# `dinfo`
# Directed Information and Causality Measures

Guillaume Becq, Pierre-Olivier Amblard and Olivier Michel

May 27, 2014

## 1 Introduction

This document describes the package `dinfo` for the computation of directed information and causality measures on multidimensional data. It also contains multivariate models and utility functions. The proposed measures are based on: directed information theory, Geweke's measures estimated with kernel functions, non parametric Bayesian approach (Gaussian processes). Functions in this package are written in Python and Matlab. In this document, functions are presented and their mathematical formulations are given.

### 1.1 Mathematical notation

Mathematical notation are given in Tab.1

### 1.2 Code convention

In Python, `dinfo` depends on Scipy and Numpy packages. In Matlab, there is no dependencies. Let $X \in R^d$ a variate with $n$ samples or observations. In both languages, the variable `X` is used to represent the collection of samples, $n$ and $d$ are respectively noted `nObs` and `nDim`. In Python, `X` is an `numpy.ndarray` with shape `(nObs, )` if there is only one dimension or `(nDim, nObs)` for `nDim` $> 1$. In Matlab, `X` is a `matrix` with size `nDim-by-nObs`. Saying it in another way, data are given in column vectors with the sequence of samples given in rows. For example, data corresponding

Table 1: Mathematical notations

| | | |
|---|---|---|
| $X$ | multivariate, multidimensional variable | |
| $n$ | number of samples | |
| $d$ | dimension of the variate | |
| $x$ | a sample of $X$ | |
| $x_t$, $x(t)$ | a sample of $X$ at time $t$ | |
| $x^t$ | a variate with samples of $X$ up to time $t$ | $x^t = (x(0), \ldots, x(t))$ |
| $x_{t_1}^{t_2}$ | a variate with samples of $X$ from time $t_1$ to $t_2$ | $x_{t_1}^{t_2} = (x(t_1), \ldots, x(t_2))$ |
| $Dx$ | retarded operator on $x$ | $Dx = x^{t-1}$ |
| $x_S$ | a variate with samples constrained by $S$ | $x_{t_1,t_3} = (x_{t_1}, x_{t_3})$ |
| $L$ | a learning set | |
| $T$ | a test set | |
| $F$ | cumulative distribution function | $F(x) = \Pr(X \leq x)$ |
| $f$ | probability density function | $f(x) = F'(x)$ |
| $h$ | differential entropy | $h(X) = -\int_S f(x) \log(f(x))\, dx$ |
| $S$ | a support set of a random variable | |
| $i$ | mutual information | $i(X;Y) = -\int_S f(x,y) \log\left(\frac{f(x,y)}{f(x)\, f(y)}\right) dx$ |
| $U(a,b)$ | univariate, uniform distribution on $[a,b]$ | $f(x) = \frac{1}{(b-a)}$ <br> $h(X) = -\int_a^b \frac{1}{(b-a)} \log\left(\frac{1}{(b-a)}\right) dx = \log(b-a)$ |
| $\mu$ | mean of a univariate | |
| $\sigma$ | standard deviation of a univariate | |
| $N(\mu, \sigma^2)$ | univariate, normal or Gaussian distribution | $f(x) = \frac{1}{\sqrt{2\,\pi\,\sigma^2}}\, \mathrm{e}^{\frac{-(x-\mu)^2}{2\,\sigma^2}}$ <br> $h(X) = \frac{1}{2}\log\left(2\,\pi\,\mathrm{e}\,\sigma^2\right)$ |
| $M$ | mean vector of a multivariate | |
| $V$ | covariance matrix of a multivariate | |
| $N(M,V)$ | multivariate, normal or Gaussian distribution | $f(x) = \frac{1}{\sqrt{2\,\pi} \lvert V \rvert^{\frac{1}{2}}}\, \mathrm{e}^{-\frac{1}{2}(x-M)^T\, V^{-1}\, (x-M)}$ <br> $h(X) = \frac{1}{2}\log\left((2\,\pi\,\mathrm{e})^d\, \lvert V \rvert\right)$ |

to a sequence of 4 samples for a 3 dimensional variable will be represented by `X`.

In Python:

```
>>> X = numpy.array([[1, 2, 3, 4], [11, 12, 13, 14],
    [21, 22, 23, 24]])
```

In Matlab:

```
>> X = [1, 2, 3, 4; 11, 12, 13, 14; 21, 22, 23, 24];
```

A documentation is given in the beginning of the code defining the function, using docstrings in Python, and comments in Matlab. In Python, an `help(function)` will print this documentation. In Matlab, an `help function` or `help('function')` will display this documentation on the console. The documentation generally follows these sections:

| | |
|---|---|
| *1st line* | gives the short description of the function. |
| *Syntax* | gives the general syntax. |
| *Input* | description of the input arguments. |
| *Output* | description of the output arguments. |
| *Description* | description of the function. |
| *Example* | an example of code to test the function. |
| *See also* | other linked functions. |
| *Copyright* | if not mentionned all codes are copyrighted according to the file `LICENSE.txt` given in the root of the distribution. It contains the clauses of a CeCILL-B license compatible with GNU GPL licenses but respecting French law. |

## 1.3 Related works

The C++ library `LibMI` [5] seems interesting but can only be applied to univariate time series. This library uses the `ANN` library [8] for approximate nearest neighbors evaluation. This considerably improves the computation time when doing k nearest neighbors detection on large datasets.

The Python package `NPEET` [11] proposes interesting functions for non-parametric continuous entropy estimators. We have not tested it yet, and it seems that the code implementation are comparable. The documentation is talking about the problem of manifolds, a problem we have also encountered uring the evaluation of the algorithms and the search of relevant examples. The example given in [11] will result in the example below with `dinfo`:

```
>>> x = numpy.array([1.3, 3.7, 5.1, 2.4, 3.4])
>>> y = numpy.array([1.5, 3.32, 5.3, 2.3, 3.3])
>>> dinfo.mi(x, y, 'Kraskov', (3, ))

 0.1167
```

## 1.4 Note about differential entropy

In the computation of entropy for binning, the discrete entropy is computed and the log of the volume is added to obtain the differential entropy. This is problematic when there is a degenerate case or when the variates are in manifolds with dimension less than the variate dimension. For example, when trying to compute $h(X, X)$, the bidimensional variable (X, X) contains values that are related in each dimension. The bivariate lives in a manifold with a dimension smaller than the representation space. In this condition, the computation of the density is biased since the probability is the same but the volume is increased. This implies that all results with related variables are not correct. This may be problematic since with real data, there may be a strong correlation or a link between variables. Future versions will have to study more seriously these cases. One solution is to use entropy computation on manifold or to add some small noises like it is generally done [9].

## 1.5 Organization of this document

In the rest of this document, a short description of each function is given. The functions are grouped in Python modules or Matlab packages described in Table 2. The remaining of the document follows the order given in this table.

Table 2: Organization of modules or packages

| module name | description |
|---|---|
| `dinfo` | directed information and causality measure |
| `model` | models and theoretic tools used in articles or in tests |
| `util` | some utility functions for multidimensional variables |
| `binning` | information theory measures using binning |
| `distance` | functions for computing distances |
| `kNN` | k nearest neighbors |
| `funKraskov` | information theory measures using Kraskov's methods |
| `funFrenzel` | information theory measures using Frenzel and Pompe's methods |
| `kernel` | function using kernel methods |
| `funGewekeKernel` | Geweke measures using kernel methods |
| `gpr` | causality measures using Gaussian processes with kernel methods |

## 2 Modules

### 2.1 dinfo

**h** Entropy

```
hX = h(x, method="bin", param=(2, ))
```

$$h(X)$$

This is the differential entropy estimator. It is a wrapper to `binning.h`, `funKraskov.h` depending on the method used.

**hc** Conditional entropy

```
hXkY = hc(x, y, method="bin", param=(2, ))
```

$$h(X|Y)$$

Wrapper to `binning.hc`, `funKraskov.hc`.

**mi** Mutual information

```
miXY = mi(x, y, method="bin", param=(2, ))
```

$$i(X;Y)$$

Wrapper to `binning.mi`, `funKraskov.mi`, `funFrenzel.mi`.

**mic** Conditional mutual information

```
miXYkZ = mic(x, y, z, method="bin", param=(2, ))
```

$$i(X;Y|Z)$$

Wrapper to `binning.mic`, `funFrenzel.mic`.

**te** Transfert entropy from $X$ to $Y$

```
dXY = te(x, y, p, method="bin", param=(2,))
```

This is the approximation given in [4] at order $p$ for stationary sequences:

$$I(DX \to Y)(p) = I(x_{t-p}^{t-1}; y_t | y_{t-p}^{t-1})$$

**tec** Conditional transfert entropy from $X$ to $Y$ given $Z$

```
dXYkZ = tec(x, y, z, p, method="bin", param=(2, ))
```

This is the approximation given in [4] at order $p$ for stationary sequences:

$$I(DX \to Y \| DZ)(p) = I(x_{t-p}^{t-1}; y_t | y_{t-p}^{t-1}, z_{t-p}^{t-1})$$

**iie** Instantaneous information exchange from $X$ to $Y$

```
dXY = iie(x, y, p, method="bin", param=(2, ))
```

This is the approximation given in [4] at order $p$ for stationary sequences:

$$I(X \to Y \| DX)(p) = I(x_t; y_t | x_{t-p}^{t-1}, y_{t-p}^{t-1})$$

**iieu** Instantaneous unconditional information exchange from $X$ to $Y$ given $Z$

```
dXYkZ = iieu(x, y, z, p, method="bin", param=(2, ))
```

This is the approximation given in [4] at order $p$ for stationary sequences:

$$I(X \to Y \| DX, DZ)(p) = I(x_t; y_t | x_{t-p}^{t-1}, y_{t-p}^{t-1}, z_{t-p}^{t-1})$$

**iiec** Instantaneous conditional information exchange from $x$ to $y$ given $z$

```
dXYkZ = iiec(x, y, z, p, method="bin", param=(2, ))
```

$$I(X \to Y \| DX, Z)(p) = I(x_t; y_t | x_{t-p}^{t-1}, y_{t-p}^{t-1}, z_{t-p}^{t})$$

**gcm** Geweke's causal measure with kernel methods

```
dXY = gcm(x, y, z, p, condition="", method="dynamic",
    kernelMethod="Gaussian", nFold=10,
    listLambda=[1.], param=([1.], ))
```

This is a wrapper to `dinfo.gcmd`, `dinfo.gcmi`, `dinfo.gcmcd`, `dinfo.gcmci`.

**gcmd** Geweke's dynamic causal measure from $x$ to $y$ using kernel methods

```
dXY = gcmd(x, y, p, kernelMethod="Gaussian",
    nFold=10, listLambda = [1.], param=([1.], ))
```

This is the approximation at order $p$ given in [3]:

$$G_{X \to Y}(p) = \frac{\sigma^2(y_t | y_{t-p}^{t-1})}{\sigma^2(y_t | y_{t-p}^{t-1}, x_{t-p}^{t-1})}$$

This is a wrapper to `funGewekeKernel.gcmd`.

**gcmi** Geweke's instantaneous causal measure from $x$ to $y$ using kernel methods

```
dXY = gcmi(x, y, p, kernelMethod="Gaussian",
    nFold=10, listLambda=[1.], param=([1.], ))
```

This is the approximation at order $p$ given in [3]:

$$G_{X.Y}(p) = \frac{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t-1})}{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t})}$$

This is a wrapper to `funGewekeKernel.gcmi`.

**gcmcd** Geweke's conditional dynamic causal measure from $x$ to $y$ given $z$ using kernel methods

```
dXYkZ = gcmcd(x, y, z, p, kernelMethod="Gaussian",
    nFold=10, listLambda=[1.], param=([1.], ))
```

This is the approximation at order $p$ given in [3]:

$$G_{X\to Y||Z}(p) = \frac{\sigma^2(y_t|y_{t-p}^{t-1}, z_{t-p}^{t-1})}{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t-1}, z_{t-p}^{t-1})}$$

This is a wrapper to `funGewekeKernel.gcmcd`.

**gcmci** Geweke's conditional instantaneous causal measure from $x$ to $y$ given $z$ using kernel methods

```
dXYkZ = gcmci(x, y, z, p, kernelMethod="Gaussian",
    nFold=10, listLambda=[1.], param=([1.],))
```

This is the approximation at order $p$ given in [3]:

$$G_{X.Y||Z}(p) = \frac{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t-1}, z_{t-p}^{t})}{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t}, z_{t-p}^{t})}$$

This is a wrapper to `funGewekeKernel.gcmci`.

**gprcm** Compute Gaussian process regression causality measure from $x$ to $y$

```
dXY = gprcm(x, y, kernelMethod="Gaussian",
    listLambda=[1.], param=([1.],))
```

This is the approximation at order $p$ given in [2]:

$$d_{X\to Y}(p) = \max_{\theta_2}\log\big(P_2(y_t|x_{t-p}^{t-1}, y_{t-p}^{t-1})\big) - \max_{\theta_1}\log\big(P_1(y_t|y_{t-p}^{t-1})\big)$$

Wrapper to `gpr.gprcm_Gaussian` or `gpr.gprcm_Linear`.

**gprcmc** Compute Gaussian process regression causality measure from $x$ to $y$ given $z$

```
dXYkZ = gprcmc(x, y, z, kernelMethod="Gaussian",
    listLambda=[1.], param=([1.],))
```

This is the approximation at order $p$ given in [2]:

$$d_{X\to Y|Z}(p) = \max_{\theta_2}\log\big(P_2(y_t|x_{t-p}^{t-1}, y_{t-p}^{t-1}, z_{t-p}^{t-1})\big) - \max_{\theta_1}\log\big(P_1(y_t|y_{t-p}^{t-1}, z_{t-p}^{t-1})\big)$$

Wrapper to `gpr.gprcmc_Gaussian` or `gpr.gprcmc_Linear`.

## 2.2 model

**chain** Simulate samples from a chain system, the example 5.1. in [3].

```
(x, y, z, e, M, M2) = chain(nObs)
```

Noise samples are drawn according to: $\epsilon_{x,t} \sim N(0,1)$, $\epsilon_{y,t} \sim N(0,1)$, $\epsilon_{z,t} \sim N(0,1)$.

$$\left\{\begin{array}{lllllll} x_t & = & a\,x_{t-1} & & & & + \ \epsilon_{x,t} \\ y_t & = & d_{xy}\,x_{t-1}^2 & + & b\,y_{t-1} & & + \ \epsilon_{y,t} \\ z_t & = & & & c_{yz}\,y_{t-1} & + \ c\,z_{t-1} & + \ \epsilon_{z,t} \end{array}\right.$$

$$(a, b, c, d_{xy}, c_{yz}) = (0.2, 0.5, 0.8, 0.8, 0.7)$$

**fourDimensional** Simulate samples from the example 5.2. in [3].

```
(w, x, y, z, e, M, M2) = fourDimensional(nObs)
```

Noise are $\epsilon_{w,t}$, $\epsilon_{x,t}$, $\epsilon_{y,t}$, $\epsilon_{z,t}$ with covariance given by:

$$\Gamma_\epsilon = \begin{pmatrix} 1 & \rho_1 & 0 & \rho_1\,\rho_2 \\ \rho_1 & 1 & 0 & \rho_2 \\ 0 & 0 & 1 & \rho_3 \\ \rho_1\,\rho_2 & \rho_2 & \rho_3 & 1 \end{pmatrix}$$

with $(\rho_1, \rho_2, \rho_3) = (0.66, 0.55, 0.48)$

The multivariate data are given by:

$$\begin{cases} w_t &= 0.2\,w_{t-1} & - & 0.2\,x_{t-1}^2 & & & + & 0.3\,z_{t-1} & + & \epsilon_{w,t} \\ x_t &= & & 0.3\,x_{t-1} & & & + & 0.3\,z_{t-1}^2 & + & \epsilon_{x,t} \\ y_t &= & & 0.8\,x_{t-1} - 0.5\,x_{t-1}^2 & - & 0.8\,y_{t-1} & & & + & \epsilon_{y,t} \\ z_t &= 0.2\,w_{t-1} & & & & & - & 0.4\,z_{t-1} & + & \epsilon_{z,t} \end{cases}$$

**GlassMackay** Two dimensional system based on a Glass and Mackey's model used in [2].

```
(x, y) = GlassMackey(nObs, epsilonX, epsilonY, alpha)
```

$$\begin{cases} x_t = x_{t-1} - 0.4\left(x_{t-1} - \frac{2\,x_{t-4}}{1+x_{t-4}^{10}}\right) y_{t-5} + 0.3\,y_{t-3} + \epsilon_{x,t} \\ y_t = y_{t-1} - 0.4\left(y_{t-1} - \frac{2\,y_{t-2}}{1+y_{t-2}^{10}}\right) + \alpha\,x_{t-2} + \epsilon_{y,t} \end{cases}$$

In the article, noise samples are drawn according to: $\epsilon_{x,t} \sim N(0, 1e-2)$, $\epsilon_{y,t} \sim N(0, 1e-2)$ and values of $\alpha$ are set to $0, 0.01, 0.1, 0.2$.

**chain2** A chain system used in [2].

```
(x, y, z) = chain2(nObs, epsX, epsY, epsZ, a)
```

$$\begin{cases} x_t &= 1 - a\,x_{t-1}^2 & & & & & + & \epsilon_{x,t} \\ y_t &= 0.2\,(1 - a\,x_{t-1}^2) & + & 0.8\,(1 - a\,y_{t-1}^2) & & & + & \epsilon_{y,t} \\ z_t &= & & 0.2\,(1 - a\,y_{t-1}^2) & + & 0.8\,(1 - a\,z_{t-1}^2) & + & \epsilon_{z,t} \end{cases}$$

In the article, $\epsilon_{x,t} \sim N(0, 1e-4)$, $\epsilon_{y,t} \sim N(0, 1e-4)$, $\epsilon_{z,t} \sim N(0, 1e-4)$, $a = 1.8$.

**GaussianCovariate** Generate Gaussian covariates with a given covariance matrix C.

```
x = GaussianCovariate(nObs, m, C)
```

**GaussianBivariate** Generate Gaussian bivariates $X$ with a given correlation coefficient $\rho$.

```
x = GaussianBivariate(nObs, rho=1)
```

$$x_0 \sim N(0,1)$$
$$V = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$
$$R\,R^t = V \text{ Cholesky decomposition of } V$$
$$x = R\,x_0$$

**GaussianXY** Simulate samples from a Gaussian bivariate model $(X, Y)$ with a given correlation coefficient $\rho$.

```
(x, y) = GaussianXY(nObs, rho)
```

$x$ and $y$ Gaussian bivariate with variance $V$:

$$V = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$

**GaussianXYZ** Simulate samples from a Gaussian trivariate model $(X, Y, Z)$ with given correlation coefficients $\rho_{xy}$, $\rho_{yz}$ and $\rho_{zx}$.

```
(x, y, z) = GaussianXYZ(nObs, rhoXY, rhoYZ, rhoZX)
```

$(x, y, z)$ Gaussian trivariate model with variance $V = \begin{pmatrix} 1 & \rho_{xy} & \rho_{zx} \\ \rho_{xy} & 1 & \rho_{yz} \\ \rho_{zx} & \rho_{yz} & 1 \end{pmatrix}$.

```
>>> numpy.random.seed(1)
>>> (x, y, z) = model.GaussianXYZ(100, 0.9, 0.5, 0.1)
>>> print(corrcoef((x, y, z)))

[[ 1.          0.89961328  0.15558826]
 [ 0.89961328  1.          0.54838638]
 [ 0.15558826  0.54838638  1.        ]]
```

**sameCovariance** Generate a covariance matrix with the same correlation for all pairs of variables. This model is used in [7].

```
C = sameCovariance(nDim, c)
```

Example pour nDim $= 3$ :

$$C = \begin{pmatrix} 1 & c & c \\ c & 1 & c \\ c & c & 1 \end{pmatrix}$$

**aR1Bivariate** Simulate samples from a bivariate aR1 model

```
X = aR1Bivariate(nObs, cXX=1, cXY=0, cYX=1, cYY=0,
    gVW=0, sV=1, sW=1, dX=0, dY=0)
```

$$\begin{cases} x(t) = c_{xx}\, x(t-1) + c_{xy}\, y(t-1) + d_x + \epsilon_x(t) \\ y(t) = c_{yx}\, x(t-1) + c_{yy}\, y(t-1) + d_y + \epsilon_y(t) \end{cases}$$

avec

$$E = \begin{pmatrix} \epsilon_x \\ \epsilon_y \end{pmatrix} \sim N(0, V),\ V = \begin{pmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{yx} & \sigma_y^2 \end{pmatrix}$$

This is the model used in [1] with $c_{xx} = 0.4$, $c_{xy} = 0$, $c_{yx} = 0.4$, $c_{yy} = 0.5$, $d_x = 0$, $d_y = 0$, $\sigma_x = 1$, $\sigma_y = 1$, $\sigma_{xy} = \sigma_{yx} = 0.5$,

**aR1Trivariate** Simulate samples from a trivariate aR1 model

```
X = aR1Trivariate(nObs, C, D, S)
```

$$X(t) = C\,X(t-1) + D + E(t)$$

with

$$X = \begin{pmatrix} x \\ y \\ z \end{pmatrix},\ C = \begin{pmatrix} c_{xx} & c_{xy} & c_{xz} \\ c_{yx} & c_{yy} & c_{yz} \\ c_{zx} & c_{zy} & c_{zz} \end{pmatrix},\ E \sim N(0,\Sigma),\ \Sigma = \begin{pmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_y^2 & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_z^2 \end{pmatrix}$$

This is the model used in [1] with :

$$C = \begin{pmatrix} 0.4 & 0.0 & -0.6 \\ 0.4 & 0.5 & 0.0 \\ 0.0 & 0.5 & -0.5 \end{pmatrix} \text{ or } C = \begin{pmatrix} 0.4 & 0.5 & -0.6 \\ 0.4 & 0.5 & 0.0 \\ 0.0 & 0.5 & -0.5 \end{pmatrix},\ \Sigma = \begin{pmatrix} 1 & 0.5 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

**GaussianH** Compute Gaussian entropy given $V$ matrix of covariance.

```
hTh = GaussianH(V)
```

$$h = \frac{1}{2}\log\big((2\,\pi\,e)^d\,|V|\big)$$

**coupledLorenzSystems** Simulate samples from three coupled Lorenz systems used in [7].

```
(x, y, z) = coupledLorenzSystems(nObs, K, tau,
    b=8./3., sigma=10., r=28., tauMax=60)
```

$$\begin{cases} \dot{X}_i(t) = \sigma\,(Y_i(t) - X_i(t)) \\ \dot{Y}_i(t) = r\,X_i(t) - Y_i(t) - X_i(t)\,Z_i(t) + \sum_{j\neq i} K_{ij}\,Y_j^2(t-\tau_{ij}) \\ \dot{Z}_i(t) = X_i(t)\,Y_i(t) - b\,Z_i(t) \end{cases}$$

## 2.3 util

**getListOfCases** Get a list of all cases combining lists in a tuple

```
listOfCases = util.getListOfCases(tupleListParam)
```

This is used to generate a list of values in a grid. This function is used during parameters optimization for evaluating all cases in one loop.

```
>>> param = (theta1, theta2, theta3) = ([1, 2, 3, 4],
    [11., 12.], [21, 22, 23])
>>> listOfCases = getListOfCases(param)
>>> print(listOfCases)

[[ 1.  2.  3.  4.  1.  2.  3.  4.  1.  2.  3.  4.  1.
   2.  3.  4.  1.  2.  3.  4.  1.  2.  3.  4.]
 [11. 11. 11. 11. 12. 12. 12. 12. 11. 11. 11. 11. 12.
  12. 12. 12. 11. 11. 11. 11. 12. 12. 12. 12.]
 [21. 21. 21. 21. 21. 21. 21. 21. 22. 22. 22. 22. 22.
  22. 22. 22. 23. 23. 23. 23. 23. 23. 23. 23.]]
```

**getTM1** Generate the extension $x^{t-1}$ at order $p$

```
>>> x = numpy.array([[1, 2, 3, 4, 5],
    [11, 12, 13, 14, 15]])
>>> print(util.getTM1(x, 2))
[[  0.   0.   2.   3.   4.]
 [  0.   0.   1.   2.   3.]
 [  0.   0.  12.  13.  14.]
 [  0.   0.  11.  12.  13.]]
```

$$x_{t-p}^{t-1}$$

Unknown values for the beginning of the sequence are set to 0. Known values start at index $p$ in Python and $p+1$ in Matlab.

**getT** Generate the extension $x^t$ given order $p$

```
>>> x = numpy.array([[1, 2, 3, 4, 5],
    [11, 12, 13, 14, 15]])
>>> print(util.getT(x, 2))
[[  0.   0.   3.   4.   5.]
 [  0.   0.   2.   3.   4.]
 [  0.   0.   1.   2.   3.]
 [  0.   0.  13.  14.  15.]
 [  0.   0.  12.  13.  14.]
 [  0.   0.  11.  12.  13.]]
```

$$x_{t-p}^{t}$$

Unknown values for the beginning of the sequence are set to 0. Known values start at index $p$ in Python and $p+1$ in Matlab.

**getTList** Compute the extension $x_{t-t_0,t-t_1,\ldots,t-t_{nT}}$ at time $t$

```
>>> x = numpy.array([[1, 2, 3, 4, 5],
    [11, 12, 13, 14, 15]])
>>> print(util.getTM1(x, [1, 3]))
[[  0.   0.   0.   3.   4.]
 [  0.   0.   0.   1.   2.]
 [  0.   0.   0.  13.  14.]
 [  0.   0.   0.  11.  12.]]
```

$$x_{t-t_0,t-t_1,\ldots,t-t_{nT}}$$

Unknown values for the beginning of the sequence are set to 0. Known values start at index $\max(t_i)$ in Python and $\max(t_i+1)$ in Matlab.

**mse** Compute the mean squared error between $x$ and $\hat{x}$

```
s = util.mse(x, xHat)
```

$$\epsilon = \frac{1}{N}\sum_{i=1}^{N}(x_i-\hat{x}_i)^2$$

**msecoef** Compute the mean squared errors between vectors in `x`

```
E = util.msecoef(x)
```

**preprocessRedCent** Reduce and center a vector

```
xc = util.preprocessRedCent(x)
```

$$x_c = \frac{x-\mu}{\sigma}$$

## 2.4 binning

This section contains functions for computing entropies and mutual information using binning. Binning referred to the operation of dividing the support of the variate into hypercubic cells also known as bins. This is done by dividing each marginal support into $m$ bins. For multivariate with $d$ dimensions and using $m$, this leads to the creation of $m^d$ hypercubes of the same size. An estimation of the probability density function of the variate is done on each bin. This can be viewed as doing an histogram in $d$ dimensions.

For the estimation of the differential entropy, the discrete entropy is estimated with a probability estimated by counting the number of samples in each bin. A bias is added to obtain the differential entropy. The formulation of the differential entropy is then given by [6]:

$$H(X^\Delta) + \log(\Delta) \to h(X), \text{when}\, \Delta \to 0$$

Where $H(X^\Delta)$ is the discrete entropy and $\Delta$ is the volume of a bin.

When there is a strong correlation between some dimensions in a multidimensional variable, the result may not be meaningful (see also 1.4).

**findBin** Find the index of the bin given a set of thresholds such that $\theta_i \leq x < \theta_{i+1}$

```
iBin = findBin(th, x)
```

**prob** Compute probabilities in bins: $[-\infty, \theta_0[, [\theta_i, \theta_{i+1}[, [\theta_m, \infty[$

```
(p, stepX, thX, xS) = prob(x, nBin=2)
```

$$\begin{array}{rcl} b_0 & : & -\infty \leq x < \theta_0 \\ b_i & : & \theta_{i-1} \leq x < \theta_i \\ b_m & : & \theta_m \leq x < \infty \end{array}$$

$$\begin{array}{rcl} r(x) & = & \max(x) - \min(x) \\ \theta_k & = & \min(x) + \frac{k}{m}\, r(x),\; k \in [0, m] \\ \mu_k & = & \min(x) + \frac{(k+1/2)}{m}\, r(x),\; k \in [0, m-1] \\ \mu_k & = & \frac{1}{2}\,(\theta_k + \theta_{k+1}) \\ \theta_k & = & \mu_k - \frac{1}{2}\, r(x)/m \end{array}$$

$$-\infty < \theta_0 = \min(x) < \mu_0 < \theta_1 < \ldots < \theta_{m-1} < \mu_{m-1} < \theta_m = \max(x) < \infty$$

With : $m$, number of bins; $\min(x)$, minimal value of $x$; $\max(x)$, minimal value of $x$; $b_i$, bin number $i$; $\mu_i$, center value of the bin, $\theta_i$, threshold value number $i$.

It looks like Matlab **hist**, for one dimensional data. For example, if $m = 3$, for a uniform distribution the theoretical result is: $p = [0.33, 0.33, 0.33]$

```
>>> numpy.random.seed(1)
>>> x = numpy.random.rand(1, 100)
>>> p = binning.prob(x, nBin=3)[0]
>>> print(p)

[ 0.35  0.29  0.36]
```

**h** Compute entropy with binning

```
hX = h(x, nBin=3, mode="marginal")
```

- mode="marginal": nBin is considered for marginals
- mode="total": nBin is the total number of bins. nBinM the number of marginal bins is adjusted to $nBinM = \text{ceil}(nBin^{1/nDim})$

$$h(X)$$

$h(X)$ is given in nats, divide by $\log(2)$ to have it in bits.

**hc** Compute conditional entropy $h(X|Y)$

```
(hXKY, hXY, hY) = hc(x, y, nBin=2, mode="marginal")
```

$$h(X|Y) = h(X,Y) - h(Y)$$

**mi** Compute mutual information

```
(miXY, hX, hY, hXY) = mi(x, y, nBin=2,
    mode="marginal")
```

$$i(X;Y) = h(X) + h(Y) - h(X,Y)$$

**mic**

```
(micXYKZ, hXZ, hYZ, hXYZ, hZ) = mic(x, y, z, nBin=2,
    mode="marginal")
```

Compute conditional mutual information $i(X;Y|Z)$

$$i(X;Y|Z) = h(X,Z) + h(Y,Z) - h(X,Y,Z) - h(Z)$$

## 2.5 distance

**Euclid_xXI** Compute all the distances of samples in $x$ to the sample $x_i$ using the Euclidean distance

```
dist = Euclid_xXI(x, xI)
```

$$d(x, x_i) = \left( \sum_{j=1}^{d} \left(x(j) - x_i(j)\right)^2 \right)^{1/2}$$

**max_xXI** Compute all the distances of samples in $x$ to the sample $x_i$ using the maximum distance also known as the Chebyshev distance

```
dist = max_xXI(x, xI)
```

$$d(x, x_i) = \max_j |x(j) - x_i(j)|$$

## 2.6 kNN

**dist** Distances of the k nearest neighbors (kNN) for all samples in $x$

```
d = dist(x, k=1, method="Euclidean")
```

```
>>> numpy.random.seed(1)
>>> x = numpy.random.rand(1, 100)
>>> d = kNN.dist(x, 3)
>>> set_printoptions(precision=5, suppress=True)
>>> print(d[:, 0:5])

[[ 0.00028  0.00567  0.00276  0.00872  0.00003 ]
 [ 0.00217  0.02057  0.01817  0.01109  0.00637 ]
 [ 0.00284  0.02592  0.01925  0.01318  0.00748 ]]
```

## 2.7 funKraskov

The functions implements definitions given in [9].

**h** Compute entropy $h(X)$

```
hX = h(x, k=1)
```

$$h(X) = \psi(k) + \psi(N) + \log(c_d) + \frac{d}{N} \sum_{i=1}^{N} \log(\epsilon_i)$$

This is actually the Leonenko estimator given in [10].

**hc** Compute conditional entropy $h(X|Y)$

```
(hXkY, hX, miXY) = hc(x, y, k=1)
```

$$h(X|Y) = h(X) - i(X;Y)$$

**mi1** Compute mutual information $i(X;Y)$ using Kraskov's first method

```
(mi, epsilon, nX, nY) = mi1(x, y, k=1,
    metric="Euclidean")
```

$$I^1(X,Y) = \psi(k) - \langle \psi(n_x + 1) + \psi(n_y + 1) \rangle + \psi(N)$$

**mi2** Compute mutual information $i(X;Y)$ using Kraskov's second method

```
(mi, epsilon, nX, nY) = mi2(x, y, k=1,
    metric="Euclidean")
```

$$I^2(X,Y) = \psi(k) - 1/k - \langle \psi(n_x) + \psi(n_y) \rangle + \psi(N)$$

**mi** Compute mutual information $i(X;Y)$

```
miXY = mi1(x, y, k=1, metric="Euclidean")
```

$$i(X;Y)$$

This is a wrapper to `funKraskov.mi1`

## 2.8 funFrenzel

The functions in this section correspond to definitions given in the article by Frenzel and Pompe [7].

**hN** Compute the opposite of the sum of inverse from 1 to $N$

```
s = hN(N)
```

$$h_N = -\sum_{n=1}^{N} \frac{1}{n}$$

**mic** Compute conditional mutual information or partial mutual information

```
(miXYKZ, nXZ, nYZ, nZ) = mic(x, y, z, k=1,
    metric="Euclidean")
```

$$I(X;Y|Z) = \langle h_{N_{xz}(t)} + h_{N_{yz}(t)} - h_{N_z(t)} \rangle - h_{k-1}$$

**mi** Compute the mutual information

```
(miXY, nX, nY) = mi(x, y, k=1, metric="Euclidean")
```

$$I(X;Y) = \langle h_{N_x(t)} + h_{N_y(t)} \rangle - h_{T-1} - h_{k-1}$$

## 2.9 kernel

This section contains functions about kernel based methods.

**kernelGaussian** Compute a Gaussian kernel between vector $x$ and $y$.

```
k = kernelGaussian(x, y, beta)
```

$$k(x,y) = \mathrm{e}^{-\frac{\|x-y\|^2}{\beta^2}}$$

**GaussianGram** Compute a Gram matrix with Gaussian kernel from matrix $A$.

```
G = GaussianGram(A, beta)
```

The elements of the Gram matrix $G$ are given by:

$$G_{i,j} = k(X_i, X_j)$$

with $k$ a kernel function and $X_i = A_{.i}$, a column vector.

**GaussianGramXY** Compute a Gram matrix with Gaussian kernel from matrix $X$ and $Y$

```
(GXY, GYX) = GaussianGramXY(X, Y, beta)
```

**vectorGaussianKernel** Compute the vector of kernel function $k(w_i, w_{.j})$

```
k = vectorGaussianKernel(wi, w, beta)
```

**kernelLinear** Compute a linear kernel between vector $x$ and $y$

```
k = kernel.kernelLinear(x, y, param=0)
```

$$k(x,y) = x^t\, y$$

**LinearGram** Same as `GaussianGram` with linear kernel.

```
G = LinearGram(A, param=0)
```

**LinearGramXY** Same as `GaussianGramXY` with linear kernel

```
(GXY, GYX) = LinearGramXY(X, Y, param=0)
```

**vectorLinearKernel** Same as `vectorGaussianKernel` with linear kernel

```
k = vectorLinearKernel(wi, w, param=0)
```

**optimalKernelLearn** Learn optimal parameters for kernel

```
alpha = optimalKernelLearn(xL, wL, kernelMethod,
    lambda_, param)
```

The optimal vector $A$ (`alpha`) is given by:

$$A = M^{-1}\, x_L$$

with :

$$x_L = M\, A = L\, L^t\, A$$

$$M = (K + \lambda\, I)$$

$K$ is the Gram matrix obtained with $w_L$ the learning predictor, $x_L$ is the learning target, $I$ is the identity matrix. $\lambda$ is an optimization parameter. $L$ is the Cholesky decomposition of M :

$$M = L\, L^t$$

See [2, 12] for details.

In Matlab the \ operator is used leading to `A = L' \ xL`. In Python the `solve` function from `numpy.linalg` is used.

Example :

```
>>> xL = numpy.array([1, 2, 3])
>>> wL = numpy.array([[2, 3, 4], [1, 1, 1]])
>>> alpha = kernel.optimalKernelLearn(xL, wL,
    "Gaussian", 1., 0.1)
>>> print(alpha)
    [[ 0.60159903]
     [ 0.79742798]
     [ 2.45056725]]
```

**optimalKernelTest** Test prediction with optimal parameters using kernels

```
xTHat = optimalKernelTest(wT, kernelMethod, wL,
    alpha, param)
```

The prediction is given by:

$$\hat{x}_T = k_W^t\, A$$

where: $w_T$ is the vector of test predictors; $w_L$ is the vector of learnt predictors; $k_W$ is the Gram matrix obtained from $w_T$ and $w_L$; and $A$ (`alpha`) is the optimal vector learnt.

Example:

```
>>> wL = numpy.array([[2., 3, 4], [1, 1, 1]])
>>> alpha = numpy.array([[0.6],[0.8],[2.5]])
>>> wT = numpy.array([[2., 3, 4], [1, 1, 1]])
>>> xT = kernel.optimalKernelTest(wT, "Gaussian",
    wL, alpha, 1.)
>>> print(xT)
>>> wT = numpy.array([[1., 2, 3], [1, 1, 1]])
>>> xT = kernel.optimalKernelTest(wT, "Gaussian",
    wL, alpha, 1.)
>>> print(xT)
>>> wT = numpy.array([[2., 3, 4], [3, 2, 1]])
>>> xT = kernel.optimalKernelTest(wT, "Gaussian",
    wL, alpha, 1.)
>>> print(xT)
```

```
    [ 0.94009265   1.94042627   2.80529294]
    [ 0.2356887    0.94009265   1.94042627]
    [ 0.0172184    0.71384293   2.80529294]
```

**mspe** Compute the mean squared prediction error with kernel method

```
(mspe, xHat) = kernel.mspe(xL, wL, xT, wT,
    kernelMethod, lambda_,
    param)
```

This is done to evaluate the performance of the prediction. For example, here $mspe = 0.23$:

```
>>> xL = numpy.array([1.1 , 1.9, 3.1])
>>> wL = numpy.array([[1., 2., 3.], [11., 12., 13.]])
>>> xT = numpy.array([1.1 , 1.9, 3.1])
>>> wT = numpy.array([[1., 2., 3.], [11., 12., 13.]])
>>> (mspe, xHat) = kernel.mspe(xL, wL, xT, wT,
    "Gaussian", 0.1, 10.)
>>> print((mspe, xHat))
   (0.23023282040840856,
    array([ 1.51802599,  1.98981872,  2.38733926]))
```

**mspe_crossfold** Compute mean square prediciton error for kernel with cross validation.

```
mmspe = mspe_crossfold(x, w, kernelMethod, nFold,
    lambda_, param)
```

In this function, the initial vectors for target $x$, and predictor $w$ are cut into $F$ (`nFold`) parts. The learning is realized on $F-1$ parts and tested on the last one. The learning and testing sets are exchanged $F$ times. The $mmspe$ is the mean of the $mspe$ on the $F$ trials.

If there is $nObs$ observations, parts contained $nPart = \lfloor nObs/F \rfloor$. The signal must be stationary on each part to ensure acceptable results.

**mspe_crossfold_search** Compute mean square prediciton error for Gaussian kernel with cross validation and search of the minimum for parameters in the given lists.

```
(mspeMin, lambdaMin, paramMin) =
    mspe_crossfold_search(x, w, kernelMethod,
    nFold, listLambda, tupleParam)
```

Example:

```
>>> numpy.random.seed(1)
>>> x = numpy.random.rand(100, )
>>> w = numpy.random.rand(2, 100)
>>> res = kernel.mspe_crossfold_search(x, w,
    "Gaussian", nFold, [1., 10.], ([1., 10.], ))
>>> print(res)
    (0.089259493160828068, 1.0, array([ 10.]))
```

the list of parameters (`listLambda`, lists in `tupleParam`) are combined to give a full grid on all parameters.

## 2.10 funGewekeKernel

Functions in this section are defined in [3, 4]. Here we used approximations of Geweke's measures at order $p$ for numerical purposes. For example, Geweke's dynamic causal measure will be approximated by this function at order $p$ :

$$G_{X\to Y}(p) \approx G_{X\to Y}$$

**gcmd** Compute Geweke's dynamic causal measure at order $p$ on $Y$ from $X$ with kernel using n-fold cross-validation:

```
dXY = gcmd(x, y, p, kernelMethod="Gaussian",
    nFold=10, listLambda=[1.], param=([1.], ))
```

$$G_{X\to Y}(p) = \frac{\sigma^2(y_t|y_{t-p}^{t-1})}{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t-1})}$$

**gcmi** Compute Geweke's instantaneous causal measure at order $p$ on $Y$ from $X$

```
dXY = gcmi(x, y, p, kernelMethod="Gaussian",
    nFold=10, listLambda=[1.], param=([1.], ))
```

$$G_{X.Y}(p) = \frac{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t-1})}{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t})}$$

**gcmcd** Compute Geweke's conditional dynamic causal measure on $Y$ from $X$ given $Z$

```
dXY = gcmcd(x, y, z, p, kernelMethod="Gaussian",
    nFold=10, listLambda=[1.], param=([1.], ))
```

$$G_{X\to Y||Z}(p) = \frac{\sigma^2(y_t|y_{t-p}^{t-1}, z_{t-p}^{t-1})}{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t-1}, z_{t-p}^{t-1})}$$

**gcmci** Compute Geweke's conditional instantaneous causal measure at order $p$ on $Y$ from $X$ given $Z$

```
dXY = gcmci(x, y, z, p, kernelMethod="Gaussian",
    nFold=10, listLambda=[1.], param=([1.], ))
```

$$G_{X.Y||Z}(p) = \frac{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t-1}, z_{t-p}^{t})}{\sigma^2(y_t|y_{t-p}^{t-1}, x_{t-p}^{t}, z_{t-p}^{t})}$$

## 2.11 gpr

This section describes methods used for Gaussian processes regression with definitions given in [2].

**logP_Gaussian** Compute log-evidence for Gaussian process regression with Gaussian kernel

```
(logP, mT, vT, VT) = logP_Gaussian(xL, xP, xT,
    sigma, beta)
```

$$\log(P(f_y|X)) = a + b + c$$
$$a = -\frac{1}{2}\, x_P^t\, (K + \sigma^2\, I)^{-1}\, x_P$$
$$b = -\frac{1}{2}\log\big(|K + \sigma^2\, I|\big)$$
$$c = -\frac{1}{2}\, n \log(2\,\pi)$$

$K$ is a Gram matrix computed using a Gaussian kernel. $\sigma$ is a parameter for optimization. $n$ is the number of samples in the learning set. $\frac{1}{2}\log(|K + \sigma^2\, I|)$ is computed using `sum(log(diag(L)))`, with `L` Cholesky decomposition of `M = K + sigma ** 2 * I`, and `diag(L)` is the diagonalization of matrix `L`.

**gprcm_Gaussian** Compute Gaussian processes regression causality measure from $X$ to $Y$ at order $p$ using Gaussian kernels

```
(dXY, resP2, resP1, optimalParamP2, optimalParamP1) =
    gprcm_Gaussian(x, y, p, listSigma=[1.],
    listBeta=[1.])
```

$$d_{X\to Y} = \max_{\theta_2}\log(P_2(f_y|X,Y)) - \max_{\theta_1}\log(P_1(f_y|Y))$$
$$d_{X\to Y}(p) = \max_{\theta_2}\log\big(P_2(y(t)|x_{t-p}^{t-1}, y_{t-p}^{t-1})\big) - \max_{\theta_1}\log\big(P_1(y(t)|y_{t-p}^{t-1})\big)$$

Parameters are optimized on a grid created with the values in `listSigma` and `listBeta`.

**gprcmc_Gaussian** Compute Gaussian processes regression causality measure from $X$ to $Y$ given $Z$ at order $p$ with Gaussian kernels

```
(dXYkZ, resP2, resP1, optimalParamP2,
    optimalParamP1) = gprcmc_Gaussian(x, y, z, p,
    listSigma=[1.], listBeta=[1.])
```

$$d_{x\to y|z} = \max_{\theta_2}\log(P_2(f_y|X,Y,Z)) - \max_{\theta_1}\log(P_1(f_y|Y,Z))$$
$$d_{x\to y|z} = \max_{\theta_2}\log\big(P_2(y(t)|x_{t-p}^{t-1}, y_{t-p}^{t-1}, z_{t-p}^{t-1})\big) - \max_{\theta_1}\log\big(P_1(y(t)|y_{t-p}^{t-1}, z_{t-p}^{t-1})\big)$$

Parameters are optimized on a grid created with the values in `listSigma` and `listBeta`.

**logP_Linear** Same as `logP_Gaussian` with linear kernels

```
(logP, mT, vT, VT) = logP_linear(xL, xP, xT, sigma)
```

**gprcm_Linear** same as `gprcm_Gaussian` with linear kernels

```
(dXY, resP2, resP1, optimalParamP2, optimalParamP1) =
    gprcm_linear(x, y, p, listSigma=[1.])
```

**gprcmc_Linear** same as `gprcmc_Gaussian` with linear kernels

```
(dXYkZ, resP2, resP1, optimalParamP2, optimalParamP1) =
    gprcmc_linear(x, y, z, p, listSigma=[1.])
```

## 3 Conclusion

### 3.1 Future versions and TODO list

This distribution has been developped in order to put together different algorithms for the detection of causality in multivariate time series. The first point was to prototype and organize these algorithms. Here are a list of improvements to be done:

- A library in C is in development for improving the performance and will be wrapped to Matlab and Python.
- Studies on differential entropies to exclude lower dimensions must be done.
- Extension to discrete entropy and appropriate transformations from continuous to discrete.
- Improvements of the optimum findings in gpr and kernel.
- Use of approximate nearest neighbors instead of kNN for boosting computations.
- Some issues to better manage float, list and numpy.ndarray

## References

[1] Amblard, P.-O. and Michel, O., “Measuring information flow in networks of stochastic processes”, arXiv, 2009, 0911.2873

[2] Amblard, P.-O.; Michel, O. J.; Richard, C. and Honeine, P., “A Gaussian process regression approach for testing Granger causality between time series data”, Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on, 3357-3360, 2012.

[3] Amblard, P.-O.; Vincent, R.; Michel, O. J. and Richard, C., “Kernelizing Geweke’s measures of granger causality”, Machine Learning for Signal Processing (MLSP), 2012 IEEE International Workshop on, 1-6, 2012.

[4] Amblard, P.-O. and Michel, O., “Causal conditioning and instantaneous coupling in causality graphs”, Information Sciences, Elsevier, 2014.

[5] R. S. Calsaverini and R. Vicente, “libMI - user manual”, Nov 20, 2009.

[6] Cover, T. and Thomas, J., “Elements of information theory”, Wiley Online Library, 1991, 6.

[7] Frenzel, S. and Pompe, B., “Partial mutual information for coupling analysis of multivariate time series”, Physical review letters, APS, 99, 204101, 2007.

[8] David M. Mount, “ANN Programming Manual - Version 1.1.1”, Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland. Copyright, 2006.

[9] Kraskov, A.; Stogbauer, H. and Grassberger, P., “Estimating mutual information”, Physical Review E, APS, 69, 066138, 2004.

[10] Kozachenko, L. and Leonenko, N., “On statistical estimation of entropy of random vector”, Problems Infor. Transmiss, 1987, 23, 95-101 L’article est en russe.

[11] Greg Ver Steeg, “Non-parametric Entropy Estimation Toolbox (NPEET)”, Version 1.1, July 6, 2013.

[12] Rasmussen, C. E. and Williams, C. K. I., “Gaussian Processes for Machine Learning”, MIT Press, 2006.

# Index