SpiSeMe Package User Manual

v1.1

A. Perinelli
M. Castelluzzo
L. Minati
L. Ricci

May 2020

Contents

1. Overview & License

The **SpiSeMe** package (**spi**ke **se**quences mi**me**) provides a C++, MATLAB and Python implementation of several algorithms for the generation of surrogates data out of sequences of *spikes* (or *events*).

One of the surrogate generation algorithms implemented in this package, and henceforth referred to as JODI, was originally published in

> L. Ricci, M. Castelluzzo, L. Minati and A. Perinelli, *Generation of surrogate event sequences via joint distribution of successive inter-event intervals*, Chaos **29**:121102 (2019).
> DOI: 10.1063/1.5138250

This paper is henceforth referred to as **"main reference"**. Citing this reference in manuscripts describing works that rely on JODI is appreciated.

In addition, the package implements three other algorithms for surrogate generation:

- the Iterative Amplitude-Adjusted Fourier Transform (IAAFT) method that was first proposed by T. Schreiber and A. Schmitz in Phys. Rev. Lett. **77** (1996), 635;
- a generalized algorithm for surrogate generation relying on simulated annealing that was first proposed by T. Schreiber in Phys. Rev. Lett. **80** (1998), 2105;
- a dithering (also known as *jittering*) routine that produces surrogates by adding random variates to the arrival times of events.

These algorithms are henceforth referred to as "IAAF", "SA" and "dithering", respectively.

Finally, the package also provides three analytical tools that implement the assessment of the autocorrelation of spike sequences, the assessment of cross correlation between spike sequences, and the assessment of compatibility between the distributions of inter-event intervals associated to two spike sequences.

Licensing

This package, all the included source code, documentation and examples are released under the **GNU General Public License (GPL) v3**. A copy of the license is provided in the package root folder.

MathWorks, MATLAB and Simulink are registered trademarks of The MathWorks, Inc. All rights reserved. See mathworks.com/trademarks for additional information.

"Python" and the Python logos are trademarks or registered trademarks of the Python Software Foundation, and are here used in agreement with the "nominative use rules". See the Python software foundation page at www.python.org/psf/trademarks/ for additional information.

Download & setup

The SpiSeMe package can be downloaded at https://github.com/LeonardoRicci/SpiSeMe. Setup instructions are available in `README.md` files within `/code/C++/`, `/code/MATLAB/` and `/code/Python/` for the respective languages.

Package authors

Alessio Perinelli
Department of Physics, University of Trento, 38123 Trento, Italy
alessio.perinelli@unitn.it

Michele Castelluzzo
Department of Physics, University of Trento, 38123 Trento, Italy
michele.castelluzzo@unitn.it

Ludovico Minati
CIMeC, Center for Mind/Brain Sciences, University of Trento, 38068, Rovereto, Italy
FIRST, Institute of Innovative Research, Tokyo Institute of Technology, Yokohama 226-8503, Japan
Complex Systems Theory Dept., Institute of Nuclear Physics, Polish Academy of Sciences, 31-342 Kraków, Poland
lminati@ieee.org

Leonardo Ricci
Department of Physics, University of Trento, 38123 Trento, Italy
CIMeC, Center for Mind/Brain Sciences, University of Trento, 38068, Rovereto, Italy
leonardo.ricci@unitn.it
Nonlinear Systems & Electronics Lab website: nse.physics.unitn.it

2. Summary of algorithms

2.1 JODI algorithm for surrogate generation

The JODI algorithm for surrogate generation of event sequences is graphically summarized in Fig. 2.1.

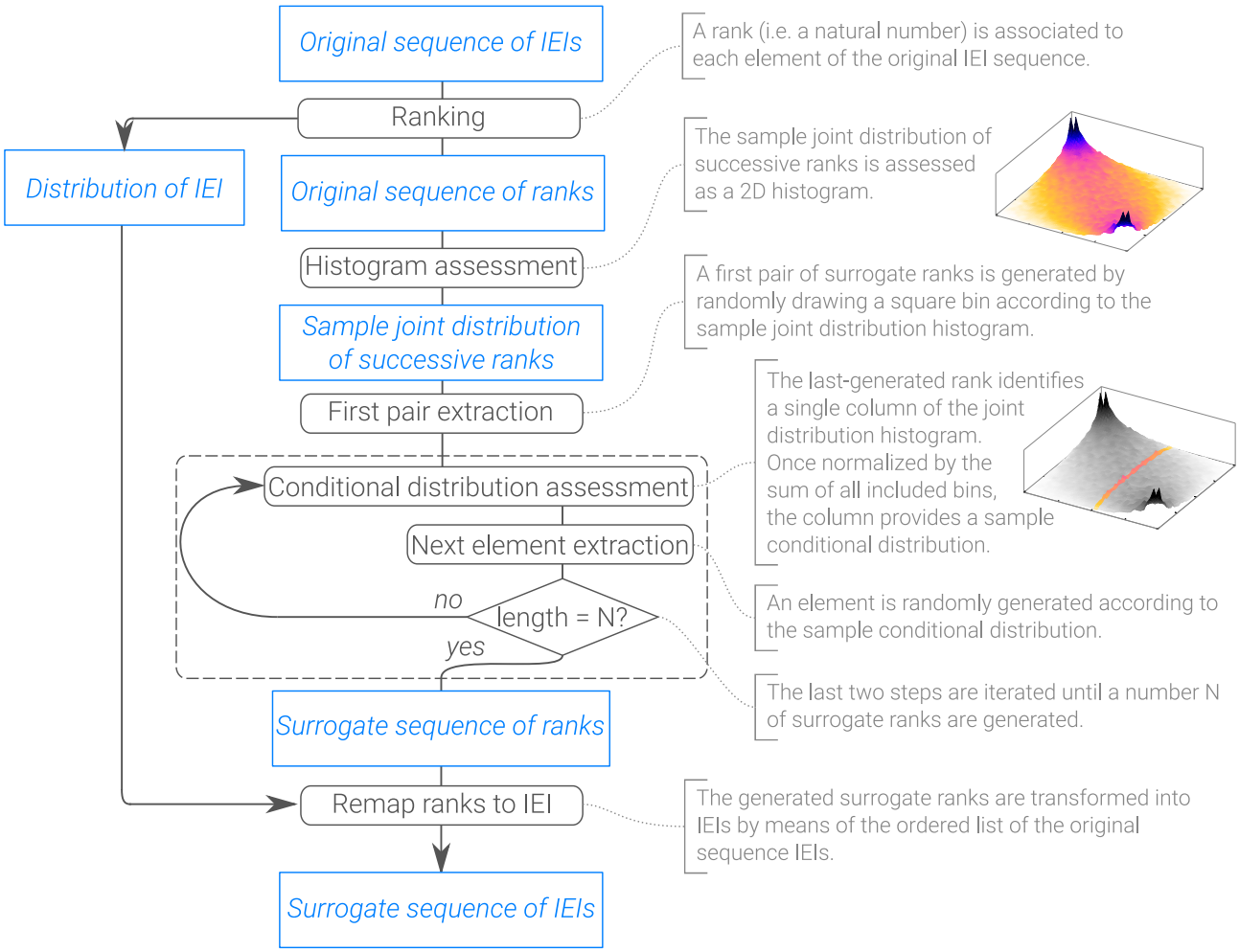


Figure 2.1: Graphical summary of JODI.

Please refer to the *main reference* for mathematical details concerning the algorithm.

The function `spiSeMe_surrogate_jodi` implements JODI:

`spiSeMe_surrogate_jodi`	C++	MATLAB	Python	Chaos **29** (2019) 121102

2.2 IAAFT algorithm for surrogate generation

The IAAFT algorithm for surrogate generation of event sequences is graphically summarized in Fig. 2.2.

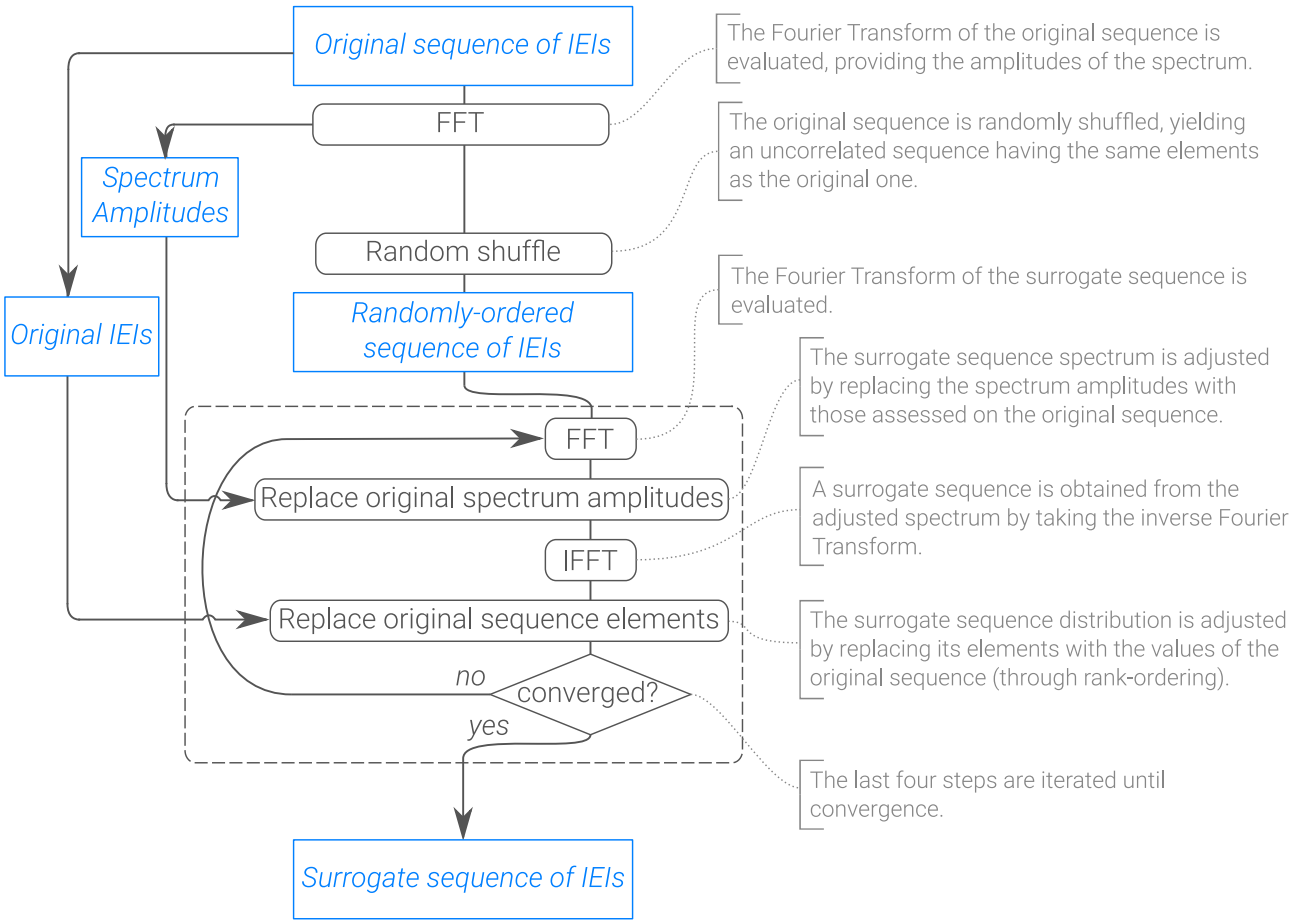


Figure 2.2: Graphical summary of IAAFT.

In the present implementation of IAAFT, convergence is acknowledged if the sequence generated at a given iteration is identical to the one generated at the previous iteration.

IAAFT can *exactly* preserve only one function between the spectrum amplitudes and the distribution of sequence values: the other function is only *approximately* preserved. The diagram of Fig. 2.2 concerns the case of the distribution of sequence values being exactly preserved. Please note that in the case of the spectrum amplitudes being exactly preserved, a negligible mismatch might still be present between the original and the surrogate spectrum amplitudes due to rounding effects in IFFT evaluation.

The function `spiSeMe_surrogate_iaaft` implements IAAFT:

`spiSeMe_surrogate_iaaft`	C++	MATLAB	Python	Phys. Rev. Lett. **77** (1996) 635

2.3 SA algorithm for surrogate generation

The SA algorithm for surrogate generation of event sequences is graphically summarized in Fig. 2.3.

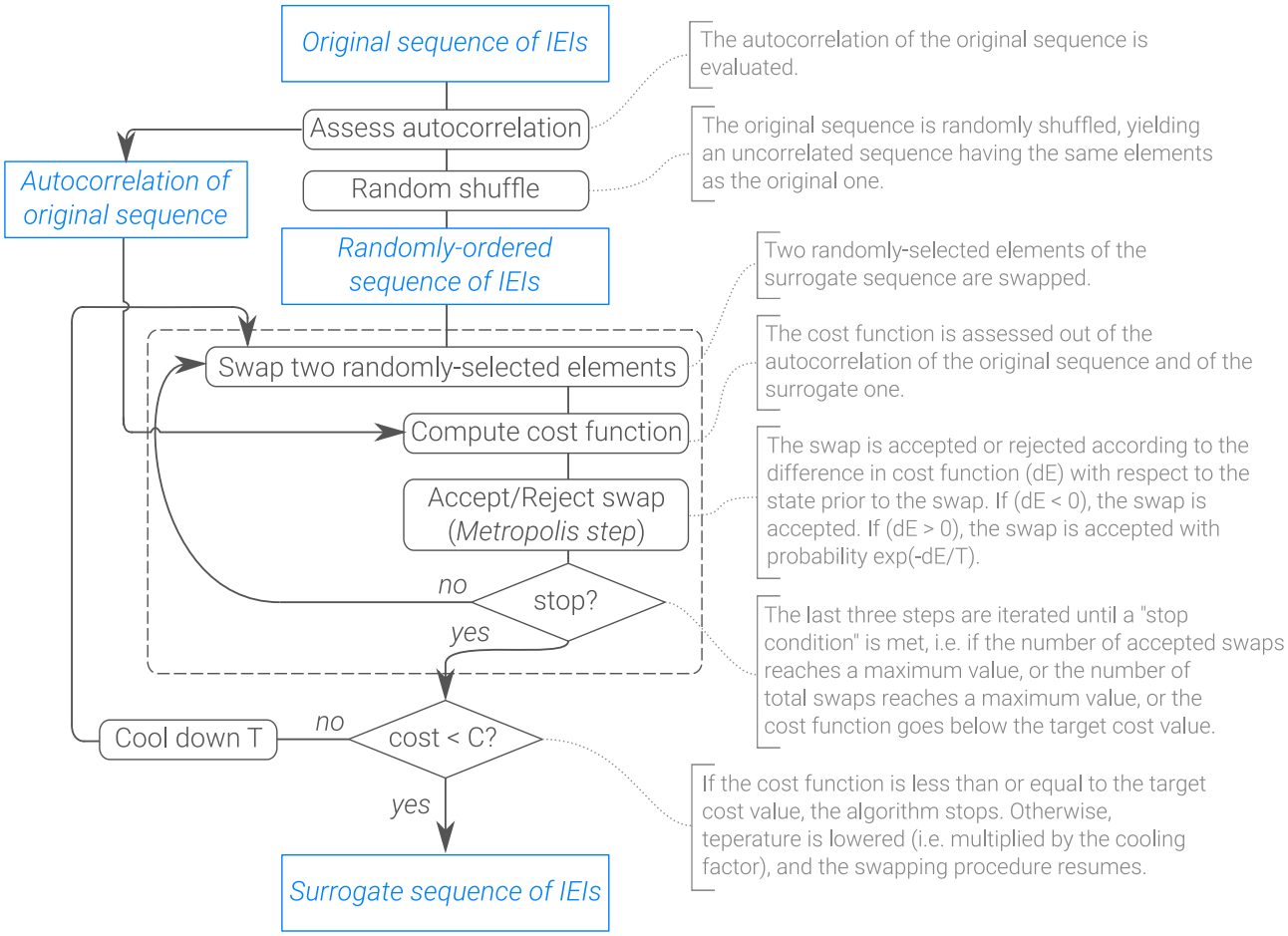


Figure 2.3: Graphical summary of SA.

The cooling schedule is determined by the starting temperature T_0 and the cooling factor α ($\alpha < 1$). At each cooling step i, temperature is lowered according to $T_i = \alpha T_{i-1}$. Cooling is carried out whenever a "stop condition" is met, namely if either the number of performed swaps or the number of accepted swaps reach the corresponding maximum values n_{total} and $n_{\text{successful}}$, respectively. The latter parameters, which are set by the user, as well as the cooling factor and starting temperature, have to be tuned according to the specific sequence: no universal recipe exists.

The target cost C determines the threshold for the cost function below which the algorithm is stopped. In the present implementation the cost function is defined so as to be minimum if the autocorrelation of the surrogate sequence (A'_k) matches the autocorrelation of the original sequence (A_k). Three alternative ways to compute the cost function are available in the SpiSeMe package:

$$\text{"max"} \quad \text{cost} = \max_k \left|A_k - A'_k\right| \qquad \text{"L1"} \quad \text{cost} = \sum_k \left|A_k - A'_k\right| \qquad \text{"L2"} \quad \text{cost} = \sum_k \left[A_k - A'_k\right]^2$$

The function `spiSeMe_surrogate_sa` implements the SA algorithm:

`spiSeMe_surrogate_sa`	C++	MATLAB	Python	Phys. Rev. Lett. **80** (1998) 2105

2.4 Dithering

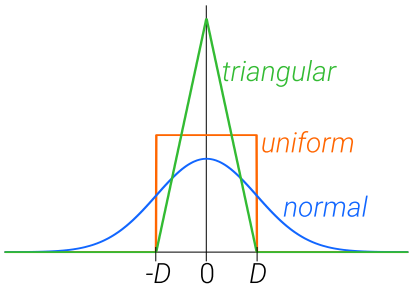
Dithering consists in randomly displacing the arrival times of events in a sequence. Assuming a sequence to be made of $N+1$ events t_i, $i \in [0, N]$, dithering yields a surrogate sequence t'_i according to

$$t'_i = t_i + \eta_i \,,$$

where η_i are realization of a random variable (*r.v.* in the following). In the present implementation, the values η_i are independent and identically distributed. The distribution followed by the r.v. has a width determined by a *"width parameter"* D: distribution shapes available in the present package are listed in the following table.

uniform	$\mathcal{U}(-D,\, D)$
normal	$\mathcal{N}(0,\, \sigma^2),\ \sigma = D$
triangular	Triang$(-D,\, D)$ (symmetric)

The figure on the right shows a comparison of the probability density functions corresponding to the three dithering distributions, given the same D parameter.



In the present implementation, if D is not specified, its value is assessed as half the minimum IEI in the sequence. This choice guarantees that the probability of the order of two events to be changed in the surrogate sequences is null (in the case of *uniform* or *triangular*).

The function `spiSeMe_surrogate_dither` implements surrogate generation via dithering:

`spiSeMe_surrogate_dither`	C++	MATLAB	Python

2.5 Assessment of event autocorrelation

Autocorrelation of a sequence of delta functions is itself a sequence of delta function, each centered at a lag value τ_{ij} corresponding to a time difference $t_i - t_j$ between the arrival of two events i, j. Because handling such autocorrelation function is cumbersome, a possible approach is to partition the lag axis in bins of width Δ and to count the number of deltas falling in each bin. This corresponds to the assessment of the histogram of all time differences $t_i - t_j$ within the sequence, with $i > j$:

$$A(\tau) = \frac{\#\left\{(i,j) \mid i > j,\ \tau - \Delta/2 \leqslant t_i - t_j < \tau + \Delta/2\right\}}{\frac{N^2\Delta}{T}},$$

where T is the sequence duration and N the number of sequence elements. The reason for the normalization is thoroughly discussed in the *main reference*. Typically, autocorrelation is evaluated up to a maximum lag $\tau_{\max}$. The lag τ is given by $(m + 1/2)\Delta$, with $0 \leqslant m \leqslant \tau_{\max}/\Delta$ and m is an integer.

The assessment of event autocorrelation is graphically summarized in Fig. 2.4.

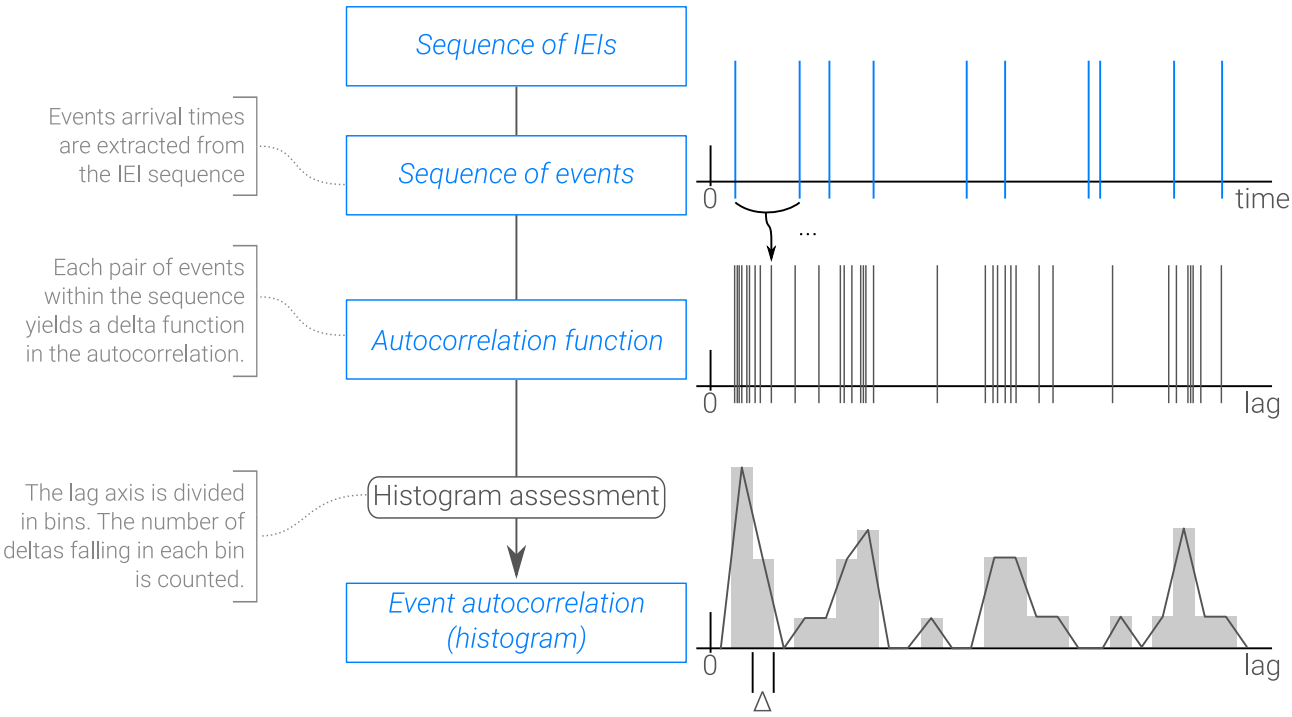


Figure 2.4: Graphical summary of the assessment of event autocorrelation.

The assessment of event autocorrelation provides two vectors of data, one corresponding to the central lag value of each bin in which the lag axis is partitioned and the other corresponding to the bin counts normalized by $N^2\Delta/T$.

The function `spiSeMe_event_autocorrelation` implements the assessment of the autocorrelation for event sequences:

`spiSeMe_event_autocorrelation`	C++	MATLAB	Python	Chaos **29** (2019) 121102

2.6 Assessment of cross correlation of event sequences

Similarly to autocorrelation, cross correlation between two sequences of delta functions is itself a sequence of delta functions, each centered at a lag value τ_{ij} corresponding to a time difference $t_{A;i} - t_{B;j}$ between the arrival of events i (belonging to sequence A) and j (belonging to sequence B). As for autocorrelation, in order to handle this function the lag axis is partitioned in bins of width Δ and the number of deltas falling in each bin is counted.

$$C(\tau) = \#\left\{(i,j) \mid \tau - \Delta/2 \leqslant t_{A;i} - t_{B;j} < \tau + \Delta/2\right\} ,$$

where the lag τ is given by $m\Delta$ and m is an integer. Because cross correlation is typically evaluated on a symmetric interval of τ bounded by $\pm\tau_{\max}$, m runs over the range given by $-\tau_{\max}/\Delta \leqslant m \leqslant \tau_{\max}/\Delta$.

The assessment of cross correlation between event sequences is graphically summarized in Fig. 2.5.

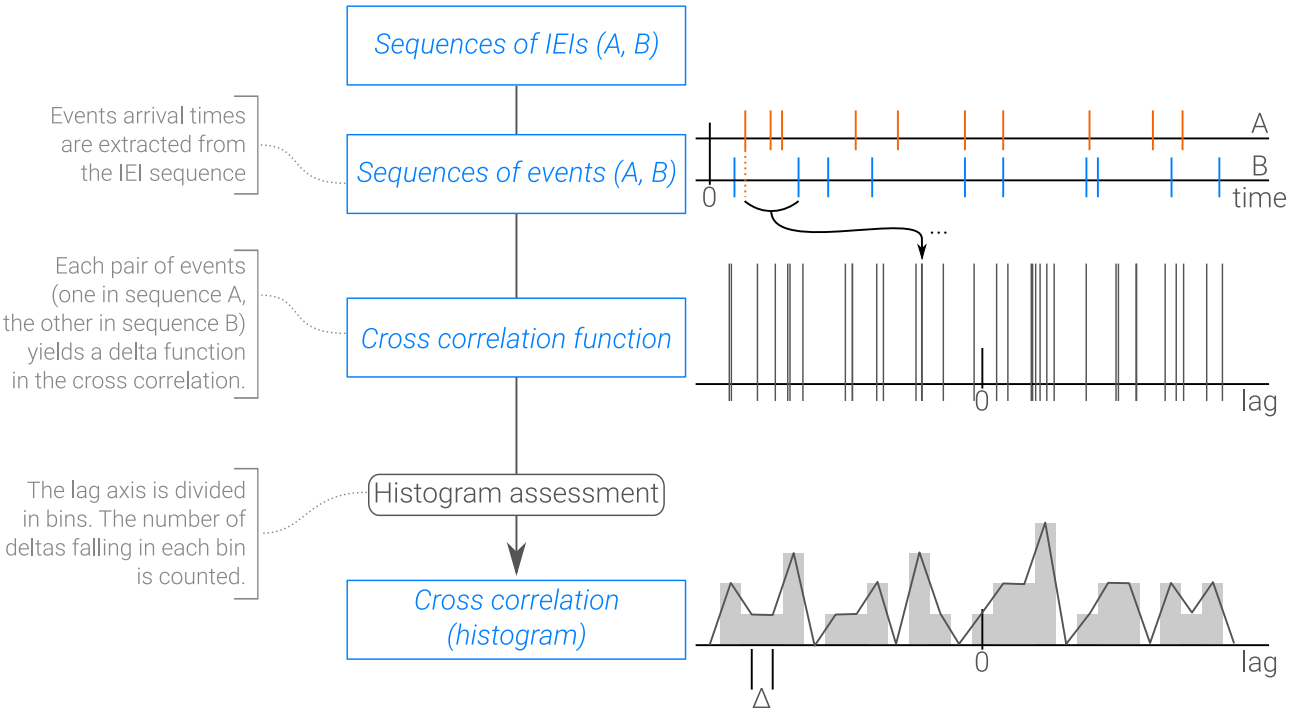


Figure 2.5: Graphical summary of the assessment of cross correlation between event sequences.

The assessment of cross correlation between event sequences provides two vectors of data, one corresponding to the central lag value of each bin in which the lag axis is partitioned and the other one corresponding to the bin counts. No normalization is applied.

The function `spiSeMe_event_cross_correlation` implements the assessment of cross correlation between event sequences:

`spiSeMe_event_cross_correlation`	C++	MATLAB	Python

2.7 Compatibility of IEI distributions

The statistical compatibility of IEI distributions associated to a pair of event sequences can be tested by means of a Kolmogorov-Smirnov test. The test (analytically) provides a p value corresponding to the null-hypothesis that the two sample IEI distributions are drawn from the same parent distribution.

The test is carried out by assessing the Kolmogorov-Smirnov statistic d out of the two cumulative IEI distributions and by evaluating the p value corresponding to d by means of the analytical expression of the Kolmogorov-Smirnov distribution. The assessment of d in the SpiSeMe package implementation relies on the algorithm presented in Section 14.3 of Numerical Recipes[1] and the evaluation of the Kolmogorov-Smirnov distribution follows the method discussed in Section 6.14 of the same reference.

The function `spiSeMe_distribution_test` implements the Kolmogorov-Smirnov test given two event sequences:

`spiSeMe_distribution_test`	C++	MATLAB	Python

[1] W. H. Press, S. A. Teukolsky, H. A. Bethe, W. T. Vetterling, B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, Third Edition (2007) Cambridge University Press ISBN-13: 978-0521880688.

3. C++ implementation

This section contains the documentation of the C++ implementation of the SpiSeMe package functions. Function declarations are contained in the header file `spiSeMe.h`.

In the following, for the sake of brevity, namespace resolution for the standard library ("`std::`") is omitted. Therefore, `std::vector` is reported as `vector`, `std::string` is reported as `string`, etc. Please notice that all source and header files always explicitely resolve the `std` namespace: the "`using namespace std`" declaration is avoided within the SpiSeMe package, in agreement with the C++ coding standard FAQ.

Surrogate generation functions are declared according to the following convention.

- **Returned value** All functions return a code (`int`) among those defined in the `spiSeMe_return_code` enumerator declaration within `spiSeMe.h` :
  - successful executions return `SSM_SUCCESS`;
  - in case of unsuitable parameters, `SSM_BAD_PARAMETER` is returned;
  - in case of runtime errors, `SSM_RUNTIME_ERR` is returned.
- **1st argument -** Output surrogate sequences are stored in a `vector< vector<double> >` object passed by reference as the first function argument. Because functions generate a number M of surrogate sequences, this object has size M and each of the contained vectors has size N, where N is the original sequence length. Please notice that any previously existing content within this object is destroyed.
- **2nd argument -** The input sequence is provided by passing a reference to a `vector<double>` object as the second function argument. This object is not modified by functions (indeed, it is `const`-qualified).
- **Subsequent arguments -** Parameters and options are defined by subsequent arguments. Some of these arguments are optional: please notice that, because argument assignment is *positional*, if an optional argument is omitted in a function call, all subsequent arguments have to be omitted as well. Default values of optional arguments are declared in the `spiSeMe.h` header file.

Auxiliary tools follow the same convention for returned values, as well as the rule of providing output data within objects whose references are passed to the function. Please refer to the related function documentation for further details.

Compiling the C++ SpiSeMe functions The SpiSeMe functions have to be compiled under the C++11 standard. For example, if the GNU `g++` compiler is used, the option `-std=c++11` has to be set. Similar options are required for different compilers.
Surrogate generation functions require the GNU Scientific Libraries in order to properly compile. Both header inclusion and library linking can be achieved by means of the `gsl-config` utility provided in GSL distributions. An example of a compilation chain using this utility is found in the `Makefile` within the `/examples/C++/` directory: compiler options are provided by calling `gsl-config --cflags` and `gsl-config --libs`.
Alternatively, GSL header files have to be manually included by means of a suitable flag (e.g. `-I/usr/include`, if GSL header files are stored in `/usr/include/gsl`). Similarly, the necessary linking flags have to be specified (e.g. `-L/usr/lib/x86_64-linux-gnu -lgsl -lgslcblas`, if GSL binaries are stored in `/usr/lib/x86_64-linux-gnu`).

3.1 Surrogate generation functions

Surrogate generation - JODI

```
spiSeMe_return_code = spiSeMe_surrogate_jodi(...)
```

Mandatory arguments:

`vector< vector<double> > & iei_surrogates`	**Reference** to output data (see above).
`const vector<double> & iei_sequence`	**Reference** to input data (see above).

Optional arguments:

`unsigned int M`	**Integer** number of (independent) surrogate sequences to generate. Default is 1.
`bool verbose`	**Boolean** value setting the verbosity of the function. If `true`, all messages are displayed; if `false`, only errors are reported. Default is `true`.

Returned value:

`spiSeMe_return_code`	**Exit status (int)** (see above).

Surrogate generation - IAAFT

```
spiSeMe_return_code = spiSeMe_surrogate_iaaft(...)
```

Mandatory arguments:

`vector< vector<double> > & iei_surrogates`	**Reference** to output data (see above).
`const vector<double> & iei_sequence`	**Reference** to input data (see above).

Optional arguments:

`string exactly_preserve`	**String** selecting which function has to be exactly preserved. It can either be `"spectrum"` (power spectral density is exactly preserved) or `"distribution"` (distribution of sequence elements is exactly preserved). Default is `"distribution"`.
`unsigned int M`	**Integer** number of (independent) surrogate sequences to generate. Default is 1.
`bool verbose`	**Boolean** value setting the verbosity of the function. If `true`, all messages are displayed; if `false`, only errors are reported. Default is `true`.

Returned value:

`spiSeMe_return_code`	**Exit status (int)** (see above).

Surrogate generation - SA

```
spiSeMe_return_code = spiSeMe_surrogate_sa(...)
```

Mandatory arguments:

`vector< vector<double> > & iei_surrogates`	**Reference** to output data (see above).
`const vector<double> & iei_sequence`	**Reference** to input data (see above).
`double autocorr_bin_width`	**Double** number corresponding to the bin width to use in the assessment of autocorrelation. This value is used in the internal calls to `spiSeMe_event_autocorrelation` required for the assessment of the cost function.
`double autocorr_max_lag`	**Double** number corresponding to the maximum lag to be considered in the assessment of autocorrelation. This value is used in the internal calls to `spiSeMe_event_autocorrelation` required for the assessment of the cost function.

Optional arguments:

`double a`	**Double** number corresponding to the cooling factor. At each cooling step, temperature is multiplied by this number. Cannot be larger than or equal to 1. Default is 0.9.
`double T`	**Double** number corresponding to the starting temperature. Default is 0.1.
`double C`	**Double** number corresponding to the target cost below which the routine stops. By default this parameter is set equal to the standard deviation of the autocorrelation of the original sequence.
`string cost_function`	**String** selecting which metric has to be used in order to compute *cost* out of the original and surrogate sequence autocorrelations. It can be `"max"`, `"L1"` or `"L2"` (see Sec. 2.3). Default is `"max"`.
`unsigned int n_total`	**Integer** corresponding to the maximum sequence elements swaps to be carried out before cooling. Default is N, where N is the sequence length.
`unsigned int n_successful`	**Integer** corresponding to the maximum successful (i.e. accepted) sequence elements swaps to be carried out before cooling. Default is $N/2$, where N is the sequence length.
`unsigned int M`	**Integer** number of (independent) surrogate sequences to generate. Default is 1.
`bool verbose`	**Boolean** value setting the verbosity of the function. If `true`, all messages are displayed; if `false`, only errors are reported. Default is `true`.

Returned value:

`spiSeMe_return_code`	**Exit status (int)** (see above).

Surrogate generation - Dithering

```
spiSeMe_return_code = spiSeMe_surrogate_dither(...)
```

Mandatory arguments:

`vector< vector<double> > & iei_surrogates`	**Reference** to output data (see above).
`const vector<double> & iei_sequence`	**Reference** to input data (see above).

Optional arguments:

`string dither_distribution`	**String** specifying from which distribution the dithering r.v. are drawn. It can be `"uniform"`, `"triangular"` or `"normal"`. Default is `"uniform"`.
`double distribution_parameter`	**Double** number setting the width parameter D of the distribution, i.e. the half-width in the cases `"uniform"` or `"triangular"`, or the standard deviation in the case of `"normal"` (see also Sec. 2.4). By default, D is set as half of the smallest IEI within the original sequence.
`unsigned int M`	**Integer** number of (independent) surrogate sequences to generate. Default is 1.
`bool verbose`	**Boolean** value setting the verbosity of the function. If `true`, all messages are displayed; if `false`, only errors are reported. Default is `true`.

Returned value:

`spiSeMe_return_code`	**Exit status (int)** (see above).

3.2 Auxiliary analytical tools

Assessment of event sequences autocorrelation

```
spiSeMe_return_code = spiSeMe_event_autocorrelation(...)
```

Mandatory arguments:

`vector<double> & A`
Reference to a vector object where the autocorrelation amplitudes have to be stored. The autocorrelation amplitude at each bin is assessed as the bin count normalized by $N^2 \cdot \delta\tau / T$, where N is the number of IEI in the input sequence and T is the sequence duration (i.e. the sum of all IEIs). See Sec. 2.5 for details.

`vector<double> & lags`
Reference to a vector object where the central lag value of each bin has to be stored. For example, the zeroth element of `lags` is $\delta\tau/2$, i.e. the central lag value of the first bin that covers the range $(0, \delta\tau]$. The number of bins is given by

$$\texttt{n\_bins} = \mathrm{ceil}\left[\frac{\texttt{max\_lag}}{\delta\tau}\right].$$

Consequently, the maximum lag included in the autocorrelation assessment is $\texttt{n\_bins} \cdot \delta\tau$, which is larger than or equal to `max_lag`: equality holds if `max_lag` is an integer multiple of $\delta\tau$.

`const vector<double> & iei_sequence`
Reference to input data (see above).

`double bin_width`
Double number corresponding to the width $\delta\tau$ of the bins in which the lag axis is partitioned.

`double max_lag`
Double number corresponding to the maximum lag to be considered in the autocorrelation assessment. This value has to be larger than $\delta\tau$.

Returned value:

`spiSeMe_return_code`
Exit status (int) (see above).

Assessment of cross correlation between event sequences

```
spiSeMe_return_code = spiSeMe_event_cross_correlation(...)
```

Mandatory arguments:

`vector<double> & C` — **Reference** to a vector object where the cross correlation amplitudes have to be stored. Please notice that any previously existing content within this object is lost. The cross correlation amplitude at each bin is given by the bin count (no normalization is applied).

`vector<double> & lags` — **Reference** to a vector object where the central lag value of each bin has to be stored. The zeroth element of the vector corresponds to the bin covering the most negative values of τ, namely the bin covering the range $[-\frac{\texttt{n\_bins}}{2} \cdot \delta\tau, -\frac{\texttt{n\_bins}-1}{2} \cdot \delta\tau)$. The number of bins is given by

$$\texttt{n\_bins} = 2 \cdot \mathrm{ceil}\left[\frac{\texttt{max\_lag}}{\delta\tau}\right] + 1\,.$$

Consequently, the lags included in the cross correlation assessment cover the range $\pm\frac{\texttt{n\_bins}}{2} \cdot \delta\tau$.

`const vector<double> & iei_sequence_A` — **Reference** to the first input IEI sequence.

`const vector<double> & iei_sequence_B` — **Reference** to the second input IEI sequence.

`double bin_width` — **Double** number corresponding to the width $\delta\tau$ of the bins in which the lag axis is partitioned.

`double max_lag` — **Double** number corresponding to the maximum (in absolute value) lag to be considered in the autocorrelation assessment. This value has to be larger than $\delta\tau$.

Note that time differences are computed as $t_{A;i} - t_{B;j}$ (see also Sec. 2.6)*. Exchanging the input sequences is equivalent to a change in sign of the lag values.*

Returned value:

`spiSeMe_return_code` — **Exit status (int)** (see above).

Assessment of statistical compatibility of IEI distributions

```
spiSeMe_return_code = spiSeMe_distribution_test(...)
```

Mandatory arguments:

`double & p`	**Reference** to a double number where to store the p value provided by the Kolmogorov-Smirnov test carried out between the IEI distributions of the two input sequences.
`double & d`	**Reference** to a double number where to store the value of the Kolmogorov-Smirnov statistic statistic between the IEI distributions of the two input sequences (i.e. the maximum absolute discrepancy between the corresponding cumulative distributions).
`const vector<double> & iei_sequence_A`	**Reference** to the first input IEI sequence.
`const vector<double> & iei_sequence_B`	**Reference** to the second input IEI sequence.

Returned value:

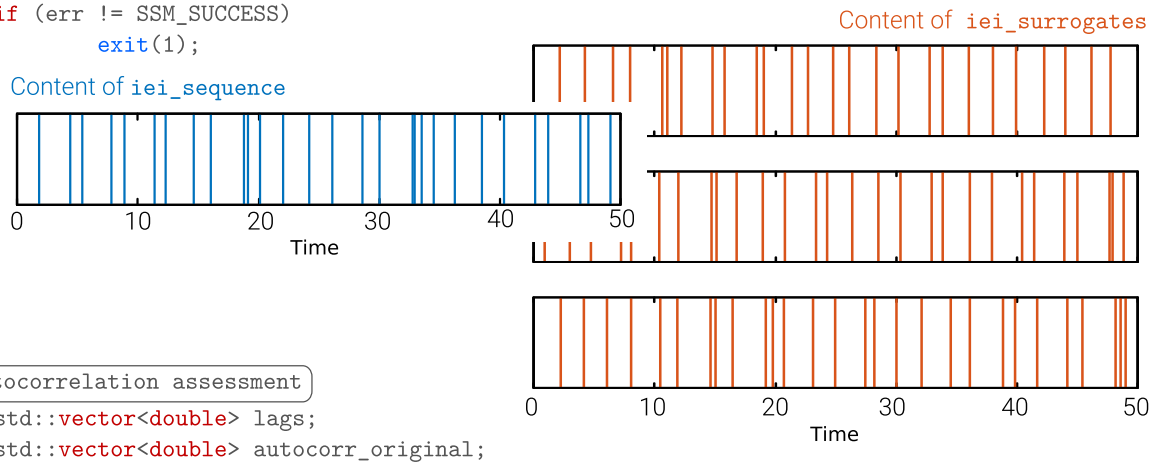
`spiSeMe_return_code`	**Exit status (int)** (see above).

3.3 Examples

The `/examples/C++/` directory stores two C++ example programs containing calls to the SpiSeMe functions. Four example sequences are provided in the `/examples/data/` directory. A `README` file within the same folder provides instructions to compile and run the programs on Linux machines by means of the GNU Compiler Collection (`g++`). The first program contains the code required to load data from file, to generate surrogates by means of different methods, as well as to compute and store autocorrelation. Example code and data are shown in Fig. 3.1; surrogates are generated by means of JODI out of the `iei_henon.dat` sequence.

Surrogate generation

```
spiSeMe_return_code err;
std::vector< std::vector <double> > iei_surrogates;
err = spiSeMe_surrogate_jodi(iei_surrogates, iei_sequence, 3, true);
if (err != SSM_SUCCESS)
        exit(1);
```



Autocorrelation assessment

```
std::vector<double> lags;
std::vector<double> autocorr_original;
std::vector<double> autocorr_surrogate;
err = spiSeMe_event_autocorrelation(autocorr_original, lags, iei_sequence, 0.06, 21);
if (err != SSM_SUCCESS) exit(1);
err = spiSeMe_event_autocorrelation(autocorr_surrogate, lags, iei_surrogates[0], 0.06, 21);
if (err != SSM_SUCCESS) exit(1);
```

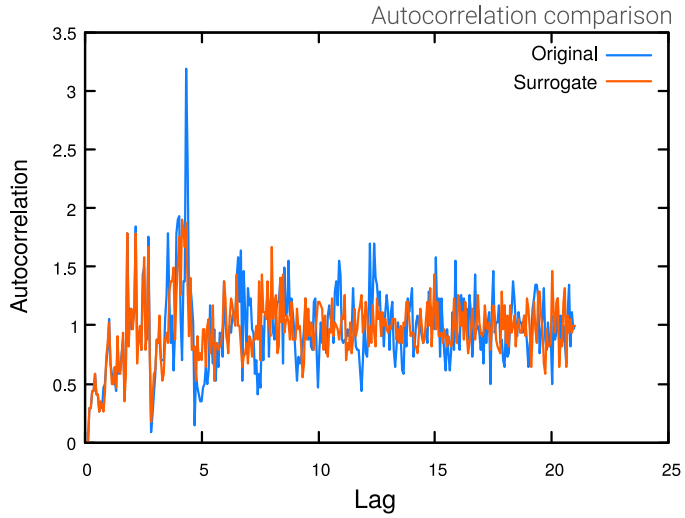


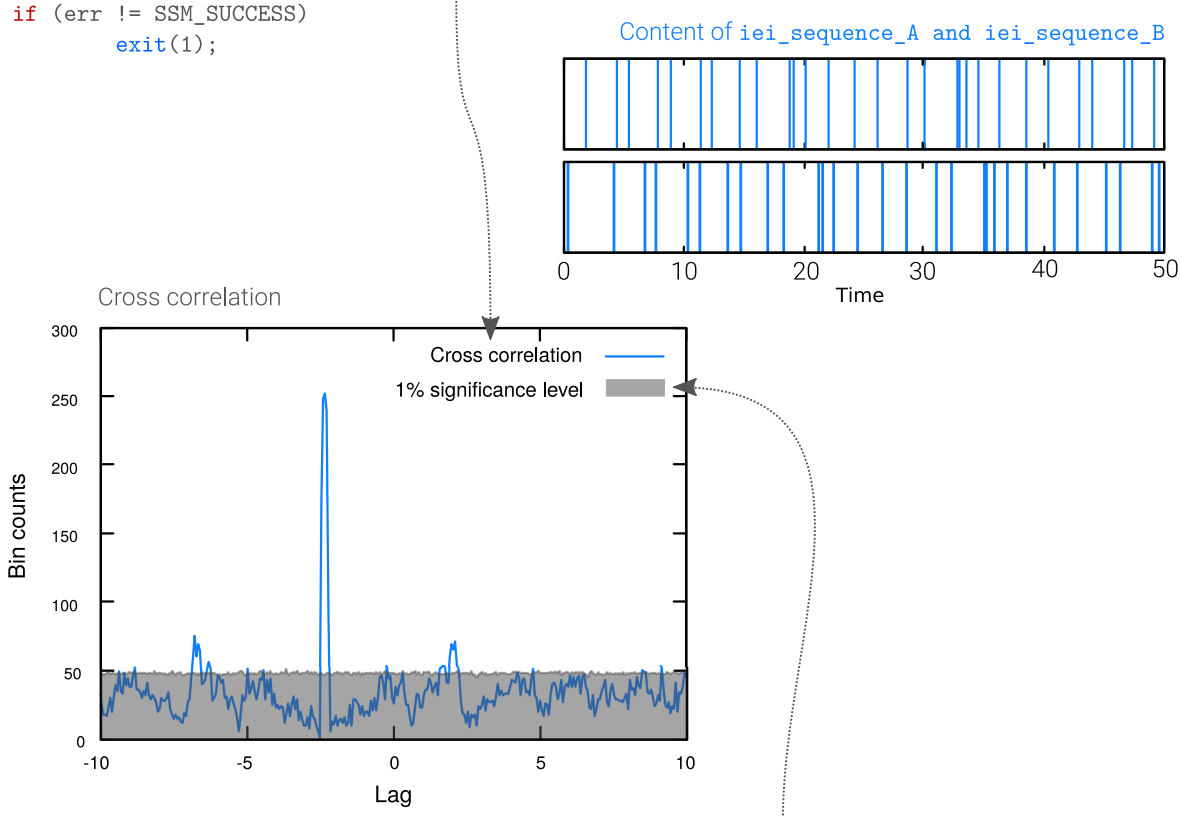
Figure 3.1: Excerpts of the sequence stored in `iei_henon.dat` and of three corresponding surrogate sequences generated by means of JODI. The autocorrelation of the original sequence and of a surrogate one are also shown. See `/examples/C++/example.cpp` for the code generating these data.

The second program provides an example of the assessment of cross correlation between event sequences and the related significance estimation. This estimation is carried out by generating 500 sur-

rogate sequences by means of JODI. Figure 3.2 shows excerpts of the original sequences, the related cross correlation and the threshold corresponding to the 1% significance level.

Cross correlation between original sequences

```
spiSeMe_return_code err;
std::vector <double> C;
std::vector <double> lags;
err = spiSeMe_event_cross_correlation(C, lags, iei_sequence_A, iei_sequence_B, 0.06, 10.0);
if (err != SSM_SUCCESS)
        exit(1);
```



Significance level assessment

```
unsigned int M = 500;
std::vector<double> lags;
std::vector< std::vector<double> > C_surr(M);
std::vector< std::vector<double> > iei_surrogates_A;
std::vector< std::vector<double> > iei_surrogates_B;
err = spiSeMe_surrogate_jodi(iei_surrogates_A, iei_sequence_A, M);
if (err != SSM_SUCCESS) exit(1);
err = spiSeMe_surrogate_jodi(iei_surrogates_B, iei_sequence_B, M);
if (err != SSM_SUCCESS) exit(1);
for (int i = 0; i < M; i++) {
    err = spiSeMe_event_cross_correlation(C_surr[i], lags, iei_surrogate_A[i],
                                          iei_surrogate_B[i], 0.06, 10.0);
    if (err != SSM_SUCCESS) exit(1);
}
// Sort C_surr values corresponding to each bin and select the m-th largest
//    according to the significance level.
```

Figure 3.2: Excerpt of the sequences stored in `iei_henon.dat` and `iei_henon_bis.dat`. The cross correlation between the two sequences, as well as the threshold corresponding to the 1% significance level, are shown. See `/examples/C++/example_cross_correlation.cpp` for the code generating these data.

4. MATLAB implementation

This section contains the documentation of the MATLAB implementation of the SpiSeMe package functions. Function signatures are reported in gray boxes, followed by the list of arguments and the list of returned values. In function signatures only mandatory arguments are explicitely listed, while "`...`" denotes optional arguments. In function calls, optional arguments have to be listed as name-value pairs, where the argument name is surrounded by quotes (`..., 'Name', value, ...`).

4.1 Surrogate generation functions

Surrogate generation - JODI

```
ieiSurrogates = spiSeMe_surrogate_jodi(ieiSequence, ...)
```

Mandatory arguments:

`ieiSequence` — **Array** containing the input IEI sequence. The array has to be one-dimensional (size $1 \times N$ or $N \times 1$).

Optional arguments:

`M` — **Number** of (independent) surrogate sequences to generate. Default is 1.

`verbose` — **Logical** value setting the verbosity of the function. If `true`, all messages are displayed; if `false`, only errors are reported. Default is `true`.

Returned value:

`ieiSurrogates` — **Array** containing the generated surrogate IEI sequence(s). The array has size $N \times M$: each of its columns corresponds to one of the generated surrogate IEI sequences.

Surrogate generation - IAAFT

```
ieiSurrogates = spiSeMe_surrogate_iaaft(ieiSequence, ...)
```

Mandatory arguments:

`ieiSequence`	**Array** containing the input IEI sequence. The array has to be one-dimensional (size $1 \times N$ or $N \times 1$).

Optional arguments:

`exactlyPreserve`	**String** selecting which function has to be exactly preserved. It can either be `'spectrum'` (power spectral density is exactly preserved) or `'distribution'` (distribution of sequence elements is exactly preserved). Default is `'distribution'`.
`M`	**Number** of (independent) surrogate sequences to generate. Default is 1.
`verbose`	**Logical** value setting the verbosity of the function. If `true`, all messages are displayed; if `false`, only errors are reported. Default is `true`.

Returned value:

`ieiSurrogates`	**Array** containing the generated surrogate IEI sequence(s). The array has size $N \times M$: each of its columns corresponds to one of the generated surrogate IEI sequences.

Surrogate generation - SA

```
ieiSurrogates = spiSeMe_surrogate_sa(ieiSequence, autocorrBinWidth, autocorrMaxLag,
                                     ...)
```

Mandatory arguments:

`ieiSequence`	**Array** containing the input IEI sequence. The array has to be one-dimensional (size $1 \times N$ or $N \times 1$).
`autocorrBinWidth`	**Number** corresponding to the bin width to use in the assessment of autocorrelation. This value is used in the internal calls to `spiSeMe_event_autocorrelation` required for the assessment of the cost function.
`autocorrMaxLag`	**Number** corresponding to the maximum lag to be considered in the assessment of autocorrelation. This value is used in the internal calls to `spiSeMe_event_autocorrelation` required for the assessment of the cost function.

Optional arguments:

`a`	**Number** corresponding to the cooling factor. At each cooling step, temperature is multiplied by this number. Cannot be larger than or equal to 1. Default is 0.9.
`T`	**Number** corresponding to the starting temperature. Default is 0.1.
`C`	**Number** corresponding to the target cost below which the routine stops. By default this parameter is set equal to the standard deviation of the autocorrelation of the original sequence.
`costFunction`	**String** selecting which metric has to be used in order to compute *cost* out of the original and surrogate sequence autocorrelations. It can be either `'max'`, `'L1'` or `'L2'` (see Sec. 2.3). Default is `'max'`.
`nTotal`	**Number** corresponding to the maximum sequence elements swaps to be carried out before cooling. Default is N, where N is the sequence length.
`nSuccessful`	**Number** corresponding to the maximum successful (i.e. accepted) sequence elements swaps to be carried out before cooling. Default is $N/2$, where N is the sequence length.
`M`	**Number** of (independent) surrogate sequences to generate. Default is 1.
`verbose`	**Logical** value setting the verbosity of the function. If `true`, all messages are displayed; if `false`, only errors are reported. Default is `true`.

Returned value:

`ieiSurrogates`	**Array** containing the generated surrogate IEI sequence(s). The array has size $N \times M$: each of its columns corresponds to one of the generated surrogate IEI sequences.

Surrogate generation - Dithering

```
ieiSurrogates = spiSeMe_surrogate_dithering(ieiSequence, ...)
```

Mandatory arguments:

`ieiSequence`	**Array** containing the input IEI sequence. The array has to be one-dimensional (size $1 \times N$ or $N \times 1$).

Optional arguments:

`ditherDistribution`	**String** specifying from which distribution the dithering r.v. are drawn. It can be `'uniform'`, `'triangular'` or `'normal'`. Default is `'uniform'`.
`distributionParameter`	**Number** setting the width parameter D of the distribution, i.e. the half-width in the cases `'uniform'` or `'triangular'`, or the standard deviation in the case of `'normal'` (see also Sec. 2.4). By default, D is set as half of the smallest IEI within the original sequence.
`M`	**Number** of (independent) surrogate sequences to generate. Default is 1.
`verbose`	**Logical** value setting the verbosity of the function. If `true`, all messages are displayed; if `false`, only errors are reported. Default is `true`.

Returned value:

`ieiSurrogates`	**Array** containing the generated surrogate IEI sequence(s). The array has size $N \times M$: each of its columns corresponds to one of the generated surrogate IEI sequences.

4.2 Auxiliary analytical tools

Assessment of event sequences autocorrelation

```
[A, lags] = spiSeMe_event_autocorrelation(ieiSequence, binWidth, maxLag)
```

Mandatory arguments:

`ieiSequence`	**Array** containing the input IEI sequence. The array has to be one-dimensional (size $1 \times N$ or $N \times 1$).
`binWidth`	**Number** corresponding to the width $\delta\tau$ of the bins in which the lag axis is partitioned.
`maxLag`	**Number** corresponding to the maximum lag to be considered in the autocorrelation assessment. This value has to be larger than $\delta\tau$.

Returned values:

`A`	**Array** (size $\texttt{nBins} \times 1$) containing the autocorrelation amplitudes. The autocorrelation amplitude at each bin is assessed as the bin count normalized by $N^2\delta\tau/T$, where N is the number of IEI in the input sequence and T is the sequence duration (i.e. the sum of all IEIs). See Sec. 2.5 for details.
`lags`	**Array** (size $\texttt{nBins}\times 1$) containing the central lag value of each bin. For example, the first element of `lags` is $\delta\tau/2$, i.e. the central lag value of the bin that covers the range $(0, \delta\tau]$. The number of bins is given by $$\texttt{nBins} = \text{ceil}\left[\frac{\texttt{maxLag}}{\delta\tau}\right].$$ Consequently, the maximum lag included in the autocorrelation assessment is $\texttt{nBins} \cdot \delta\tau$, which is larger than or equal to `maxLag`: equality holds if `maxLag` is an integer multiple of $\delta\tau$.

Assessment of cross correlation between event sequences

```
[C, lags] = spiSeMe_event_cross_correlation(ieiSequenceA,ieiSequenceB,
                                            binWidth, maxLag)
```

Mandatory arguments:

`ieiSequenceA`	**Array** containing the first input IEI sequence. The array has to be one-dimensional (size $1 \times N$ or $N \times 1$).
`ieiSequenceB`	**Array** containing the second input IEI sequence. The array has to be one-dimensional (size $1 \times N$ or $N \times 1$).
`binWidth`	**Number** corresponding to the width $\delta\tau$ of the bins in which the lag axis is partitioned.
`maxLag`	**Number** corresponding to the maximum (in absolute value) lag to be considered in the cross correlation assessment. This value has to be larger than $\delta\tau$.

Note that time differences are computed as $t_{A;i} - t_{B;j}$ (see also Sec. 2.6). *Exchanging the input sequences is equivalent to a change in sign of the lag values*.

Returned values:

`C`	**Array** (size $\texttt{nBins} \times 1$) containing the cross correlation amplitudes for each bin, given by the bin count (no normalization is applied).
`lags`	**Array** (size $\texttt{nBins} \times 1$) containing the central lag value of each bin. The first element of the array corresponds to the bin concerning the most negative values of τ, namely the bin covering the range $[-\frac{\texttt{nBins}}{2} \cdot \delta\tau, -\frac{\texttt{nBins}-1}{2} \cdot \delta\tau)$. The number of bins is given by $$\texttt{nBins} = 2 \cdot \mathrm{ceil}\left[\frac{\texttt{maxLag}}{\delta\tau}\right] + 1\,.$$ Consequently, the lags included in the cross correlation assessment cover the range $\pm\frac{\texttt{nBins}}{2} \cdot \delta\tau$.

Assessment of statistical compatibility of IEI distributions

```
[p, d] = spiSeMe_distribution_test(ieiSequenceA, ieiSequenceB)
```

Mandatory arguments:

`ieiSequenceA`	**Array** containing the first input IEI sequence. The array has to be one-dimensional (size $1 \times N$ or $N \times 1$).
`ieiSequenceB`	**Array** containing the second input IEI sequence. The array has to be one-dimensional (size $1 \times N$ or $N \times 1$).

Returned values:

`p`	**Number** corresponding to the p value provided by the Kolmogorov-Smirnov test carried out between the IEI distributions of the two input sequences.
`d`	**Number** corresponding to the assessed Kolmogorov-Smirnov statistic between the IEI distributions of the two input sequences (i.e. the maximum absolute discrepancy between the corresponding cumulative distributions).

4.3 Examples

The `/examples/MATLAB/` directory stores two MATLAB example scripts containing calls to the SpiSeMe functions. Four example sequences are provided in the `/examples/data/` directory. The first script contains the code required to load data from file, to generate surrogates by means of different methods, as well as to display sequence excerpts and autocorrelation plots. Examples of these plots are shown in Fig. 4.1 in the case of surrogates generated by means of JODI out of the `iei_henon.dat` sequence.

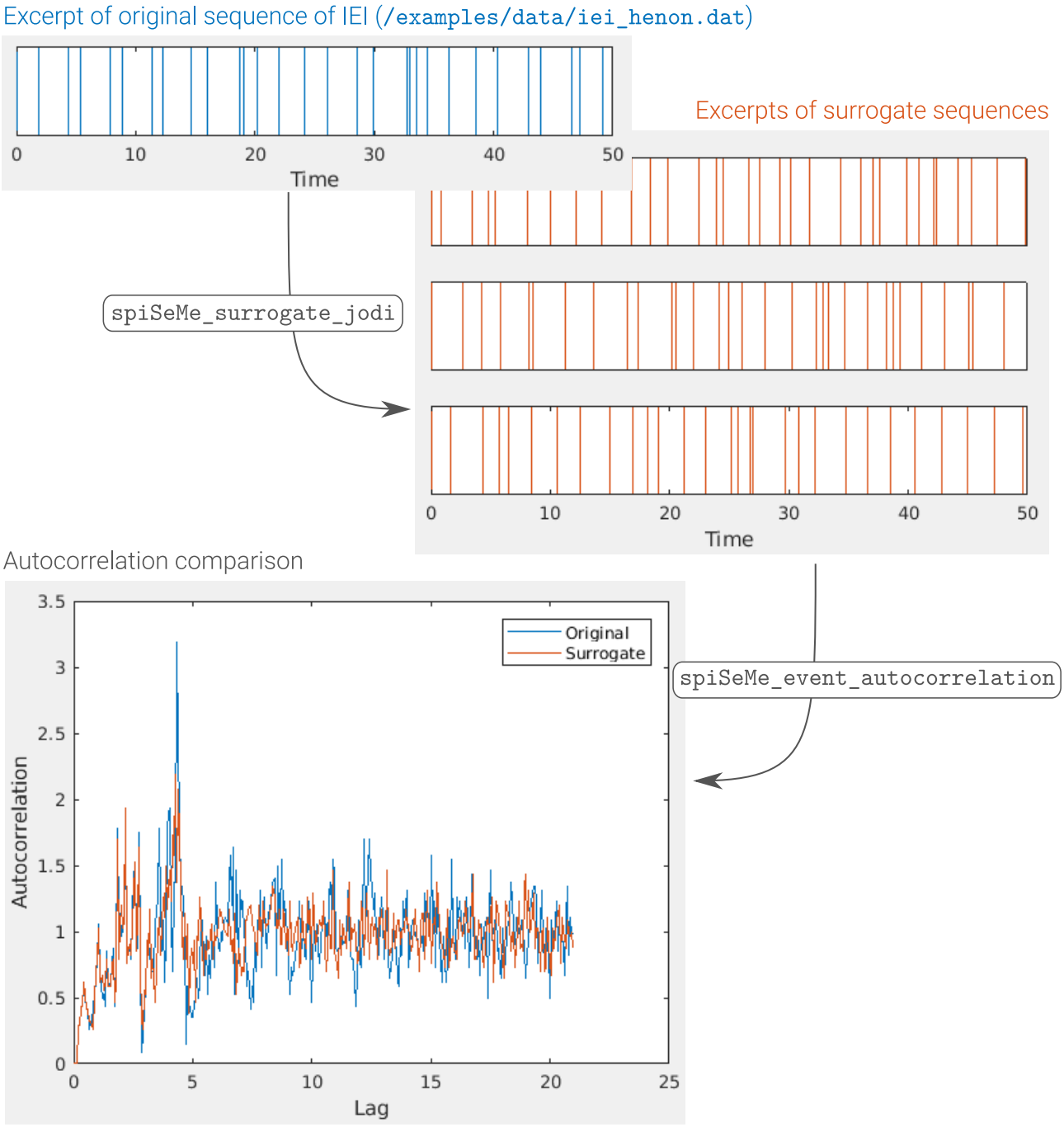


Figure 4.1: Excerpts of the sequence stored in `iei_henon.dat` and of three corresponding surrogate sequences generated by means of JODI. The autocorrelation of the original sequence and of a surrogate one are also shown. See `/examples/MATLAB/example.m` for the code generating these plots.

The second script provides an example of the assessment of cross correlation between event se-

quences and the related significance estimation. This estimation is carried out by generating 500 surrogate sequences by means of JODI. Figure 4.2 shows excerpts of the original sequences, the related cross correlation and the threshold corresponding to the 1% significance level.

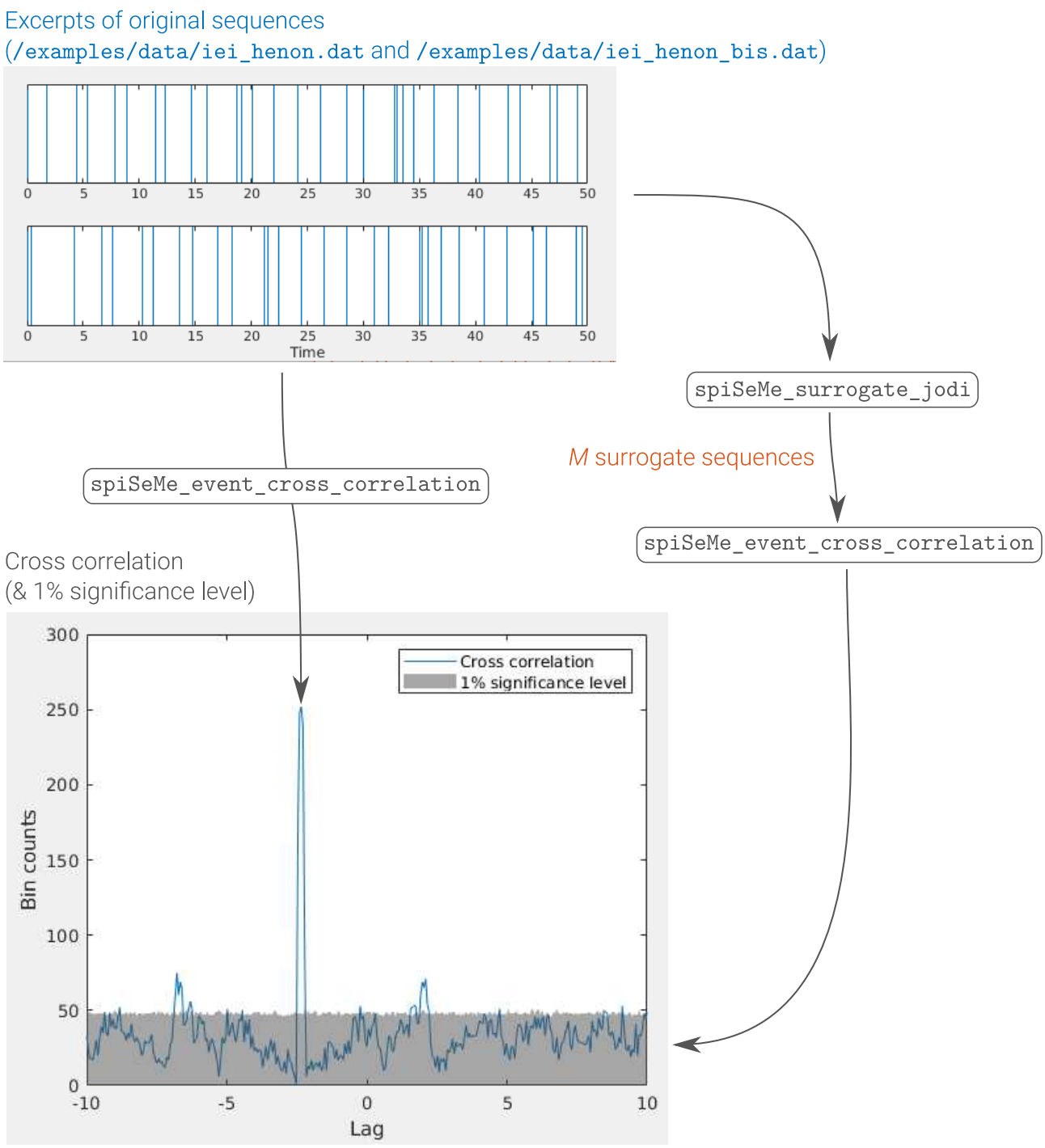


Figure 4.2: Excerpt of the sequences stored in `iei_henon.dat` and `iei_henon_bis.dat`. The cross correlation between the two sequences, as well as the threshold corresponding to the 1% significance level, are shown. See `/examples/MATLAB/example_cross_correlation.m` for the code generating these plots.

5. Python implementation

This section contains the documentation of the Python implementation of the SpiSeMe package functions. Function signatures are reported in gray boxes, followed by the list of arguments and the list of returned values. In function signatures only mandatory arguments are explicitely listed, while "`...`" denotes optional arguments. In function calls optional arguments can be either provided in the same order as in the related list (positional assignment), or they can be specified as name-value pairs (`..., name=value, ...`).

5.1 Surrogate generation functions

Surrogate generation - JODI

```
iei_surrogates = spiSeMe_surrogate_jodi(iei_sequence, ...)
```

Mandatory arguments:

`iei_sequence` **NumPy array** containing the input IEI sequence. The array has to be one-dimensional.

Optional arguments:

`M` **Number** of (independent) surrogate sequences to generate. Default is 1.

`verbose` **Boolean** value setting the verbosity of the function. If `True`, all messages are displayed; if `False`, only errors are reported. Default is `True`.

Returned value:

`iei_surrogates` **NumPy array** containing the generated surrogate IEI sequence(s). The array has size $M \times N$: each of its rows corresponds to one of the generated surrogate IEI sequences.

Surrogate generation - IAAFT

```
iei_surrogates = spiSeMe_surrogate_iaaft(iei_sequence, ...)
```

Mandatory arguments:

`iei_sequence`	**NumPy array** containing the input IEI sequence. The array has to be one-dimensional.

Optional arguments:

`exactly_preserve`	**String** selecting which function has to be exactly preserved. It can either be `'spectrum'` (power spectral density is exactly preserved) or `'distribution'` (distribution of sequence elements is exactly preserved). Default is `'distribution'`.
`M`	**Number** of (independent) surrogate sequences to generate. Default is 1.
`verbose`	**Boolean** value setting the verbosity of the function. If `True`, all messages are displayed; if `False`, only errors are reported. Default is `True`.

Returned value:

`iei_surrogates`	**NumPy array** containing the generated surrogate IEI sequence(s). The array has size $M \times N$: each of its rows corresponds to one of the generated surrogate IEI sequences.

Surrogate generation - SA

```
iei_surrogates = spiSeMe_surrogate_sa(iei_sequence, autocorr_bin_width,
                                      autocorr_max_lag, ...)
```

Mandatory arguments:

`iei_sequence`	**NumPy array** containing the input IEI sequence. The array has to be one-dimensional.
`autocorr_bin_width`	**Number** corresponding to the bin width to use in the assessment of autocorrelation. This value is used in the internal calls to `spiSeMe_event_autocorrelation` required for the assessment of the cost function.
`autocorr_max_lag`	**Number** corresponding to the maximum lag to be considered in the assessment of autocorrelation. This value is used in the internal calls to `spiSeMe_event_autocorrelation` required for the assessment of the cost function.

Optional arguments:

`a`	**Number** corresponding to the cooling factor. At each cooling step, temperature is multiplied by this number. Cannot be larger than or equal to 1. Default is 0.9.
`T`	**Number** corresponding to the starting temperature. Default is 0.1.
`C`	**Number** corresponding to the target cost below which the routine stops. By default this parameter is set equal to the standard deviation of the autocorrelation of the original sequence.
`cost_function`	**String** selecting which metric has to be used in order to compute *cost* out of the original and surrogate sequence autocorrelations. It can be `'max'`, `'L1'` or `'L2'` (see Sec. 2.3). Default is `'max'`.
`n_total`	**Number** corresponding to the maximum sequence elements swaps to be carried out before cooling. Default is N, where N is the sequence length.
`n_successful`	**Number** corresponding to the maximum successful (i.e. accepted) sequence elements swaps to be carried out before cooling. Default is $N/2$, where N is the sequence length.
`M`	**Number** of (independent) surrogate sequences to generate. Default is 1.
`verbose`	**Boolean** value setting the verbosity of the function. If `True`, all messages are displayed; if `False`, only errors are reported. Default is `True`.

Returned value:

`iei_surrogates`	**NumPy array** containing the generated surrogate IEI sequence(s). The array has size $M \times N$: each of its rows corresponds to one of the generated surrogate IEI sequences.

Surrogate generation - Dithering

```
iei_surrogates = spiSeMe_surrogate_dither(iei_sequence, ...)
```

Mandatory arguments:

`iei_sequence`	**NumPy array** containing the input IEI sequence. The array has to be one-dimensional.

Optional arguments:

`dither_distribution`	**String** specifying from which distribution the dithering r.v. are drawn. It can be `'uniform'`, `'triangular'` or `'normal'`. Default is `'uniform'`.
`distribution_parameter`	**Number** setting the width parameter D of the distribution, i.e. the half-width in the cases `'uniform'` or `'triangular'`, or the standard deviation in the case of `'normal'` (see also Sec. 2.4). By default, D is set as half of the smallest IEI within the original sequence.
`M`	**Number** of (independent) surrogate sequences to generate. Default is 1.
`verbose`	**Boolean** value setting the verbosity of the function. If `True`, all messages are displayed; if `False`, only errors are reported. Default is `True`.

Returned value:

`iei_surrogates`	**NumPy array** containing the generated surrogate IEI sequence(s). The array has size $M \times N$: each of its rows corresponds to one of the generated surrogate IEI sequences.

5.2 Auxiliary analytical tools

Assessment of event sequences autocorrelation

```
A, lags = spiSeMe_event_autocorrelation(iei_sequence, bin_width, max_lag)
```

Mandatory arguments:

`iei_sequence`	**NumPy array** containing the input IEI sequence. The array has to be one-dimensional.
`bin_width`	**Number** corresponding to the width $\delta\tau$ of the bins in which the lag axis is partitioned.
`max_lag`	**Number** corresponding to the maximum lag to be considered in the autocorrelation assessment. This value has to be larger than $\delta\tau$.

Returned values:

`A`	**NumPy array** (one-dimensional) containing the autocorrelation amplitudes. The autocorrelation amplitude at each bin is assessed as the bin count normalized by $N^2 \cdot \delta\tau / T$, where N is the number of IEI in the input sequence and T is the sequence duration (i.e. the sum of all IEIs). See Sec. 2.5 for details.
`lags`	**NumPy array** (one-dimensional) containing the central lag value of each bin. For example, the zeroth element of `lags` is $\delta\tau/2$, i.e. the central lag value of the bin that covers the range $(0, \delta\tau]$. The number of bins is given by $$\texttt{n\_bins} = \mathrm{ceil}\left[\frac{\texttt{max\_lag}}{\delta\tau}\right].$$ Consequently, the maximum lag included in the autocorrelation assessment is $\texttt{n\_bins}\cdot\delta\tau$, which is larger than or equal to `max_lag`: equality holds if `max_lag` is an integer multiple of $\delta\tau$.

Assessment of cross correlation between event sequences

```
C, lags = spiSeMe_event_cross_correlation(iei_sequence_A, iei_sequence_B,
                                          bin_width, max_lag)
```

Mandatory arguments:

`iei_sequence_A`	**NumPy array** containing the first input IEI sequence. The array has to be one-dimensional.
`iei_sequence_B`	**NumPy array** containing the second input IEI sequence. The array has to be one-dimensional.
`bin_width`	**Number** corresponding to the width $\delta\tau$ of the bins in which the lag axis is partitioned.
`max_lag`	**Number** corresponding to the maximum (in absolute value) lag to be considered in the autocorrelation assessment. This value has to be larger than $\delta\tau$.

Note that time differences are computed as $t_{A;i} - t_{B;j}$ (see also Sec. 2.6). *Exchanging the input sequences is equivalent to a change in sign of the lag variable.*

Returned values:

`C`	**NumPy array** (one-dimensional) containing the cross correlation amplitudes for each bin, given by the bin count (no normalization is applied).
`lags`	**NumPy array** (one-dimensional) containing the central lag value of each bin. The zeroth element of the array corresponds to the bin concerning the most negative values of τ, namely the bin covering the range $[-\frac{\texttt{n\_bins}}{2} \cdot \delta\tau, -\frac{\texttt{n\_bins}-1}{2} \cdot \delta\tau)$. The number of bins is given by $$\texttt{n\_bins} = 2 \cdot \mathrm{ceil}\left[\frac{\texttt{max\_lag}}{\delta\tau}\right] + 1\,.$$ Consequently, the lags included in the cross correlation assessment cover the range $\pm\frac{\texttt{n\_bins}}{2} \cdot \delta\tau$.

Assessment of statistical compatibility of IEI distributions

```
p, d = spiSeMe_distribution_test(iei_sequence_A, iei_sequence_B)
```

Mandatory arguments:

`iei_sequence_A`	**NumPy array** containing the first input IEI sequence. The array has to be one-dimensional.
`iei_sequence_A`	**NumPy array** containing the second input IEI sequence. The array has to be one-dimensional.

Returned values:

`p`	**Number** corresponding to the p value provided by the Kolmogorov-Smirnov test carried out between the IEI distributions of the two input sequences.
`d`	**Number** corresponding to the assessed Kolmogorov-Smirnov statistic between the IEI distributions of the two input sequences (i.e. the maximum absolute discrepancy between the corresponding cumulative distributions).

5.3 Examples

The `/examples/Python/` directory stores two Python example scripts containing calls to the SpiSeMe functions. Four example sequences are provided in the `/examples/data/` directory. The first script contains the code required to load data from file, to generate surrogates by means of different methods, as well as to display sequence excerpts and autocorrelation plots. Examples of these plots are shown in Fig. 5.1 in the case of surrogates generated by means of JODI out of the `iei_henon.dat` sequence.

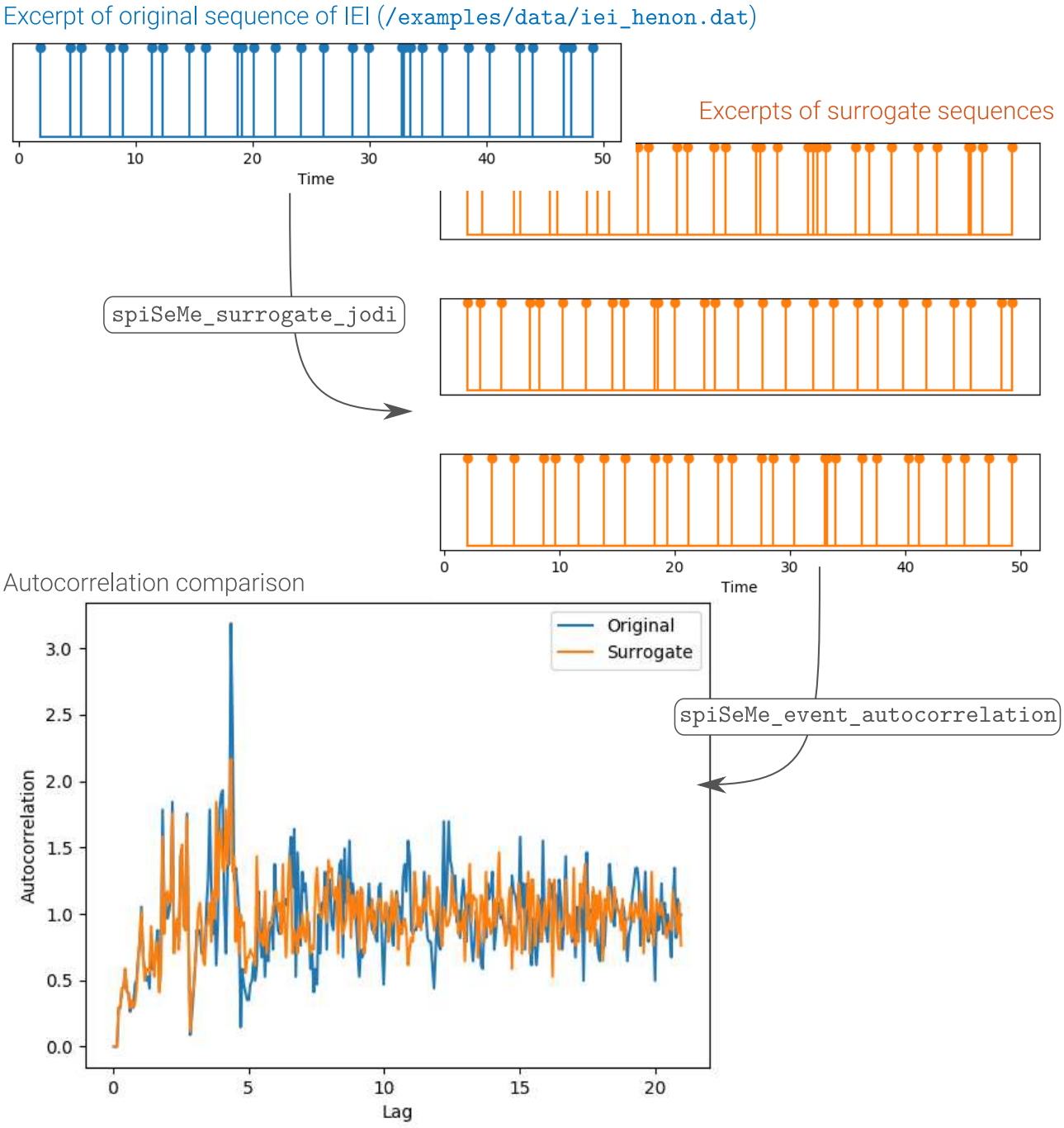


Figure 5.1: Excerpts of the sequence stored in `iei_henon.dat` and of three corresponding surrogate sequences generated by means of JODI. The autocorrelation of the original sequence and of a surrogate one are also shown. See `/examples/Python/example.py` for the code generating these plots.

The second script provides an example of the assessment of cross correlation between event se-

quences and the related significance estimation. This estimation is carried out by generating 500 surrogate sequences by means of JODI. Figure 5.2 shows excerpts of the original sequences, the related cross correlation and the threshold corresponding to the 1% significance level.

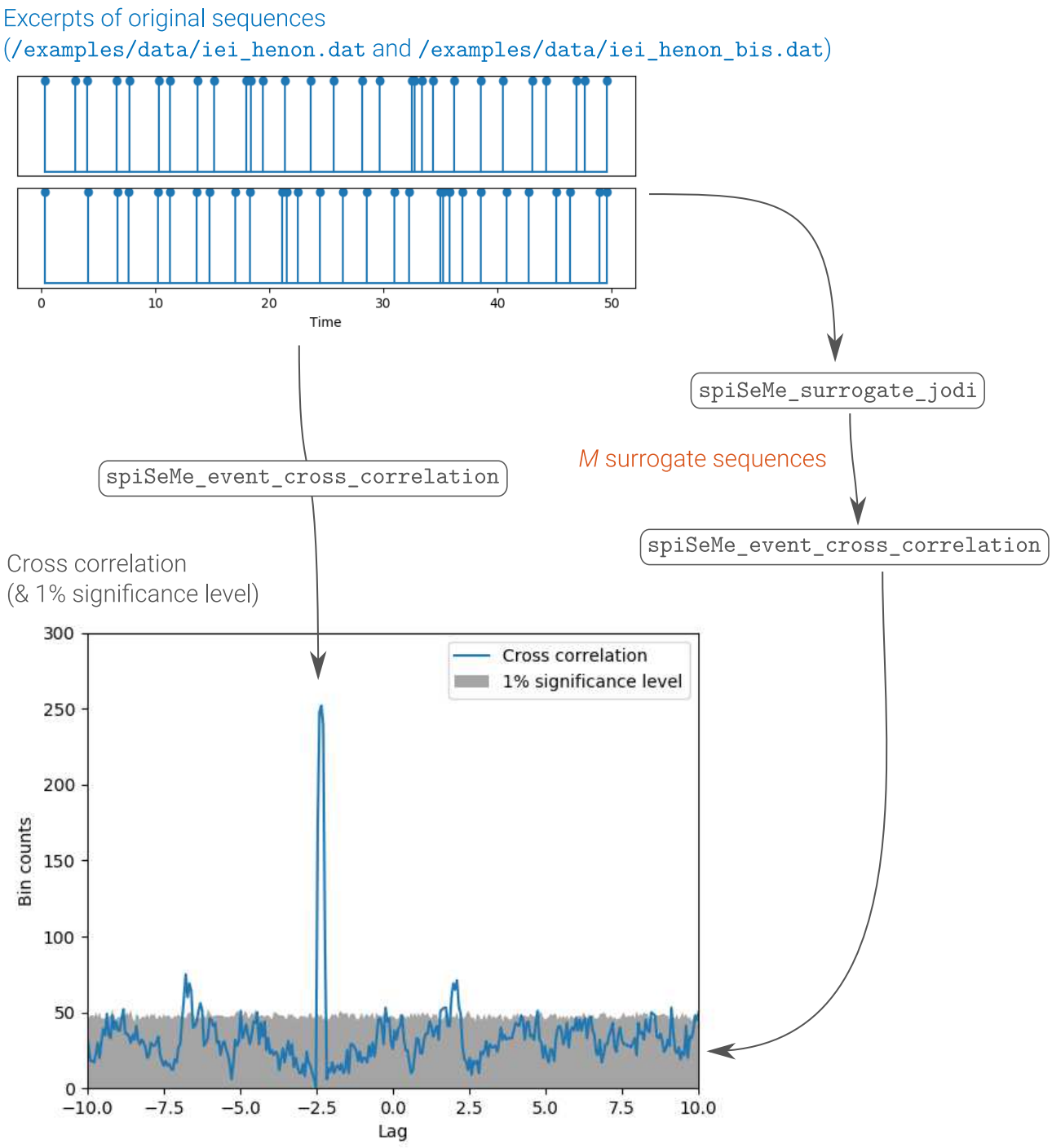


Figure 5.2: Excerpt of the sequences stored in `iei_henon.dat` and `iei_henon_bis.dat`. The cross correlation between the two sequences, as well as the threshold corresponding to the 1% significance level, are shown. See `/examples/Python/example_cross_correlation.py` for the code generating these plots.

A. Unit testing

For all three languages, unit testing scripts/programs are provided. These resources might be helpful for developers who want to modify, expand or adapt the present code. Please note that, because surrogate generation is not a deterministic process, it is impossible to provide a unit testing that checks the numerical output of functions. Nevertheless, wherever possible (JODI, IAAFT, SA), programs that perform tests also check that IEI distributions are exactly conserved.

A.1 C++

Unit tests for the C++ implementation are stored in `code/C++/unit_tests`. The chosen unit testing framework is **Catch2**, which can be found at www.github.com/catchorg/Catch2, and whose header file `catch.hpp` is included in the unit tests directory. **Catch2** is licensed under the Boost Software License 1.0.

Two programs for unit testing are provided, namely `test_surrogate_generation.cpp` and `test_auxiliary_functions.cpp`, which follow the syntax required by **Catch2**. A `Makefile` is provided for a quick compilation and run of the tests on Linux machines. Users might need to modify the `Makefile` in order to adapt it to their own environment. To compile the functions, to compile the tests and to run them, use

```
make test-surrogate
make test-auxiliary
```

Both tests will check the correct behavior of functions both for invalid and for valid arguments. For this reason, during the tests the console will display error statements coming from the spiSeMe functions intentionally fed with invalid arguments. At the end of the tests, a summary of the outcomes is provided.

A.2 MATLAB

Unit tests for the MATLAB implementation are stored in `code/MATLAB/unit_tests`. The unit testing framework is directly provided by MATLAB.

Two MATLAB script files for unit testing are provided, namely `surrogate_generation_test.m` and `auxiliary_functions_test.m`, which follow the syntax required by MATLAB's function-based unit testing framework (see also the corresponding guide on Mathworks website). To run the tests call (from MATLAB):

```
results = runtests('surrogate_generation_test.m')
results = runtests('auxiliary_functions_test.m')
```

Both tests will check the correct behavior of functions both for invalid and for valid arguments. At the end of the tests, a summary of the outcomes is provided in the data structure `results`.

A.3 Python

Unit tests for the Python implementation are stored in `code/Python/unit_tests`. The unit testing framework is directly provided by Python through the `unittest` module.

Two Python script files for unit testing are provided, namely `test_surrogate_generation.py` and `test_auxiliary_functions.py`, which follow the syntax required by Python's unit testing framework (see also https://docs.python.org/3/library/unittest.html). To run the tests (from command line) use:

```
python3 test_surrogate_generation.py
python3 test_auxiliary_functions.py
```

Both tests will check the correct behavior of functions both for invalid and for valid arguments. For this reason, during the tests the console will display error statements coming from the spiSeMe functions intentionally fed with invalid arguments. At the end of the tests, a summary of the outcomes is provided.