson, and M. A. Bedau (MIT Press, Cambridge, MA, 2008) pp. 374–381.
[48] C. Damiani, S. Kauffman, R. Serra, M. Villani, and A. Colacci, in *Cellular Automata*, Lecture Notes in Computer Science, Vol. 6350, edited by S. Bandini, S. Manzoni, H. Umeo, and G. Vizzari (Springer, Berlin / Heidelberg, 2010) pp. 1–11.
[49] C. Damiani and P. Lecca, in *Fifth UKSim European Symposium on Computer Modeling and Simulation (EMS)* (IEEE, 2011) pp. 129–134.
[50] L. Sandoval, Entropy **16**, 4443 (2014).
[51] J. T. Lizier, S. Pritam, and M. Prokopenko, Artificial Life **17**, 293 (2011).
[52] J. T. Lizier, F. M. Atay, and J. Jost, Physical Review E **86**, 026110+ (2012).
[53] G. V. Steeg and A. Galstyan, in *Proceedings of the Sixth ACM international conference on Web search and data mining*, WSDM '13 (ACM, New York, NY, USA, 2013) pp. 3–12, arXiv:1208.4475.
[54] M. Oka and T. Ikegami, PLoS ONE **8**, e60398+ (2013).
[55] T. L. Bauer, R. Colbaugh, K. Glass, and D. Schnizlein, in *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, CSIIRW '13 (ACM, New York, NY, USA, 2013).
[56] M. Prokopenko, V. Gerasimov, and I. Tanev, in *Proceedings of the 10th International Conference on the Simulation and Synthesis of Living Systems (ALifeX), Bloomington, Indiana, USA*, edited by L. M. Rocha, L. S. Yaeger, M. A. Bedau, D. Floreano, R. L. Goldstone, and A. Vespignani (MIT Press, 2006) pp. 185–191.
[57] M. Lungarella and O. Sporns, PLoS Computational Biology **2**, e144+ (2006).
[58] K. Nakajima, T. Li, R. Kang, E. Guglielmino, D. G. Caldwell, and R. Pfeifer, in *2012 IEEE International Conference on Robotics and Biomimetics (ROBIO)* (IEEE, 2012) pp. 1273–1280.
[59] K. Nakajima and T. Haruna, The European Physical Journal Special Topics **222**, 437 (2013).
[60] P. L. Williams and R. D. Beer, in *From Animals to Animats 11*, Lecture Notes in Computer Science, Vol. 6226, edited by S. Doncieux, B. Girard, A. Guillot, J. Hallam, J.-A. Meyer, and J.-B. Mouret (Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010) Chap. 4, pp. 38–49.
[61] O. Obst, J. Boedecker, B. Schmidt, and M. Asada, "On active information storage in input-driven systems," (2013), arXiv:1303.5526.
[62] J. Boedecker, O. Obst, J. T. Lizier, Mayer, and M. Asada, *Theory in Biosciences*, Theory in Biosciences **131**, 205 (2012).
[63] O. M. Cliff, J. T. Lizier, X. R. Wang, P. Wang, O. Obst, and M. Prokopenko, in *RoboCup 2013: Robot World Cup XVII*, Lecture Notes in Computer Science, Vol. 8371, edited by S. Behnke, M. Veloso, A. Visser, and R. Xiong (Springer, Berlin/Heidelberg, 2014) pp. 1–12.
[64] J. T. Lizier, M. Piraveenan, D. Pradhana, M. Prokopenko, and L. S. Yaeger, in *Proceedings of the European Conference on Artificial Life (ECAL)*, Lecture Notes in Computer Science, Vol. 5777, edited by G. Kampis, I. Karsai, and E. Szathmáry (Springer, Berlin / Heidelberg, 2011) Chap. 18, pp. 140–147.
[65] M. Prokopenko, V. Gerasimov, and I. Tanev, in *From Animals to Animats 9: Proceedings of the Ninth International Conference on the Simulation of Adaptive Behavior (SAB'06)*, Lecture Notes in Computer Science, Vol. 4095, edited by S. Nolfi, G. Baldassarre, R. Calabretta, J. C. T. Hallam, D. Marocco, J.-A. Meyer, O. Miglino, and D. Parisi (Springer, Berlin Heidelberg, 2006) pp. 558–569.
[66] J. T. Lizier, M. Prokopenko, I. Tanev, and A. Y. Zomaya, in *Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems (ALife XI), Winchester, UK*, edited by S. Bullock, J. Noble, R. Watson, and M. A. Bedau (MIT Press, Cambridge, MA, 2008) pp. 366–373.
[67] A. S. Klyubin, D. Polani, and C. L. Nehaniv, PLoS ONE **3** (2008).
[68] N. Ay, N. Bertschinger, R. Der, F. Güttler, and E. Olbrich, European Physical Journal B **63**, 329 (2008).
[69] S. Dasgupta, F. Wörgötter, and P. Manoonpong, *Evolving Systems* **4**, 235 (2013).
[70] O. Obst, J. Boedecker, and M. Asada, in *Neural Information Processing. Models and Applications*, Lecture Notes in

Computer Science, Vol. 6444, edited by K. Wong, B. Mendis, and A. Bouzerdoum (Springer, Berlin/Heidelberg, 2010) Chap. 24, pp. 193–200.
[71] M. Prokopenko, HFSP Journal **3**, 287 (2009).
[72] P. L. Williams and R. D. Beer(2010), arXiv:1004.2515.
[73] M. Harder, C. Salge, and D. Polani, Physical Review E **87**, 012130+ (2013).
[74] V. Griffith and C. Koch, in *Guided Self-Organization: Inception*, Emergence, Complexity and Computation, Vol. 9, edited by M. Prokopenko (Springer, Berlin/Heidelberg, 2014) pp. 159–190, arXiv:1205.4265.
[75] J. T. Lizier, B. Flecker, and P. L. Williams, in *Proceedings of the 2013 IEEE Symposium on Artificial Life (ALIFE)* (IEEE, 2013) pp. 43–51, arXiv:1303.3440.
[76] N. Bertschinger, J. Rauh, E. Olbrich, and J. Jost, “Shared Information – New Insights and Problems in Decomposing Information in Complex Systems,” (2012), arXiv:1210.5902.
[77] J. T. Lizier, M. Prokopenko, and A. Y. Zomaya, in *Proceedings of the 9th European Conference on Artificial Life (ECAL 2007)*, Lecture Notes in Computer Science, Vol. 4648, edited by Almeida, L. M. Rocha, E. Costa, I. Harvey, and A. Coutinho (Springer, Berlin / Heidelberg, 2007) Chap. 90, pp. 895–904.
[78] P. Grassberger, International Journal of Theoretical Physics **25**, 907 (1986).
[79] J. P. Crutchfield and K. Young, Physical Review Letters **63**, 105 (1989).
[80] C. R. Shalizi, *Causal Architecture, Complexity and Self-Organization in Time Series and Cellular Automata*, Ph.D. thesis, University of Wisconsin-Madison (2001).
[81] F. Takens, in *Dynamical Systems and Turbulence, Warwick 1980*, Lecture Notes in Mathematics, Vol. 898, edited by D. Rand and L.-S. Young (Springer, Berlin / Heidelberg, 1981) Chap. 21, pp. 366–381.
[82] P. L. Williams and R. D. Beer(2011), arXiv:1102.1507.
[83] L. Barnett and T. Bossomaier, Physical Review Letters **109**, 138105+ (2012).
[84] M. Lindner, R. Vicente, V. Priesemann, and M. Wibral, BMC Neuroscience **12**, 119+ (2011).
[85] L. Barnett, A. B. Barrett, and A. K. Seth, Physical Review Letters **103**, 238701+ (2009).
[86] J. T. Lizier, in *Directed Information Measures in Neuroscience*, Understanding Complex Systems, edited by M. Wibral, R. Vicente, and J. T. Lizier (Springer, Berlin/Heidelberg, 2014) pp. 161–193.
[87] R. Vicente and M. Wibral, in *Directed Information Measures in Neuroscience*, Understanding Complex Systems, edited by M. Wibral, R. Vicente, and J. T. Lizier (Springer, Berlin/Heidelberg, 2014) pp. 37–58.
[88] J. T. Lizier and M. Prokopenko, European Physical Journal B **73**, 605 (2010).
[89] M. Wibral, N. Pampu, V. Priesemann, F. Siebenhühner, H. Seiwert, M. Lindner, J. T. Lizier, and R. Vicente, PLoS ONE **8**, e55809+ (2013).
[90] N. Ay and D. Polani, Advances in Complex Systems **11**, 17 (2008).
[91] D. Chicharro and A. Ledberg, PLoS ONE **7**, e32466+ (2012).
[92] M. Prokopenko, J. T. Lizier, and D. C. Price, Entropy **15**, 524 (2013).
[93] M. Prokopenko and J. T. Lizier, Scientific Reports **4**, 5394+ (2014).
[94] L. Faes, G. Nollo, and A. Porta, Computers in Biology and Medicine **42**, 290 (2012).
[95] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing* (The MIT Press, Cambridge, MA, USA, 1999).
[96] C. R. Shalizi, R. Haslinger, J.-B. Rouquier, K. L. Klinkner, and C. Moore, Physical Review E **73**, 036104 (2006).
[97] T. Helvik, K. Lindgren, and M. G. Nordahl, in *Proceedings of the International Conference on Cellular Automata for Research and Industry, Amsterdam*, Lecture Notes in Computer Science, Vol. 3305, edited by P. M. A. Sloot, B. Chopard, and A. G. Hoekstra (Springer, Berlin/Heidelberg, 2004) pp. 121–130.
[98] J. Dasan, T. R. Ramamohan, A. Singh, and P. R. Nott, Physical Review E **66**, 021409 (2002).
[99] R. B. Ash, *Information Theory* (Dover Publications Inc., New York, 1965).
[100] R. M. Fano, *Transmission of information: a statistical theory of communications* (M.I.T. Press, Cambridge, MA, USA, 1961).
[101] M. R. DeWeese and M. Meister, Network: Computation in Neural Systems **10**, 325 (1999).
[102] M. Chávez, J. Martinerie, and M. Le Van Quyen, Journal of Neuroscience Methods **124**, 113 (2003).
[103] P. F. Verdes, Physical Review E **72**, 026222+ (2005).
[104] J. Geweke, Journal of the American Statistical Association **77**, 304 (1982).
[105] D. R. Brillinger, Brazilian Journal of Probability and Statistics **18**, 163 (2004).
[106] P. E. Cheng, J. W. Liou, M. Liou, and J. A. D. Aston, Journal of Data Science **4**, 387 (2006).
[107] L. Barnett(2013), Personal Communication.
[108] P. Wollstadt, M. Martínez-Zarzuela, R. Vicente, F. J. Díaz-Pernas, and M. Wibral, PLoS ONE **9**, e102833+ (2014).
[109] L. Paninski, Neural Computation **15**, 1191 (2003).
[110] J. A. Bonachela, H. Hinrichsen, and M. A. Muñoz, Journal of Physics A: Mathematical and Theoretical **41**, 202001+ (2008).
[111] L. Barnett, C. L. Buckley, and S. Bullock, Physical Review E **79**, 051914+ (2009).
[112] A. Kaiser and T. Schreiber, Physica D **166**, 43 (2002).
[113] H. Kantz and T. Schreiber, *Nonlinear Time Series Analysis* (Cambridge University Press, Cambridge, MA, 1997).
[114] P. Grassberger, Physics Letters A **128**, 369 (1988).
[115] M. Lungarella, T. Pegors, D. Bulwinkle, and O. Sporns, Neuroinformatics **3**, 243 (2005).
[116] K. Marton and P. C. Shields, The Annals of Probability **22**, 960 (1994).
[117] A. Kraskov, H. Stögbauer, and P. Grassberger, Physical Review E **69**, 066138+ (2004).

[118] A. Kraskov, *Synchronization and Interdependence Measures and their Applications to the Electroencephalogram of Epilepsy Patients and Clustering of Data*, Publication Series of the John von Neumann Institute for Computing, Vol. 24 (John von Neumann Institute for Computing, Jülich, Germany, 2004).
[119] L. Kozachenko and N. Leonenko, Problems of Information Transmission **23**, 9 (1987).
[120] S. Frenzel and B. Pompe, Physical Review Letters **99**, 204101+ (2007).
[121] G. Gomez-Herrero, W. Wu, K. Rutanen, M. C. Soriano, G. Pipa, and R. Vicente(2010), arXiv:1008.0539.
[122] J. Lizier, J. Heinzle, C. Soon, J. D. Haynes, and M. Prokopenko, BMC Neuroscience **12**, P261+ (2011).
[123] C. Bandt and B. Pompe, Physical Review Letters **88**, 174102+ (2002).
[124] M. Staniek and K. Lehnertz, Physical Review Letters **100**, 158101+ (2008).
[125] R. Oostenveld, P. Fries, E. Maris, and J.-M. Schoffelen, Computational Intelligence and Neuroscience **2011**, 156869+ (2011).
[126] M. Wibral, B. Rahm, M. Rieder, M. Lindner, R. Vicente, and J. Kaiser, Progress in Biophysics and Molecular Biology **105**, 80 (2011).
[127] A. Montalto, L. Faes, and D. Marinazzo, in *8th Conference of the European Study Group on Cardiovascular Oscillations (ESGCO)* (IEEE, 2014) pp. 59–60.
[128] S. Astakhov, P. Grassberger, A. Kraskov, and H. Stögbauer, "Mutual information least-dependent component analysis (MILCA)," (2013), Software, `http://www.ucl.ac.uk/ion/departments/sobell/Research/RLemon/MILCA/MILCA`.
[129] H. Stögbauer, A. Kraskov, S. Astakhov, and P. Grassberger, Physical Review E **70**, 066123+ (2004).
[130] K. Rutanen, "Tim 1.2.0," (2011), Software, `http://www.cs.tut.fi/~timhome/tim-1.2.0/tim.htm`.
[131] M. Nilsson and W. B. Kleijn, IEEE Transactions on Information Theory **53**, 2330 (2007).
[132] D. Stowell and M. D. Plumbley, IEEE Signal Processing Letters **16**, 537 (2009).
[133] L. Barnett and A. K. Seth, Journal of Neuroscience Methods **223**, 50 (2014).
[134] "Computer Language Benchmarks Game," Accessed July 25, 2014, `http://benchmarksgame.alioth.debian.org/u64q/java.php`.
[135] J. Orlandi, M. Saeed, and I. Guyon, "Chalearn connectomics challenge sample code," (2014), Software, `http://connectomics.chalearn.org`.
[136] D. R. Rigney, A. L. Goldberger, W. Ocasio, Y. Ichimaru, G. B. Moody, and R. Mark, in *Time Series Prediction: Forecasting the Future and Understanding the Past*, edited by A. S. Weigend and N. A. Gershenfeld (Addison-Wesley, Reading, MA, 1993) pp. 105–129.
[137] J. T. Lizier and J. R. Mahoney, Entropy **15**, 177 (2013).
[138] B. Pompe and J. Runge, Physical Review E **83**, 051122+ (2011).
[139] X. R. Wang, J. T. Lizier, T. Nowotny, A. Z. Berna, M. Prokopenko, and S. C. Trowell, PLoS ONE **9**, e89840+ (2014).
[140] J. T. Lizier and B. Flecker, "Java Partial Information Decomposition toolkit," (2012), `http://github.com/jlizier/jpid`.