
lzcomplexity: entropic analysis of time series

Pleros Group

support@pleros.ai

ver 0.9.0 (Beta)

June 16, 2026

1 Introduction

lzcomplexity is a multithreaded scientific application and library in c++/python, designed for the analysis and characterization of time series through entropy-based metrics. It computes a set of information-theoretic measures to quantify the randomness, structure, and predictability of time series data. The parallel version has been tested in data larger than 10^9 symbols. Time of execution can vary depending on the system hardware (e.g. in an Intel Xenon E5-2670, 2.6 GHz, 16 CPU, 32 threads, 64GB RAM, 10^9 symbols, less than 2 minutes). It has been deployed successfully from PC, laptops, MacPro, up to high performance parallel computing systems.

Both the application and the library computes two key quantities: **entropy density**¹ and **random shuffling effective complexity measure**². Additionally, it provides a metric to compare two sequences using an information-theoretic techniques called **information distance**. To estimate these three metrics, **lzcomplexity** uses the Lempel-Ziv factorization algorithm, which enables accurate approximations of the entropy density.

lzcomplexity is an open-source c++/python library accessible via the source code on (link repo) or the standalone command-line interface (CLI) application under the GPL license. The application can be used in Linux and Mac OS. Additionally, the library in python can be used in any system with python support.

Compiled version exist for Linux Debian, Ubuntu, Mint, Fedora and also for Mac OS.

The manual will first describe the installation and then the use of the stand alone application and the library. A more thorough explanation of the entropic magnitudes follows, and some benchmark plot and comparison to other approaches. Some examples are explained.

2 Getting started

The package includes the *lzcore* library and the CLI application *lzcomplexity*. The *lzcore* library provides the core functionalities of the lzcomplexity package. It includes the fundamental methods for all entropic measures and is defined atomically for processing a single string. The official application, the CLI standalone *lzcomplexity*, use the core functionalities optimally and allows for batch processing of sequences as input files.

¹Entropy density is also known as entropy rate, Kolmogorov-Sinai entropy in different contexts.

²Effective measure complexity is also known as excess entropy

2.1 Installation

The *lzcomplexity* package can be installed in different ways: as a ready to use standalone application, as a python library package or you can build it from source code. The software has been tested on:

- Debian-based GNU/Linux distributions (Ubuntu 22.04+, Mint 22+)
- macOS 12+ (Darwin platform)
- Python 3.10+
- CMake 3.20+

2.1.1 Command-line interface

Debian-based distributions:

1. Add the repository

```
sudo add-apt-repository ppa:pleros/lzcomplexity
sudo apt update
```

2. Install the package

```
sudo apt install lzcomplexity
man lzcomplexity
```

macOS (Darwin Platform)

1. Step 1: Install via Homebrew

```
brew tap pleros
brew install lzcomplexity
man lzcomplexity
```

2.1.2 Installation as a Python Library

1. Configure a virtual environment (this is optional)

```
python -m venv .venv
source .venv/bin/activate
```

2. Install the library via PyPI

```
pip install lzcomplexity
```

2.1.3 Build and Install from source code

1. Clone the repository

```
git clone git@github.com:pleros-ai/lzcomplexity.git
cd lzcomplexity
# [optional] to use the external submodules (tbb and nanobind).
git submodule init
# [optional] to use bundled oneTbb parallel library.
patch external/tbb/CMakeLists.txt patches/tbb.patch
```

2. Configure and build

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j$(nproc)
```

3. Install

```
sudo make install
```

3 How to cite the use of lzcomplexity

When using lzcomplexity both the standalone application or the c++/python library please cite the paper:
XXXXXXXXXXXX

4 Stand alone application

lzcomplexity can be invoked with a number of options and flags. All are optional, the only mandatory element is the input file with the data for processing

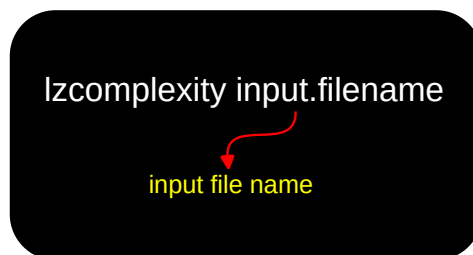


Figure 1: Minimum command line for lzcomplexity

The input file can be in a number of format which can be left to the program to guess it, or can be specified with the `-F|--format` option.

One possible `<format>` is TXT (figure 4), used for a raw text file where one sequence is a string of characters. You can have more than one sequence in a file, each in one line. In such case the `-m` flag should be set, so the lzcomplexity treats each line as a different data string. If the `-m` flag is not used, then the whole file is considered a single sequence.

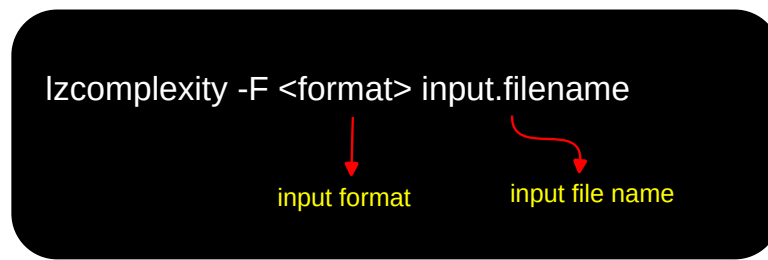


Figure 2: Use of the -F option.

```
lzcomplexity -F TXT input.filename

input.filename:

100011011110111000001010100000...

-----

lzcomplexity -F TXT -m input.filename

input.filename:

atcdddcxaaatatttaaaccccc... 1st sequence
aaatddttttcdddddccctdacdd... 2nd sequence
dcdcdttdddcxaaaddddccc... 3rd sequence
```

Figure 3: The TXT file format.

Another format is CSV (figure 4) which stands for comma separated variable, where sequence are in column format, values from different columns are separated by comma. A slight variation is the TSV format (figure 4), again, sequence are ordered by column, but the delimiter here is a white space.

Finally, the set of formats: PBM, PGM, PNM chooses the corresponding graphic formats. PBM (Portable BitMap), PGM (Portable GrayMap), and PPM (Portable PixMap) are a family of simple, platform-independent graphics file formats. Each format targets a specific image type: PBM for monochrome bitmaps, PGM for grayscale images, and PPM for full-color RGB images. They share a simple structure consisting of a plain text header containing a "magic number" (e.g., P1, P2, P3 for ASCII variants, and P4, P5, P6 for binary variants), image dimensions, and a maximum value, followed by the pixel data. This data can be stored in a human-readable ASCII format or a more compact binary format. With the umbrella term PNM (Portable Anymap) often used to refer to the family collectively.

The full set of options and flags are shown in figure 6. This is the output from `lzcomplexity` when the `-h` flag is used. The option `-a|alphabet <value>` is used to set the alphabet size. `value` must be larger than 1. By default the alphabet cardinality is guessed by scanning the input sequence before processing.

In a similar fashion, the `-l|--log-base <value>` option can be used to set the logarithm base when calculating the entropic magnitudes. If not specified, then the cardinality of the alphabet is used as the logarithm base.

Another option is `-f|--factors <factor-file>`, which commands to save the Lempel-Ziv factorization to the file specified by `<factor-file>`. The output file is in json format

```
lzcomplexity -m -F CSV input.filename

input.filename:
1st sequence
2nd sequence
3rd sequence
4th sequence
1 2,3,2, ...
3 1,1,1,...
0 2,2,2,...
2 3,3,1...
...

lzcomplexity -m -F TSV input.filename

input.filename:
1st sequence
2nd sequence
3rd sequence
4th sequence
1 2 3 2 ...
3 1 1 1,...
0 2 2 2,...
2 3 3 1...
...
```

Figure 4: The CSV and TSV file formats.

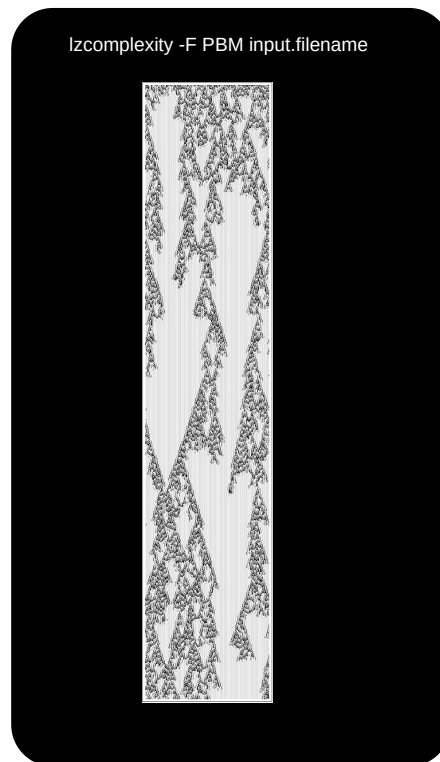


Figure 5: The PBM, PGM, PNM file formats.

```
LempelZiv-76 complexity engine. Suited for complexity analysis of time series. Send bug reports to estevez@fisica.uh.cu
or efrenaragon96@gmail.com.

usage: lzcomplexity [OPTIONS] <file>

OPTIONS:
  -a, --alphabet value      Alphabet cardinality. If auto it tries to guess the alphabet size.
  -d, --dlz                 The LZ distance is calculated between consecutive lines. Only valid for multiline files
                           (-m option).
  -e, --excess-options [=v1]:[f]:[v2]:[v3](=a)] Random shuffle complexity options for calculation. v1: maximum value for block
                           shuffling, f: summand factors, v2: starting line for summand factors, v3: ending line
                           for summand factors. All values are optional
  -f, --factors file_name   Saves the factorization.
  -F, --format value        Input file format. TXT for raw text format. CSV the input file is a csv array with ','
                           as delimiter. TCSV the input file is a csv array with ' ' as delimiter. PBM, PGM and
                           PNM is for the family of the graphic formats. (default: AUTO)
  -h, --help               Show this message.
  -j, --jobs value          Number of threads. (default: 10)
  -l, --log-base value      The log base value. The default is the alphabet cardinality.
  -m, --multi-line         Treat each line in the input stream as a different sequence.
  -n, --entropy-density     Computes only the entropy density.
  -o, --output file_name    Output filename. Default appends to the end of input file a .json extension
  -p, --partitions value    Number of partitions used for the parallel suffix array algorithm. (default: 2)
  -v, --verbose             Verbose output.
  -V, --version             Output the version number.
  -w, --warn-out            Hide warning messages.
```

Figure 6: lzcomplexity options.

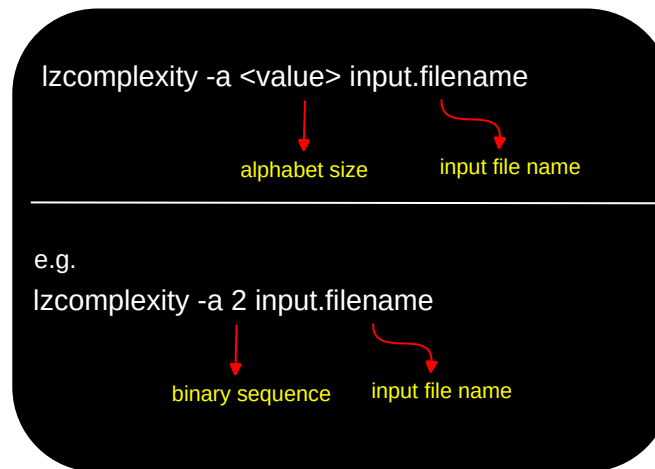


Figure 7: the use of the -a option.

Factors output file format

```
{
  "filename": string, /* input file name with sequences */
  "format": string, /* format of the input file */
  "size": integer, /* number of sequence processed in the input file */
  "factors": [$f_1$, $f_2$, $f_3$, ...], /* matrix with the factors index
    ↳ of each sequence by row. Each $f_i$ is a vector of factor index
    ↳ (e.g [1,2,4,...]) */
}
```

The `-j|--jobs` option is a natural number that sets the number of threads to be used in parallel computing. By

default, the number of threads used is the maximum number possible by the architecture.

The `-p|--partitions` option specifies the number of partitions used by the suffix array algorithm CaPS-SA. The value should be a positive integer. In case of the option is not defined or the value is set to 0, the number of partitions is calculated according to the sequence size for optimal performance in parallel processing.

The `-v|--verbose` flag enables verbose mode, generating console output during program execution, allowing users to monitor its progress.

The `-V|--version` flag displays the installed version of the application.

The `-w|--warn-out` flag disables the warnings displayed during execution.

4.1 Random shuffle effective measure complexity (RSEM)

There is an option `-e|-rsemc <value>`, where the user can specify the maximum size of blocks used for shuffling and the range of lines in the input file, where the RSEM is to be computed. Details of the calculation of RSEM can be found in section 6.2.

The format for the `-e|-rsemc` option is

```
lzcomplexity -e <v1>:<f>:<v2>:<v3> input_file
```

The `v1` value specifies the maximum size of blocks used in the RSEM calculation, it must be a number greater than 0 or the character 'a' for automatic block size calculation. If the second value in the option is `f`, then the summands of the RSEM are also reported. `<v2>` and `<v3>` define the range of lines where RSEM will be computed, the first line is denoted by index 1. If `<v2>` is specified but `<v3>` is missing, the range is taken from `<v2>` to the end of the file (see figure 8).

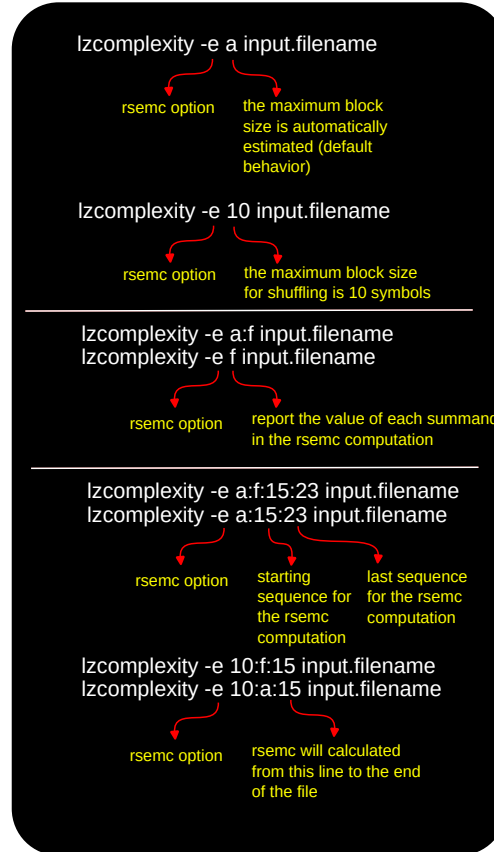


Figure 8: Different formats using the `-e` option.

The `-n|--entropy-density` flag disables the calculation of the random shuffle effective measure complexity, speeding the execution time.

4.2 Informational distance

In *lzcomplexity*, the informational distance, via the Lempel-Ziv factorization, is computed between consecutive lines from the input data by using the `-d` flag (figure 9). The distance, therefore, only makes sense if the input file is considered multiline, `-m` flag is assumed even if not specified.

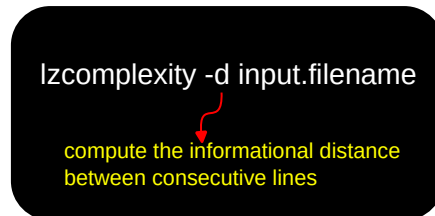


Figure 9: The `-d` option.

4.3 Output file format

By default, the output file will have the same name as the input file, except with the extension **.l76.json**. However, the user can change the default output filename using the `-o|--output` option.

```
lzcomplexity -o < output_file > < input_file >
```

lzcomplexity generates the output in JSON format. It includes general information about the processed data, such as the filename, format, and size of the input file. Additionally, the output file contains a field called *sequences*, which is an array of objects. Each object holds the computed information from one string. The order of elements in the array of objects follows the order of lines in the input file.

Results file format

```
{
  "filename": string, /* input file name */
  "format": string, /* format of the input file (e.g. TXT, PBM) */
  "size": integer, /* number of sequence in the input file */
  "sequences": [s1,s2,s3,...], /* vector of sequence objects for each
    ↳ line in the input file (see below) */
  "lz76Distance": {
    "InformationDistance": [d1,d2,d3,...], /* vector with information
      ↳ distances between consecutive lines (e.g.
      ↳ [0.1,0.4,0.2,0.9,0.2,...]). The first value corresponds to the
      ↳ distance between the first and second line, the second value
      ↳ between the second and third line, and so on.*/
  }
}
```

Sequence object

```
{
  "size": integer, /* number of characters in the sequence */
  "lz76Complexity": integer, /* Lempel-Ziv complexity of the sequence */
  "lz76EntropyDensity": double, /* Lempel-Ziv entropy density estimation
    ↪ */
  "lz76RandomShuffleComplexity": {
    "value": double, /* RSEMC estimation */
    "max_block_size": integer, /* maximum size of block used in the
    ↪ RSEMC estimation*/
    "multi_information": double, /* multi information value (the first
    ↪ summand in the RSEMC sum) */
    "summands": [...] /* vector of the summand values used in RSEMC sum
    ↪ */
  }
}
```

Figure 10: JSON schema for the output file.

5 Library

lzcore is the core library that provides the foundation for all functionalities used on the *lzcomplexity* package. It contains the atomic definitions of all structures and the functions responsible for computing entropic magnitudes.

C++

```
#include <lz/lz.h>
```

Python

```
import lzcomplexity as lz
```

5.1 Classes and Structures

sequence

The sequence class represents the stream for the library. It has the necessary methods to access and modify the elements of a sequence of symbols. All data members are protected.

Member variables	Type	Description
seq	vector<char>	the ordered set of symbols representing the stream
alphabet	vector<char>	the alphabet of symbols
alphabet_size	unsigned int	the number of symbols in the alphabet

Constructors			
Method	Parameter	Parameter Type	Description
sequence()	—	—	Default constructor. Build an empty vector and sets the alphabet size to 2.
sequence(aph_size)	aph_size	int	Constructor that sets the alphabet size.
(1) sequence(str) (2) sequence(str, aph)	str aph	const string& string_view& const char* vec unsigned int	1) Initializes the sequence from a string-type class and determines the alphabet from it. 2) Initializes the sequence from a string-type class and sets the alphabet size.
(1) sequence(vec) (2) sequence(vec, aph)	vec aph	const span<char>& vector<char>& initializer_list<char>& array<char, N>& unsigned int	1) Initializes the sequence from a string-type class and determines the alphabet from it. 2) Initializes the sequence from a string-type class and sets the alphabet size.
~sequence()	—	—	Destructor.

Member Functions			
Method [return type C++]	Parameter	Parameter Type	Description
getAlphabetSize() [unsigned int/int]	—	—	Returns the current alphabet size.

Member Functions			
Method [return type C++]	Parameter	Parameter Type	Description
(1) <code>setAlphabetSize()</code> [unsigned int] (2) <code>setAlphabetSize(aph)</code> [void/void]	aph	unsigned int	1) Sets the alphabet size based on the content of the stored sequence; returns computed size. 2) Sets the alphabet size to a specified value.
<code>getAlphabet()</code> [const vector<char>&]	—	—	Returns a const reference to the current alphabet.
(1) <code>setAlphabet()</code> [const vector<char>&] (2) <code>setAlphabet(aph)</code> [void]	aph	vector<char>	1) Sets the alphabet from the current sequence content; returns the computed alphabet. 2) Sets the alphabet to a specified vector of characters.
<code>NoZeroes()</code> [unsigned int]	—	—	Counts the number of zero-values characters ('\0') in the sequence.
<code>toString()</code> [string]	—	—	Converts the sequenc to a standard string.
<code>charDensity()</code> [map<char,unsigned int>]	—	—	Calculates the frequency of each symbol in the stream.
(1) <code>first()</code> [char&] (2) <code>first()</code> [const char&]	—	—	Returns de (const-)reference to the first character of the sequence.
(1) <code>last()</code> [char&] (2) <code>last()</code> [const char&]	—	—	Returns de (const-)reference to the last character of the sequence.
(1) <code>at()</code> [char&] (2) <code>at()</code> [const char&]	idx	int	Access to the idx position of the sequence (throws <code>out_of_range</code>).
(1) <code>Min()</code> [char] (2) <code>Min(start, final)</code> [char]	start	int	1) Finds the minimum character value in the entire sequence.
	final	int	2) Finds the minimum character in the range [start, final).

Member Functions			
Method [return type C++]	Parameter	Parameter Type	Description
(1) Max() [char]	start	int	1) Finds the maximum character value in the entire sequence. 2) Finds the maximum character in the range [start, final).
(2) Max(start, final) [char]	final	int	
push(c) [int]	c	char	Appends a character to the end of the sequence; returns the new size (throw SequenceBadAlloc on allocation failure).
pop() [int]	—	—	Removes the last character of the sequence; returns the new size (0 for undefined sequences).
back() [char]	—	—	Returns a copy of the last character.
size() [int] length() [int]	—	—	Returns the number of characters in the sequence.
SequenceVector() [const vector<char>&]	—	—	Returns a const reference to the internal character vector.
Take(len) [sequence]	len	int	Creates a new sequence containing the first len characters. If len exceeds the size, the result is truncated to the size of the sequence.
Drop(len) [sequence]	len	int	Creates a new sequence with the first len characters removed.
Split(len) [pair<sequence, sequence>]	len	int	Split the sequence at position len into two sequences: first [0, len) and second [len, end).
Granularity(gr) [sequence]	gr	unsigned int	Creates a coarse-grained version of the sequence. Increase the alphabet of the sequence by grouping gr characters as new alphabet symbol.
pi() [sequence&]	—	—	Remove the last character and returns a reference to the resulting sequence.

Member Functions			
Method [return type C++]	Parameter	Parameter Type	Description
(1) <code>reverse()</code> [sequence&] (2) <code>reverseCopy()</code> [sequence&] (3) <code>reverseCopy()</code> [const sequence]	—	—	1) Reverses the sequence in place and returns the reference. 2,3) Creates and returns a reversed copy of the sequence
(1) <code>rightShift(ls)</code> [sequence&] (2) <code>leftShift(ls)</code> [sequence&]	ls	unsigned int (default 1)	Shift all characters 1) right or 2) left by ls positions (circular shift); returns a reference to the shifted sequence.
<code>DetermineAlphabet()</code> [vector<char>]	—	—	Scans the sequence, determines unique symbols that conform the alphabet , and update alphabet_size in case of mismatch with current value. Returns the alphabet .
<code>clear()</code> [void]	—	—	Clears the sequence and resets the alphabet size to default value 2
(1) <code>swap(s1, s2)</code> [void] (2) <code>swap(s1, start1, start2, len)</code> [void]	s1	sequence&	1) Swaps the contents of two sequences. 2) Swaps two non-overlapping ranges of characters with the same length within the same sequence.
	s2	sequence&	
	start1	int	
	start2	int	
	len	int	

Operators			
Method [return type C++]	Parameter	Parameter Type	Description
<code>operator[]</code> [char&] <code>operator[]</code> [const char&]	idx	int	Accesses to a specific character at the specified idx (bounds-checked and throws <code>SequenceOutOfBounds</code>).
<code>operator=</code> [sequence&]	s	const sequence&	Copy assignment operator.

Operators			
Method [return type C++]	Parameter	Parameter Type	Description
<code>operator=</code> [sequence&]	s	sequence&&	Move assignment operator.
<code>operator=</code> [sequence&]	s	const vector<char>&	Assignment from a character vector.
<code>operator+</code> [sequence&]	lhs	const sequence&	Concatenate two sequence
	rhs	const sequence&	
<code>operator+=</code> [sequence&]	s	const vector<char>&	Appends a character vector to the end of the current sequence.
<code>operator+=</code> [sequence&]	s	const sequence&	Appends another sequence to the end of the current sequence.
<code>operator==</code> [bool]	lhs	const sequence&	Equality comparison between two sequences.
	rhs	const sequence&	
<code>operator!=</code> [bool]	lhs	const sequence&	Inequality comparison between two sequences.
	rhs	const sequence&	
<code>operator></code> [bool]	lhs	const sequence&	Lexicographically comparison greater than.
	rhs	const sequence&	
<code>operator>=</code> [bool]	lhs	const sequence&	Lexicographically comparison greater than or equal.
	rhs	const sequence&	
<code>operator<</code> [bool]	lhs	const sequence&	Lexicographically comparison less than.
	rhs	const sequence&	

Operators			
Method [return type C++]	Parameter	Parameter Type	Description
operator<= [bool]	lhs	const sequence&	Lexicographically comparison less than or equal.
	rhs	const sequence&	
operator^= [sequence&]	s	const sequence&	Element-wise <i>XOR</i> comparison (0 if equal and 1 if different). Throws <code>SequenceNoMatchSize</code> when sizes differ.
operator<< [ostream&]	os	ostream&	Serialize the sequence to an output source (format: { content, [...alphabet], alphabet_size }).
	obj	const sequence&	
operator>> [istream&]	is	istream&	Deserialize a line from an input stream into the sequence class.
	obj	const sequence&	
template<Fn> map(fn, s) [sequence]	fn	Fn&& function<char(char)>	Applies a transformation function to each element in the sequence.
	s	const sequence&	
template<Fn> map(s, fn) [sequence]	s	const sequence&	Applies a transformation function to each element in the sequence.
	fn	Fn&& function<char(char)>	
template<Fn> map(fn) [sequence]	fn	Fn&& function<char(char)>	Applies a transformation function to each element in the sequence and return a new sequence as result.

C++

```

// main.cpp
#include <lz/sequence.h>
#include <iostream>

int main(){
    lz::sequence seq1("banana");
    lz::sequence seq2({'h', 'e', 'l', 'l', 'o'});

    auto [fh, sh] = seq1.Split(seq1.size() / 2);
    std::cout << fh << std::endl; // > { ban, [ n b a ], 3 }

    seq2.push('s');
    std::cout << seq2 << std::endl; // > { hellos, [ o l h e ], 4 }

    return 0;
}

```

LempelZiv76

The `LempelZiv76` class implements the LZ76 factorization algorithm. It computes the number of distinct factors in the exhaustive story, their boundaries, and related statistics such as the ε factor and the standard deviation of factor lengths. The class is designed to work with the `sequence` class, and it can optionally use a pre-computed suffix array for efficiency. This class is only available on the C++ package. All data members are protected.

Member variables	Type	Description
<code>factorization</code>	<code>unsigned int</code>	the number of factors of the LZ76 factorization (the LZ76 complexity)
<code>lzf</code>	<code>vector<unsigned int></code>	vector of starting indices of each factor in the original sequence
<code>epsilon</code>	<code>double</code>	the ε factor. See equation (7)
<code>factors_stddev</code>	<code>double</code>	the standard deviation of the lengths of the factors

Constructors			
Method	Parameter	Parameter Type	Description
<code>LempelZiv76()</code>	—	—	Default constructor; initialises factorization to 0.
<code>LempelZiv76(SA)</code>	SA	<code>LZ_SuffixArray</code>	Constructs and compute the factorization from a pre-build suffix array.
<code>LempelZiv76(lz)</code>	SA	<code>const LempelZiv&</code>	Copy constructor.
<code>LempelZiv76(lz)</code>	lz	<code>LempelZiv&&</code>	Move constructor.

Constructors			
Method	Parameter	Parameter Type	Description
<code>~LempelZiv76()</code>	—	—	Destructor; frees all allocated memory.

Member functions			
Method [return type C++]	Parameter	Parameter Type	Description
<code>Factorize(seq)</code> [LZ_Result]	seq	const sequence&	Computes the LZ76 factorization using the CaPS algorithm; returns a structure containing the complexity, the ε factor, and the factor boundaries.
<code>Factorize(str, N)</code> [LZ_Result]	str N	const char* int	Same as above, but works directly on a raw character array.
<code>Factorize(seq, args)</code> [LZ_Result]	args	LZ_Args&	Computes the factorization with custom parameters.
<code>Factorize(SA)</code> [LZ_Result]	SA	const LZ_SuffixArray	Computes the factorization from an already constructed suffix array.
<code>FoundStddev()</code> [double]	—	—	Returns the standard deviation of the factor length (the factorization must have been calculated beforehand).
<code>Clear()</code> [void]	—	—	Clears all internal data and releases memory.
<code>getFactorization()</code> [const unsigned int]	—	—	Returns the LZ76 complexity (number of factors)
<code>getFactors()</code> [const vector<unsigned int>&]	—	—	Returns a const reference to the vector of factor starting indices.
<code>getEpsilon()</code> [const double]	—	—	Returns the ε factor of the factorization.
<code>getStddev()</code> [const double]	—	—	Returns the standard deviation of the factor lengths.
<code>getFactorsBegin()</code> [const_iterator]	—	—	Returns an iterator pointer to the beginning of the factor boundaries vector.
<code>getFactorsEnd()</code> [const_iterator]	—	—	Returns an iterator pointer to the ending of the factor boundaries vector.

Member functions			
Method [return type C++]	Parameter	Parameter Type	Description
swap(lhs, rhs) [void]	lhs	LempelZiv76&	Exchanges the content of two LempelZiv76 objects.
	rhs	LempelZiv76&	

Operators			
Method [return type C++]	Parameter	Parameter Type	Description
operator= [LempelZiv76&]	lz	const LempelZiv76&	Copy assignment operator.
operator= [LempelZiv76&]	lz	LempelZiv76&&	Move assignment operator.
operator== [bool]	lhs	LempelZiv76&	Equality comparison based on the factorization (compare the factor vectors and complexity).
	rhs	LempelZiv76&	
operator!= [bool]	lhs	LempelZiv76&	Inequality comparison (negation of operator==)
	rhs	LempelZiv76&	
operator<< [ostream&]	os	ostream&	Serialize the object to an output source with a human-readable representation of the factorization and the factors' boundaries vector.
	lz	LempelZiv76&	

Here is an example of the use of LempelZiv76.

C++

```

// main.cpp
#include <lz/lz76.h>
#include <iostream>

int main(){
    lz::internal::LempelZiv76 LZ76;
    auto res = LZ76.Factorize("banana", 6);
    std::cout << LZ76.getFactorization() << std::endl; // 3

    std::string str("1001110110101000");
    res = LZ76.Factorize(str);
    std::cout << LZ76.getFactorization() << std::endl; // 6

    auto factors = LZ76.getFactors();
    for (auto i = 0ul; i < factors.size() - 1; i++) {
        std::cout << str.substr(factors[i], factors[i + 1] - factors[i]) << ".";
    }
    // > 1.0.01.110.11010.1000.

    return 0;
}

```

CaPS-SA

The `CaPS_SA` class implements a cache-friendly Parallel Suffix array construction algorithm (CaPS) [1]. It builds the suffix array (SA) and the longest common prefix (LCP) array for a given input text following a divide-and-conquer strategy that is optimised for multi-core systems and cache efficiency. The algorithm splits the problem into small subproblems, sorts them independently in parallel (using oneTBB [2]), and then merges them with a pivot-based partitioning scheme. Just like the other classes, the data members are protected.

Member variables	Type	Description
<code>T_</code>	<code>vector<char></code>	input text
<code>n_</code>	<code>int</code>	length of the input text
<code>SA_</code>	<code>vector<int></code>	suffix array
<code>LCP_</code>	<code>vector<int></code>	longest common prefix
<code>p_</code>	<code>int</code>	number of subproblems used during parallel construction
<code>mac_context</code>	<code>int</code>	maximum prefix length considered when comparing suffixes (0 means unlimited)

The following table shows all public functions of `CaPS_SA`. The protected helper functions are omitted.

Constructors			
Method [return type C++]	Parameter	Parameter Type	Description
CaPS_SA(args)	args	SA_Args	Constructs from a configuration object containing the values of subproblem count and max context.
CaPS_SA(subproblem_count, max_context)	subproblem_count	int (default 0)	Construct the class from the values of subproblem count and max context; subproblem_count = 0 means auto-selection for the number of subproblems to process in parallel, and max_context = 0 means to use all suffix in the lexicographical comparison.
	max_context	int (default 0)	
CaPS_SA(T, n, subproblem_count, max_context)	T	vector<char>	Construct the class using a vector of characters T and its length n. Also, it can receive the subproblem count (0 means auto-select the number of subproblems to process in parallel) and max context (0 means to use all suffixes in the lexicographical comparison).
	n	int	
	subproblem_count	int (default 0)	
	max_context	int (default 0)	
CaPS_SA(other)	other	const SA_Args&	Copy constructor.
CaPS_SA(other)	other	SA_Args&&	Move constructor.
~CaPS_SA()	—	—	Destructor; frees all dynamically allocated memory.

Member Funtions			
Method [return type C++]	Parameter	Parameter Type	Description
T() [vector<char>/list]	—	—	Returns a copy of the input text.
n() [int/int]	—	—	Returns the length of the input text.
SA() [vector<int>/list]	—	—	Returns a copy of the computed suffix array.

Member Functions			
Method [return type C++]	Parameter	Parameter Type	Description
LCP() [vector<int>/list]	—	—	Returns a copy of the computed LCP array.
(1) construct() [LZ_SuffixArray/LZ_SuffixArray] (2) construct(str) [LZ_SuffixArray/LZ_SuffixArray] (3) construct(T) [LZ_SuffixArray/LZ_SuffixArray]	str	const string&	1) Constructs SA and LCP for the internally stored text; returns the structure with the both arrays.
	T	vector<char>	2) Constructs SA and LCP for the given string (overwrites any previously stored text) 3) Constructs SA and LCP for the provided text as a character vector
swap(lhs, rhs) [void]	lhs	CaPS_SA&	Exchanges the content of two CaPS_SA objects.
	rhs	CaPS_SA&	

Operators			
Method [return type C++]	Parameter	Parameter Type	Description
operator= [CaPS_SA&/CaPS_SA]	s	const CaPS_SA&	Copy assignment operator.
operator= [CaPS_SA&/CaPS_SA]	s	CaPS_SA&&	Move assignment operator.
operator== [bool/bool]	lhs	const CaPS_SA&	Equality comparison based on configuration and stored text.
	rhs	const CaPS_SA&	
operator!= [bool/bool]	lhs	const CaPS_SA&	Inequality comparison (negation of operator==).
	rhs	const CaPS_SA&	

Here is an example of the suffix array construction.

C++

```
// main.cpp
#include <lz/caps.h>
#include <iostream>

int main(){
    std::string str("banana");
    lz::suffixarray::CaPS_SA sa;

    auto res = sa.construct(str);

    for (auto suf: res.SA) {
        std::cout << suf << " ";
    }
    // > 5 3 1 0 4 2

    return 0;
}
```

5.1.1 Other structures

The following structures are only available in the C++ library.

LZ_Result

LZ_Result is a data-only structure used to store the result values obtained from the factorization method. It contains three members: **factorization**, which stores the factorization value, **epsilon**, which store the epsilon factor, and **lzf** which stores the vector with the initial index of each factor (See **Factorize** method of *LempelZiv76* class).

```
struct LZ_Result {
    unsigned int factorization;
    double epsilon;
    std::vector<unsigned int> lzf;
}
```

SA_Args

The SA_Args contains parameters that control the behavior of the suffix array algorithms. The **chunks** controls the number of subproblems for the parallel construction; its value by default is 0 which means automatic number of partitions. The **max_context** says the maximum prefix length for suffix comparison; by default, it is 0 or unlimited context for suffix comparison. This structure is only available in the C++ package; for Python, the structure's member variables are included in the LZ_Args class.

```
struct SA_Args {
    int chunks = 0;
    int max_context = 0;
}
```

LZ_Args

The LZ_Args structure contains parameters that control the behavior of *lzcore* functions. It inherits the parameters from SA_Args and adds new ones. The block_size and get_shuffle_terms parameters are used in calculating the random shuffle effective measure complexity calculation. The block_size parameter defines the maximum length of the block used, while the get_shuffle_terms parameter is a flag for storing the summands of the algorithms. Additionally, the structure stores the alphabet and the log_base used for the log calculations.

```
struct LZ_Args: SA_Args {
    int block_size = -1;
    bool get_shuffle_terms = false;
    unsigned int alphabet = UINT_MAX;
    unsigned int log_base = UINT_MAX;
}
```

LZ_SuffixArray

The LZ_SuffixArray structure stores the suffix array (SA) and Longest Common Prefix (LCP) computed for a given text, along with the text length. It is the object returned by the suffix array construction method (See **construct** method of *CaPS_SA* class).

```
struct LZ_SuffixArray {
    std::vector<unsigned int> SA;
    std::vector<unsigned int> LCP;
    unsigned int n;
}
```

LZ_Shuffle

The LZ_Shuffle is a data-only structure that stores the result of the random shuffle effective measure complexity method. It includes the max_block_size, which represents the maximum block length used, the excess_value, which represents the effective measure complexity value, and a vector of summands that stores all the summands used in the calculation. Additionally, it contains the multi_information, which represents the total correlation. This value represents how much information was lost from the original sequence to the 1-block random sequence.

```
struct LZ_Shuffle {
    int max_block_size = -1;
    double emc_value = 0;
    double multi_information = 0;
    std::vector<double> summands;
}
```

LempelZiv

The Lempel-Ziv is a class that contains the LZ76 factorization, the entropy density, and the random shuffle effective measure complexity. It also stores the factor vector, the ε factor, and the standard deviation of the factor lengths. Furthermore, it can include the error estimate, assuming the Normal and Poisson distributions. All the member variables are protected, so there are also implemented all the get functions for each one of the variables in the class.

```
struct LempelZiv {
    unsigned int complexity;
    double entropy_density;
    LZ_Shuffle random_shuffle_complexity;
    LZ_Shuffle paired_shuffle_complexity;
    double lz_normal_errors;
    double lz_poisson_errors;
    double epsilon;
    double factors_stddev;
    std::vector<unsigned int> factors;
}
```

5.2 API Functions

All the functions in the library accept a **sequence** as an input argument to be processed. Optionally, they can also receive as input an instance of the `utils::LZ_Args` (See more) structure for better control over execution parameters. If the second argument is missing, the library uses the following default values for the computation.

- chunk

$$\left\{ \begin{array}{ll} 1 & \text{if the length of the input sequence} < \#_t * 10 \\ \#_t & \text{if the length of the input sequence} < 10^6 \\ 2^8 & \text{if the length of the input sequence} < 10^7 \\ 2^{10} & \text{if the length of the input sequence} < 10^8 \\ 2^{13} & \text{otherwise} \end{array} \right.$$
- block_size = -1
- get_shuffle_terms = false
- alphabet = 2
- log_base = 2
- max_context = 0

lz76

This function optimizes the calculation of the library magnitudes. It is an all-in-one utility function for obtaining all entropic measures of an input sequence while minimizing the use of the LZ76 factorization and memory usage. Additionally to LZ76 factorization, h_{lz} , and E_{lz} , the method calculates the factors, the ε factor, the standard deviation of the factor lengths, and the error estimations following Normal and Poisson distributions. Return an instance of the `LempelZiv` object.

C++

```
LempelZiv lz76(const sequence&, utils::LZ_Args)
LempelZiv lz76(const sequence&)
```

```

// main.cpp
#include <lz/lempelziv.h>
#include <iostream>

int main() {
    // Analyze a symbolic sequence
    std::string seq = "banana";
    auto result = lz::lz76(seq);

    std::cout << "LZ76_Complexity:_" << result.getComplexity() << std::endl;
    // > LZ76 Complexity: 3
    std::cout << "Entropy_density:_" << result.getEntropyDensity() << std::endl;
    // > Entropy density: 0.815465
    std::cout << "Effective_complexity:_" << result.getRandomShuffleComplexity().emc_value << std::endl;
    // > Effective complexity: 0

    return 0;
}

```

Python

```

tuple[int, float, list, tuple[int, float, float]] lz76(
    seq: list | str,
    partitions: int = 1,
    alphabet: int = 2,
    log_base: int = 2,
    max_block_size: int = -1,
    jobs: int = std::thread::hardware_concurrency()
)

```

```

# main.py
import lzcomplexity as lz

# Analyze a symbolic sequence
seq = "banana"
complexity, h, factors, (block_size, emc, multi_info) = lz.lz76(seq)

print(f"LZ76_Complexity:_{complexity}")
print(f"Entropy_density:_{h}")
print(f"Effective_complexity:_{emc}")

```

lz76Factorization

The function calculates the LZ76 factorization of an input sequence.

```

unsigned int lz76Factorization(const sequence&, utils::LZ_Args)
unsigned int lz76Factorization(const sequence&)

```

C++

```
// main.cpp
#include <lz/lempelziv.h>
#include <iostream>

int main() {
    std::string seq = "banana";
    auto result = lz::lz76Factorization(seq);

    std::cout << "LZ76 Complexity: " << result << std::endl;
    // > LZ76 Complexity: 3

    return 0;
}
```

Python

```
int factorization(seq: list | str,
                  partitions: int = 1,
                  alphabet: int = 2,
                  log_base: int = 2,
                  jobs: int = std::thread::hardware_concurrency())
```

```
# main.py
import lzcomplexity as lz

seq = "banana"
result = lz.factorization(seq)

print(f"LZ76 Complexity: {result}")
```

lz76Factors

Calculate the factors that make up the factorization of LZ76, in addition to its factorization. The method returns an instance of LZ_Result structure (See more details of the structure)

```
LZ_Result lz76Factors(const sequence&, utils::LZ_Args)
LZ_Result lz76Factors(const sequence&)
```

C++

```
// main.cpp
#include <lz/lempelziv.h>
#include <iostream>

int main() {
    std::string seq = "banana";
    auto result = lz::lz76Factors(seq);

    std::cout << "LZ76complexity:_" << result.factorization << std::endl;
    // > LZ76 complexity: 3
    for (auto i = 0ul; i < result.lzf.size() - 1; i++) {
        std::cout << seq.substr(result.lzf[i], result.lzf[i + 1] - result.lzf[i]) << ".";
    }
    // > b.a.n.ana.

    return 0;
}
```

Python

```
tuple[int, list] factors(seq: list | str,
                        partitions: int = 1,
                        alphabet: int = 2,
                        log_base: int = 2,
                        jobs: int = std::thread::hardware_concurrency())
```

```
# main.py
import lzcomplexity as lz

seq = "banana"
factorization, factors = lz.factors(seq)

print(f"LZ76complexity:_{factorization}")
print(f"LZ76factors:_{factors}")
```

la76EntropyDensity:

Calculates the entropy density (h_{lz}) of the input sequence.

```
double lz76EntropyDensity(const sequence&, utils::LZ_Args)
double lz76EntropyDensity(const sequence&)
```

C++

```
// main.cpp
#include <lz/lempelziv.h>
#include <iostream>

int main() {
    std::string seq = "banana";
    auto result = lz::lz76EntropyDensity(seq);

    std::cout << "Entropy_density:_" << result << std::endl;
    // > Entropy density: 0.815465

    return 0;
}
```

Python

```
float entropy_density(seq: list | str,
                      partitions: int = 1,
                      alphabet: int = 2,
                      log_base: int = 2,
                      jobs: int = std::thread::hardware_concurrency())
```

```
# main.py
import lzcomplexity as lz

seq = "banana"
result = lz.entropy_density(seq)

print(f"Entropy_density:_{result}")
```

lz76RandomShuffleComplexity:

Calculates the random shuffle effective measure complexity (E_{lz}) of the input sequence. The method returns an instance of LZ_Shuffle structure (See more details of the structure).

```
LZ_Shuffle lz76RandomShuffleComplexity(const sequence&, utils::LZ_Args)
LZ_Shuffle lz76RandomShuffleComplexity(const sequence&)
```

C++

```
// main.cpp
#include <lz/lempelziv.h>
#include <iostream>

int main() {
    std::string seq = "10001011101110100101010101110101110001101110000100101011110101101";
    auto result = lz::lz76RandomShuffleComplexity(seq);

    std::cout << "Effective_measure_complexity:_" << result.emc_value << std::endl;
    // > Effective measure complexity: 0.926518

    return 0;
}
```

Python

```
tuple[int, float, float] emc(seq: list | str,
                             partitions: int = 1,
                             alphabet: int = 2,
                             log_base: int = 2,
                             max_block_size: int = -1,
                             jobs: int = std::thread::hardware_concurrency())
```

```
# main.py
import lzcomplexity as lz

seq = "10001011101110100101010101110101110001101110000100101011110101101"
block_size, emc_value, multi_info = lz.emc(seq)

print(f"Entropy_density:_{emc_value}")
```

lz76InformationDistance:

Calculates the information distance (d_{lz}) between two sequences.

```
double lz76InformationDistance(const sequence&,
                              const internal::LZ_Result&,
                              const sequence&,
                              const internal::LZ_Result&)
double lz76InformationDistance(const sequence&, const sequence&, utils::LZ_Args)
double lz76InformationDistance(const sequence& T1, const sequence& T2)
```

C++

[illegible]

Python

```
float metrics.information_distance(seq1: list | str,
                                   seq2: list | str,
                                   partitions: int = 1,
                                   alphabet: int = 2,
                                   log_base: int = 2,
                                   jobs: int = std::thread::hardware_concurrency())
```

```
# main.py
import lzcomplexity as lz

seq1 = "ATGGGCCTCACCGTGTCCGCGCTCTTTTCGCGGATCTTCGGGAAGAAGCAGATGCGGATTCTCATGGTTG"
seq2 = "GTTGGTACTCTTAGGCGTAGACGAAGAAGGGCTTCTAGGCGCTTTTCTCGCGCTGTGCCACTCCGGTA"
result = lz.metrics.information_distance(seq1, seq2)

print(f"Information distance: {result}")
```

5.3 Parallel annotations

All the API functions, implemented in C++ and Python, follow a high-performance parallel pattern. OneTBB assists in determining the number of threads required for each task. The abstraction layer, provided by OneTBB, over the thread creation and management layer, allows us to focus only on the processing tasks. We design the algorithms, implemented the necessary structures, define the parallel tasks with their parameters, and OneTBB handles the creation of background threads and the partitioning of the vectors for these threads.

As part of the library, we provided users with the ability to adjust the relevant parameter related to parallel processing. The `chunk` number within the structure `SA_Args` or the `subproblem_count` number in the `CaPS_SA` class. These two parameters are identical and are used by the same function to determine the number of partitions used to divide the string of symbols and process each partition in parallel. It's worth noting that the `chunk` value is used to internally initialize the `subproblem_count` within the API functions.

The number of partitions is a crucial parameter that can significantly impact the algorithm's execution performance. While a large number of partitions can overwhelm the system with excessive threads, a small number of partitions can overload the threads with a substantial amount of sequence processing. Therefore, we recommend following the default partition value that is determined based on the string length. These values have been validated through various performance tests conducted on a cluster node equipped with up to 16 CPUs (32 threads) and strings containing at least 10^9 symbols. In scenarios where there exist a larger number of CPUs and massive strings, the upper boundary could be tuned.

6 Foundations of the entropic measures and its estimation

A thorough discussion the foundations of the used entropic magnitudes and their estimation can be found in XXXXXXX and references therein, a summary of the main ideas is given in what follows.

6.1 Entropy Density (h) and Lempel-Ziv factorization

The concept of entropy was first formalized in 1948 by Claude Shannon as a measure of uncertainty in communication systems[3]. It is defined over a stochastic variable s that takes values from a set Σ of finite cardinality $\sigma = |\Sigma|$. The probability that the variable s takes the particular value $s_i (\in \Sigma)$ is denoted by p_i . The Shannon entropy $H(s)$ is then defined as:

$$H(X) = - \sum_i p_i \log p_i. \quad (1)$$

Shannon entropy quantifies the average "surprise" or unpredictability when observing the value of s . $0 \log 0 \equiv 0$ by convention.

- A deterministic system with $p_i = 1$ for a single symbol has $H(X) = 0$, indicating no randomness.
- A uniform distribution ($p_i = 1/\sigma$) maximizes entropy: $H(X) = \log \sigma$.

Consider a source of symbols and consider the set of bi-infinite sequence $\{S\}$ where $S = \dots s_{-2}s_{-1}s_0s_1s_2 \dots \in \{S\}$ is a string from the set. Consider all possible substrings from each string $S \in \{S\}$. A probability measure can be defined over the set of all substring and consequently, the entropy definition can be extended over such probability measure. If the source is ergodic, then the probability over the substrings of a single string S , is equal to the probability measure over the substrings of the set $\{S\}$ and the analysis can be reduced to any individual string in the set. The **block entropy** $H_n(S)$ is defined over the string S as:

$$H_n(S) = - \sum_{\{S_n\}} p_i^n \log p_i^n, \quad (2)$$

where the ordered $\{S_n\}$ denotes all possible substrings of length n in the string S , and p_i^n is the probability of occurrence of the i -th substring. Block entropy grows with n , reflecting the accumulation of information across longer subsequences. The difference

$$\Delta H_n = H_n - H_{n-1} \quad (3)$$

represents the incremental uncertainty when considering taking a step from considering substring of length $n - 1$ to considering substrings of length n .

The **entropy density** $h(S)$ captures the asymptotic randomness per symbol as $n \rightarrow \infty$:

$$h(S) = \lim_{n \rightarrow \infty} \frac{H_n(S)}{n}. \quad (4)$$

$h(S)$ represents the irreducible randomness of the system after accounting for correlations of all type and lengths. It also corresponds to the theoretical compression limit of the sequence [4]. For stationary processes, where the probability of a substring does not depend on the starting point of the substring in the infinite sequence, the entropy density can also be written as

$$h(S) = \lim_{n \rightarrow \infty} [H_n(S) - H_{n-1}(S)]. \quad (5)$$

To estimate the entropy density, we use the Lempel-Ziv complexity C_{Lz} (LZ-complexity)[5]. The LZ-complexity that it is the number of factor generated by an exhaustive factorization of a string. Consider a factorization of a sequence where each subsequence appears only once and there will never be a previous subsequence that contains the current subsequence.

Let us define the drop operator π which drops the last symbol of a finite string $S_i^q = s_i s_{i+1} s_{i+2} \dots s_q$ such that $(S_i^q \pi = S_i^{q-1})$, where $S_i^{q-1} = s_1 s_2 s_3 \dots s_{q-1}$.

Lempel-Ziv complexity

$F(S) = S_1^{l_1-1} S_2^{l_2-1} S_3^{l_3-1} \dots S_{l_m-1}^{l_m-1}$ is the exhaustive factorization of the sequence $S = s_0 s_1 s_2 \dots s_N$ if:

1. $S_{l_{k-1}}^{l_{k-1}-1} \pi \subset S_1^{l_k-1} \pi^2$
2. $S_{l_{k-1}}^{l_{k-1}-1} \not\subset S_1^{l_k-1} \pi$ except perhaps for the last factor $S_{l_m-1}^{l_m-1}$

where $S\pi^2 \equiv (S\pi)\pi$.

The Lempel-Ziv complexity $C_{LZ}(S)$ is the number of factors in the exhaustive story of the string $(= |F(S)|)$.

The Lempel-Ziv complexity is bounded

$$C_{LZ}(u) < \frac{N}{(1 - \varepsilon_N) \log N}, \quad (6)$$

where

$$\varepsilon_N = 2 \frac{1 + \log \log(\sigma N)}{\log N}. \quad (7)$$

Logarithm is taken in base σ .

C_{Lz} can be normalized by its asymptotic upper boundary as

$$c_{Lz}^N = \frac{C_{Lz}}{N / \log N}. \quad (8)$$

Ziv [4] proved that for an ergodic source the following holds

$$h(S) = \limsup_{N \rightarrow \infty} c_{Lz}^N(S). \quad (9)$$

Where h as before, is the entropy density of the string S . **Equation (9) is the basis for using the normalized Lempel-Ziv complexity $\hat{h}(S) \equiv c_{Lz}^N(S)$ as an estimate of the entropy density.**

6.2 Effective measure complexity (E)

The effective measure complexity (also known as excess entropy) $E(S)$ [6, 7, 8] is an entropic magnitude used in studies of dynamical systems and information theory [9]. It measures the intrinsic redundancy of an infinite sequence, i.e., the structural information that persists beyond the local randomness captured by the entropy density $h(S)$. Mathematically, excess entropy is defined as the cumulative sum of the differences between the block entropy $h_n(S)$ and the entropy density $h(S)$:

$$E(S) = \sum_{n=1}^{\infty} E_n(S) \equiv \sum_{n=1}^{\infty} [\Delta H_n(S) - h(S)]. \quad (10)$$

$E_n(S)$ reflects how much the uncertainty observed in blocks of length n exceeds the irreducible randomness $h(S)$ of the infinite sequence. This discrepancy arises because $\Delta H_n(S)$ overestimates $h(S)$, as it does not account for correlations at scales larger than n . As $n \rightarrow \infty$, $\Delta H_n(S)$ converges to $h(S)$, and the differences in the sum of (10) diminish to zero.

By substituting $\Delta H_n(S) = H_n(S) - H_{n-1}(S)$, the telescoping sum simplifies to:

$$E(S) = \lim_{n \rightarrow \infty} [H_n(S) - n h(S)]. \quad (11)$$

A large value of $E(S)$ indicates strong structure in the system: long-range correlations reduce the apparent entropy $h_n(S)$ as n increases, revealing underlying patterns not captured by small blocks.

Interpretation via Mutual Information. Effective measure complexity can also be interpreted as the mutual information between the past ($\overleftarrow{S}$) and future ($\overrightarrow{S}$) halves of a bi-infinite sequence, that is, if the bi-infinite string is divided in two halves ($\overleftarrow{S}, \overrightarrow{S}$) at any point the following holds:

$$E(S) = I(\overleftarrow{S} : \overrightarrow{S}). \quad (12)$$

Where $I(X : Y)$ is the mutual information between X and Y .

$$\begin{aligned} I(X : Y) &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X) \\ &= H(X) + H(Y) - H(X, Y). \end{aligned} \quad (13)$$

$E(S)$ represents the amount of historical information stored in the present that influences future behavior. If the accumulated information up to a given instant were abruptly lost, the process would appear $E(S)$ -bits more random, as the long-range correlations would be ignored.

Following equation (10) **the estimation of the effective measure complexity** follows from the expression[10]:

$$\hat{E}(S) = \sum_{m=1}^M (h(S^m) - h(S)) \quad (14)$$

where S^m is the result of dividing the sequence S into non-overlapping blocks of size m and randomly shuffling each block. In this way, correlations among sets of values larger than m are broken, but the frequency of symbols is preserved. Although $\hat{E}$ is not strictly equivalent to E , it exhibits a similar behavior (more on this below). The value of M is chosen as a function of the length of S under the condition $N \geq M2^M$ [10]. Transforming this equation into

$$M = \frac{\log(\frac{N}{M-1})}{\log(2)} \quad (15)$$

M can be estimated by means of the method of successive substitution.

We will call $\hat{E}$ Random Shuffling Effective Measure Complexity (RSEMC).

6.3 Information Distance (d)

The **information distance** quantifies the dissimilarity between two systems from an algorithmic perspective[11]. The Kolmogorov complexity $K(S)$ of a sequence S is defined as the length of the shortest program that generates S on a Universal Turing Machine (UTM)[12]. This measure captures the intrinsic randomness or compressibility of S :

- A constant sequence (e.g., $S = 0000\dots$) has low $K(S)$, as it can be generated by a trivial program.
- A maximally random sequence (e.g., coin toss outcomes) requires a program of length comparable to S itself, yielding $K(S) \approx |S|$.

Having two systems S and P , it is possible to define the conditional complexity $K(S|P)$ and the joint complexity $K(S, P)$. These quantities satisfy the symmetry relation [13]:

$$K(S, P) = K(S) + K(P|S) + \mathcal{O}(1) = K(P) + K(S|P) + \mathcal{O}(1), \quad (16)$$

where $\mathcal{O}(1)$ denotes an additive constant error independent of S and P .

Conditional Complexity and Joint Complexity

For two sequences S and P :

- $K(S|P)$: Minimal program length to generate S given P .
- $K(S, P)$: Minimal program length to generate both S and P .

The mutual information between S and P is defined as:

$$I(S : P) = K(P) - K(P|S) = K(S) - K(S|P),$$

which quantifies the reduction in complexity when one sequence is known.

The normalized information distance $d_K(S, P)$ can be written as [11]:

$$d(S, P) = \frac{\max \{K(S|P), K(P|S)\}}{\max \{K(S), K(P)\}} = 1 - \frac{I(S : P)}{\max \{K(S), K(P)\}}. \quad (17)$$

The last equality assumes, without loss of generality, that $K(P) > K(S)$. d satisfies the axioms of a metric (non-negativity, symmetry, identity, and triangle inequality) up to an additive precision term [14, 15]:

$$d(S, P) = 0 \iff S = P \quad (\text{identity}),$$

$$d(S, P) \leq d(S, Q) + d(Q, P) \quad (\text{triangle inequality}).$$

If two systems exhibit low Kolmogorov complexity and one can be derived from the other via a relatively short algorithm, then d between these systems is low. Conversely, for sequences lacking algorithmic correlation, the distance will yield high values.

Kolmogorov complexity is **uncomputable** due to Turing's halting problem theorem [13], thus, in practical terms, an alternative formulation for equation (16) is needed. Starting from the relation $K(S|P) = K(S, P) - K(P)$, we estimate $K(S|P)$ as $K(SP) - K(P)$, where SP denotes the concatenation of the two sequences (this introduces a logarithmic error, which becomes negligible for sufficiently long sequences [15]). d_K will then be:

$$d_K(S, P) \approx \frac{K(SP) - \min \{K(S), K(P)\}}{\max \{K(S), K(P)\}}. \quad (18)$$

If the two sequences being compared are of equal length, **we can use the estimated entropy density $\hat{h}$ instead of the Kolmogorov complexity to estimate the information distance between the two sequences as follows:**

$$\hat{d}(S, P) = \frac{\hat{h}(SP) - \min \{\hat{h}(S), \hat{h}(P)\}}{\max \{\hat{h}(S), \hat{h}(P)\}}. \quad (19)$$

7 Case studies

7.0.1 Brain connectivity

The human brain's function relies on a complex interplay of organization across multiple scales, from individual neurons to large-scale networks. The "connectome," or the map of structural and functional connections between brain regions, is fundamental to understanding cognition and emotion. Abnormalities in this network are linked to various neurological and psychiatric conditions, making its study crucial for diagnosis and treatment. However,

analyzing brain imaging data like fMRI to infer these connections is challenging; current methods often rely on strong assumptions, are limited to linear dynamics, or are highly sensitive to user-defined parameters, which can obscure the brain’s inherently non-linear behavior.

To overcome these limitations a novel, model-free analytical approach grounded in entropic measures was used. Instead of assuming linear correlations, the method uses a combination of entropy density and effective measure complexity to analyze brain signals. This approach can detect both linear and non-linear dynamics without requiring prior assumptions about the data. By characterizing brain regions through their unpredictability (entropy density) and memory (effective measure complexity) the framework offered a robust and generalizable method for analyzing cognitive states across different experimental conditions and signal types.

The fMRI data used in this study was sourced from the Human Connectome Project (HCP) and encompasses distinct task-based experiments, as well as resting-state scans. During resting-state runs, participants simply relaxed with their eyes open. The motor task involved cues for finger tapping, toe squeezing, or tongue movements. In the working memory exercise, participants viewed images of faces, places, tools, and body parts while performing either a 2-back memory task or a 0-back control task. The emotion task required participants to match either fearful/angry faces or simple shapes, while the language task alternated between listening to short stories and solving auditory math problems, each followed by comprehension questions.

All data underwent the standard HCP preprocessing pipeline, which includes motion correction, noise removal, alignment to a standard brain template, spatial smoothing, and temporal filtering to ensure consistency and quality. Following preprocessing, the brain was parcellated into 360 distinct Regions of Interest (ROIs) using the Glasser atlas, a high-resolution cortical map. This parcellation divides the brain into 180 regions in the left hemisphere and their 180 homologous counterparts in the right hemisphere, providing a detailed and standardized framework for subsequent analysis of brain activity across the different tasks.

The data is discretized before estimating the entropic magnitudes. For each time series, the mean value is computed and used as a threshold for binarization. Binarization results in the loss of information from the original signal but it is assumed that the binarized strings still carry enough relevant information of the dynamics of the system. If this is the case, it can only be asserted by the results themselves.

There is a clear difference between the two plots, and that is what we capture by using the entropy magnitudes. The complexity-entropy map plots the relationship between entropy density (h) and RSECM (E) for all brain regions involved in the experiment, as illustrated in Figure 11. Each dot represents a (h, E) tuple for one of the 360 ROIs. All values of entropy density h , in all tasks, are above 0.6 bit/symbol, which implies that none of the regions have a completely predictable or completely unpredictable dynamic. At the same time, the LZ-effective complexity is always above 0.45 bits/symbol, which means that some pattern formation is present on the fMRI signal. Both behaviors are common conditions for complex dynamics.

For a better visualization of the brain activity using entropy density, Figure 12 illustrates the activation levels of various brain regions during each stimulus, mapped onto the brain surface. The figure provides a detailed view of how different stimuli engage specific areas, highlighting the spatial distribution of neural activity. As previously discussed, lower values of h (represented in red) correspond to the most active regions involved in task processing, while higher h values (represented in blue) indicate lower activation levels.

7.0.2 Computation at the edge of chaos and locally coupled non-linear oscillators

Nonlinear coupled oscillators (NLCO) have been extensively studied in relation with many natural phenomena such as chemical reactions, neural brain activity, insects synchronization. A study was carried out of coupled non-linear oscillator using the entropy magnitudes described. In all cases, the estimates were obtained using `lzcomplexity`³. The idea of looking into complex systems as computational devices has a long history with a strong focus in the practical point of view, that is, seeking computation abilities as some realization of an algorithmic procedure. A broader definition of computation looks more into what nature does as information handling: storage, transmission, and production of information. It has been argued that near a transition to a chaotic regime, a system enhances its computational capacities and therefore, it is in this region, with critically tuned control parameters, that interesting behaviors of the systems must be explored. In this article, we returned to this hypothesis. We show that in non-linear locally coupled oscillators enhanced computation capabilities can be found near the edge to chaotic regimes. We do not focus in finding useful computations but instead uses the entropic measures of entropy density and

³For details see [17, 18] and references therein.

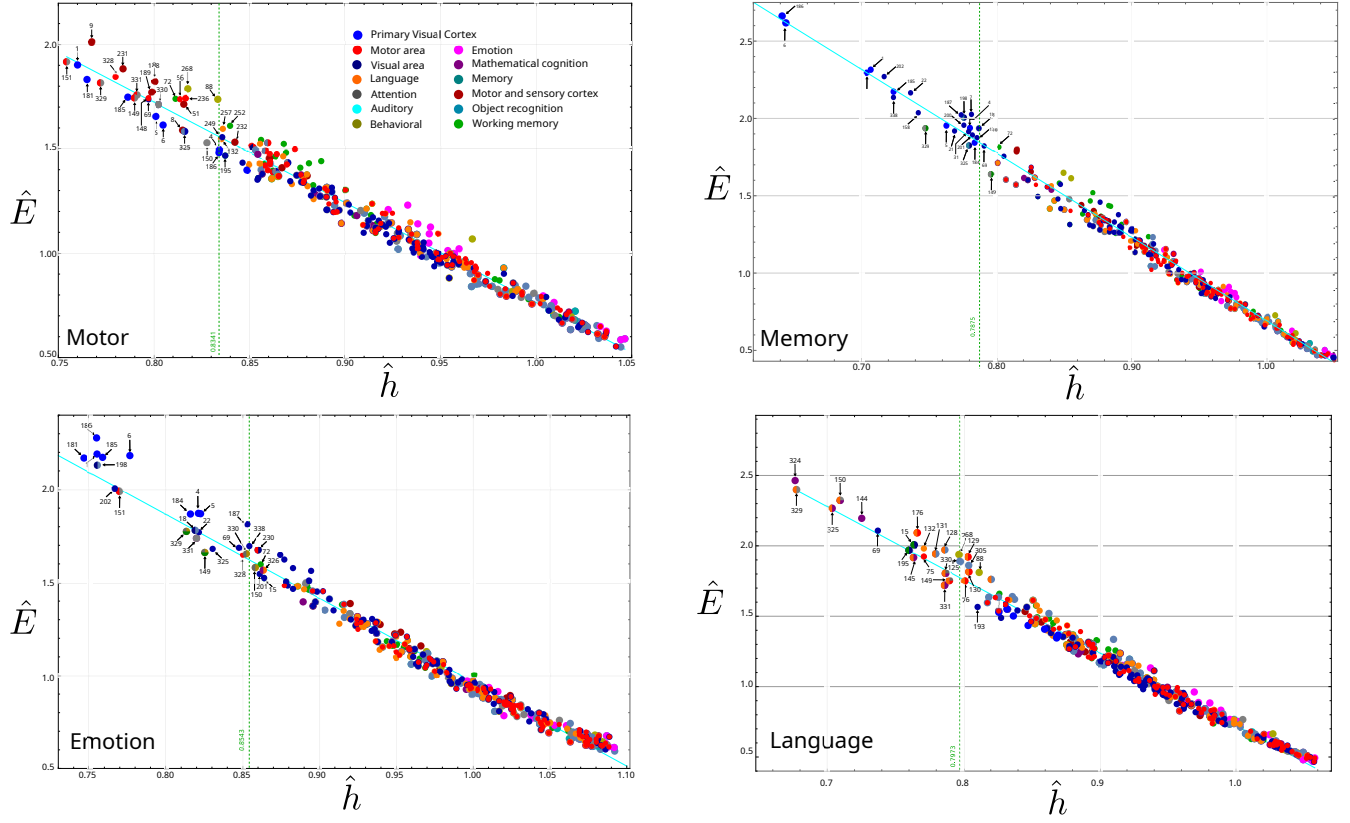


Figure 11: RSECM (E) vs entropy density (h) as calculated by **lzcomplexity** of all regions during the performance of the four tasks: motor, memory, emotion, and language. The active regions during each task present lower h and higher E , which points to less unpredictability and more patterned behavior than the rest of the regions. The results confirm that inactive regions have higher randomness as they do not suppress the noisy background [16]. The vertical line divides active from non-active areas.

effective measure complexity ($\hat{E}$) to show that in the verge of jumping into chaos, the transmission and structuring of information has a peak exposed by the sudden surge of $\hat{E}$.

The transition from regular towards chaotic behavior is one of the defining characteristics of complex non-linear dynamical systems. There has been a keen interest in the behavior of a dynamical system as it undergoes such transition, which comes from the fact that it is precisely in such region where enhanced or improved computational capabilities of the system are claimed to be found, this is known as the edge of chaos (EOC) hypothesis. The most common interpretation of computational capacity is related to the ability of the system to perform some “meaningful” task. Meaningful can be understood, given some initial condition, as the system transforming the input in some prescribed way into some output data. More broadly, it can be understood related to the system capability to behave as a universal Turing machine given some proper input and “programming”. A related but different point of view is to associate computation to the generic production, storage, transmission and manipulation of information. It is in this last sense that any dynamical system can be seen as some device amenable to be characterized by its intrinsic computational abilities. Measures of the computational capabilities of a system have been developed that aim at quantifying the efficiency, in terms of resources, while characterizing the balance between irreducible randomness and pattern production.

In favor of the EOC hypothesis several systems modeling natural and artificial phenomena exhibit interesting, if not some type of critical behavior, precisely at the edge of chaos. It is not clear, for some one dimensional systems, if the EOC criticality plays a role in the enhancement of computational capabilities.

The model used here consist in a system of locally coupled oscillators using, under the weak coupling assumption, a simply modified Adler equation. The system exhibits, for certain ranges of values of its control parameters, complex

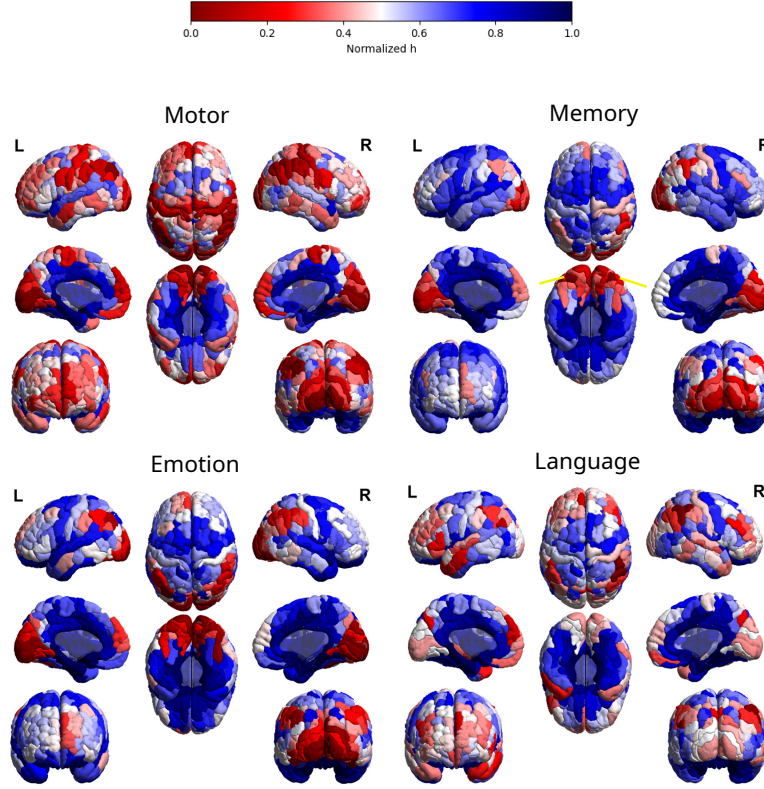


Figure 12: **Brain mapping of activity levels.** Level of activation by the normalized entropy density in each stimulus. At lower h (red color) the regions are more active than at higher h (blue color). Visual regions in the occipital area are strongly activated during motor, emotion recognition, and working memory stimuli, which have a visual cue. In contrast, during language stimuli, the visual areas are not active, but the auditory areas are. Observe the strong activation of the motor cortex in the posterior precentral gyrus, immediately anterior to the central sulcus, during the motor task but not in the others. Similarly, note the strong activation of the FFA and PIT -paraviscual areas specialized in face and object recognition, respectively- in the memory and emotion tasks, where faces are included in the stimuli; however, these areas are not active in the other tasks.

behavior. The array of coupled oscillators obeys, for the phase evolution, a system of equations given by

$$\frac{d\theta_i}{dt} = \omega + \gamma \cos \theta_i + (-1)^i [\cos \theta_{i-1} + \cos \theta_{i+1}]. \quad (20)$$

ω is the proper frequency of the oscillators, and γ controls the self-feedback of the oscillator relative to the coupling given by the third term in the right hand of equation (20). The number of oscillators N is taken even, and the alternating term in the coupling term is to guarantee balance in the interaction of the array of oscillators. Cyclic boundary conditions are enforced. The state of the system is given by the value, at a given time, of the phases θ_i of each oscillator which will be observed indirectly by the activity given by $\sin \theta_i$. A binary partition scheme is introduced: the mean value of the activity is measured for a given time step and used as a threshold to binarize the data.

Simulations were performed following with 5000 coupled oscillators that were left to evolve for 5000 discrete time steps while the activity of the oscillators was recorded. Averaging was performed over 30 different random initial phases configurations. The system of equations (20) was solved numerically using a Runge-Kutta method of order 4.

In the case of the oscillators, the variables (ω, γ) vary continuously, and the control parameters space is therefore

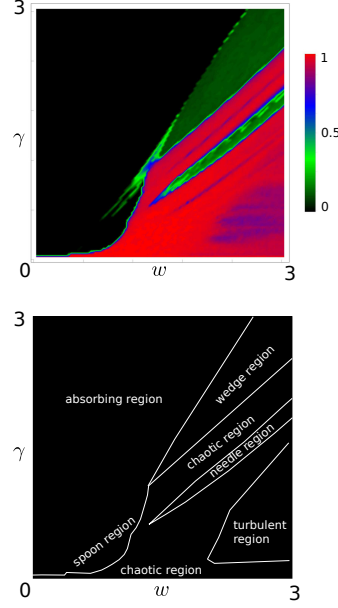


Figure 13: The entropy density estimates for the NLCO from the Lempel-Ziv complexity over the control parameters (ω, γ) space. In all cases the number of oscillators is $N = 10^4$ and 10^4 time steps are taken. (above) Corresponds to the h value from the final configuration of the oscillators after 10^4 steps. Each point is the average of 100 runs. (below) A diagram of the different regions identified in the two maps above. Taken from Ref. [19].

continuous. A detailed study of the “phase” diagram has been done using the entropy density $\hat{h}$ as the measured variable (Figure 13). There are several particular regions in the control parameter space. We focused on the needle-shaped region surrounded by a connected chaotic region (Region labeled as needle in Figure 13 below). Previous studies have indicated that the needle region exhibits low values of entropy rate and is insensitive to perturbations in the initial configuration of the oscillators, while the surrounding chaotic region has high entropy rate and high informational distance $\hat{d}$ values. We chose a fixed γ value of 1.2057 and let ω vary between 2.10 and 2.54, enough to make a transition from the needle region to the chaotic region. Figure 14 shows a peaked shaped maximum of E , at the same ω values where the entropy rate has a jump from low values to high disorder attaining the maximum possible value of $\hat{h} = 1$. The peaked maximum of $\hat{E}$ also coincides with the jump in the $\hat{d}$ values.

8 How well does the estimated magnitudes compare to the true values?

To test the accuracy of the Lempel-Ziv estimation of entropy we use finite-state process where the analytical value can be computed. Finite-state process is one that can be represented by a finite-state automaton (FSA). In a finite state automata of the type considered here, a finite number of states Λ are connected by a transition from one state to another. The transition from one state to another happens with a certain probability, and upon such a transition, a unique symbol is emitted. For a given state Λ_i , $T^a k i$ represents the probability for transiting from state i to state k while emitting a symbol a . We will be interested in the class of processes where, given a state S_i , the emitted symbol a determines the state to which the transition happens (unifilar), and we can write $T^a i = T^a_{ki}$. The process starts at an initial state, and at each step, a transition to another state is taken while a symbol is emitted. At step n , the probability that the process is at state S_i is denoted by $p_i^{(n)}$. The asymptotic probability of state is denoted by $p_i^{(\infty)}$.

If one considers, among all possible FSA, only the unifilar and minimal Markov chains [20], then.

$$h = - \sum_i \sum_a p_i^{(\infty)} T^{(a)}_{i} \log T^{(a)}_{i},$$

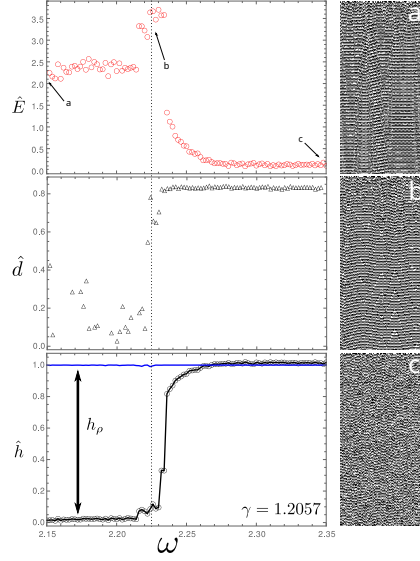


Figure 14: $\hat{E}$ (upper left plot) and Lempel-Ziv information distance $\hat{d}$ (middle left plot) and entropy density $\hat{h}$ (lower left plot), for an array of 5000 non-linear coupled locally coupled oscillators following an Adler type equation. A partition over the phase activity is taken using the mean value of the phases. h_ρ is the entropy rate for the random arrangement having the same symbol density as the corresponding sequence of the NLCO for the given ω value. Again note the clear jump in entropy rate $\hat{h}$ and $\hat{d}$ around $\omega \approx 2.23$ just after $\hat{E}$ reaches a maximum. Averaging was performed over 30 random initial phase configurations for the oscillators. On the right, the spatiotemporal diagrams at the points marked by arrows in the upper left plot.

and

$$E = \sum_{n=0}^{\infty} \sum_i \sum_a (p_i^{(\infty)} - p_i^{(n)}) T^{(a)}_i \log T^{(a)}_i$$

We have used two Markovian processes as examples: The even process and the 2-state symmetric process.

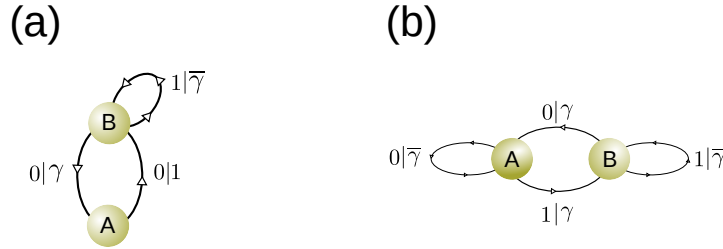


Figure 15: Diagram of the Finite State Machines for the (a) even process and (b) two state symmetric process.

The even process can be represented by a finite state automata as depicted in Figure 15(a). It generates binary strings with an even number of 0, the occurrence of a pair of 0 happens with probability γ , while the occurrence of the symbol 1 happens with probability $\bar{\gamma} = 1 - \gamma$.

The two state symmetric process (2ssp) is shown in Figure 15(b). It is a two-state FSA where the probability of transitioning from one state to the other is given by γ , and, correspondingly, the probability of staying in the same state is $\bar{\gamma} = 1 - \gamma$.

The left column of Figure 16 are the plots of the estimated entropy density $\hat{h}$ from the Lempel-Ziv factorisation for different sequence lengths in the already presented FSA corresponding to the even process (upper) and symmetric 2-state process (lower) for all values of their control parameters. First, the estimate follows the general trend even for sequence lengths as short as 256 symbols, and it's already indistinguishable from the exact entropy

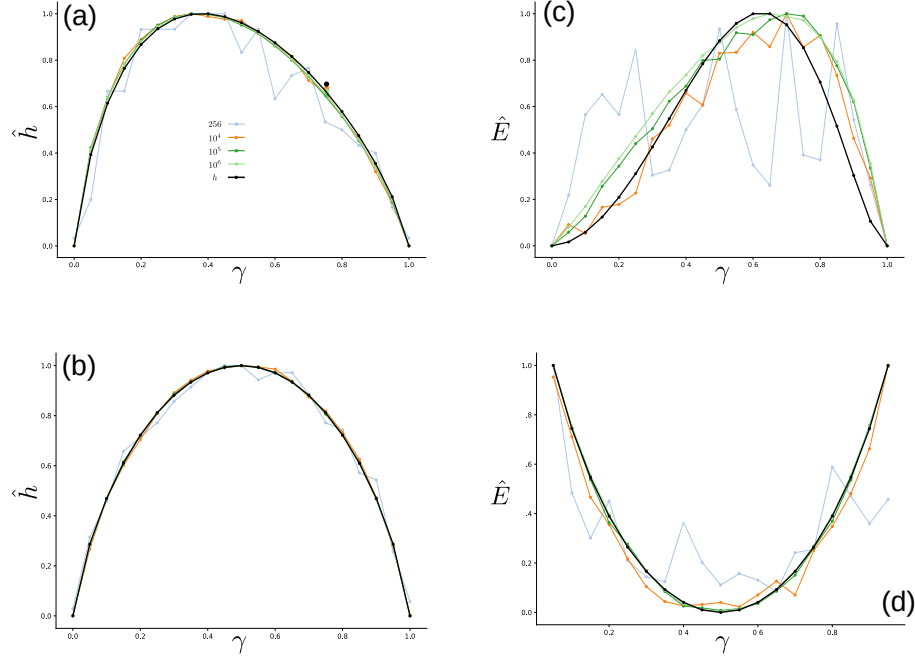


Figure 16: Lempel-Ziv estimation of the entropy density and the effective measure complexity. On the left, the estimation of the entropy density $\hat{h}$ using the LZ-complexity for strings of different sizes. In both examples, the black line is the true entropy density as a function of the transition probability γ . (a) corresponds to the even process; for string lengths above 10^3 , the estimated values are almost indistinguishable from the true values. The same holds for the two-state symmetric process (b), where the estimated values are fairly close even for string lengths as short as 256. On the right, similar plots but for the estimation of the effective measure complexity are shown. The estimated $\hat{E}$ values are scaled to lie between 0 and 1. The results for $\hat{E}$ are less satisfactory for short string lengths, but heavily depend on the nature of the data. However, for the two examples, for string lengths $\geq 10^5$, the estimated curve follows the trend of the real effective complexity measure. For the even process (c), string lengths of 256 are insufficient for any reliable estimation, 10^4 already shows the general behaviour of $\hat{E}$, and above that length, the fluctuations in the estimated curves have reduced significantly, and the estimation is not far off the true value. For the two-state symmetric process (d), except for string length of 256, all other lengths above that value are close to the real value, being indistinguishable for 10^5 and 10^6 .

density h for sequence lengths above 10^4 symbols. For the two-state symmetric process, the estimate is even better as it behaves almost identically to the h for sequences of 256 symbols and above.

The right column of Figure 16 shows $\hat{E}$ and the correct effective measure complexity for the two FSA processes studied above. The estimated curves are scaled to the interval $[0, 1]$, which is not a significant limitation, as in any analysis, it is the functional behaviour that is interesting, not the absolute values. Both processes, but more evidently the even process, show that the estimation for short sequences ($N = 256$) is poor. The estimated curve is far from smooth, and large fluctuations in the values are evident. This is no surprise: the convergence of equation (14) depends on the number of summands, and yet, for such short sequence length, there are not many values of l for which shuffling makes sense. For a sequence of length $N = 10^4$, the $\hat{E}$ behaviour already resembles the actual exact curve, and although some fluctuation is observed in both FSA, more significant in the even process, the overall trend follows the correct functional dependence, which is quite clear for the two-state symmetric process. For lengths $N > 10^5$, fluctuations are mostly suppressed, and for the two-state symmetric FSA, the estimated curve is almost indistinguishable from the real curve. It is interesting that in the case of the even process, although the form of the curve is correct, the maximum is shifted and at low values of entropy density, $\hat{E}$ overestimates the true E value.

9 License (MIT license)

lzcomplexity - stand alone software and c++/python library for the entropic analysis of time series
Copyright (C) 2026 Pleros Group

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

References

- [1] J. Khan, T. Rubel, L. Dhulipala, E. Molloy, and R. Patro, "Fast, Parallel, and Cache-Friendly Suffix Array Construction," pp. 21 pages, 1 002 465 bytes, 2023.
- [2] I. Corporation, "Intel(R) oneAPI Threading Building Blocks (oneTBB) Documentation," <https://www.intel.com/content/www/us/en/develop/documentation/onetbb-documentation/top.html>, 2023.
- [3] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. Hoboken, N.J: Wiley-Interscience, 2006.
- [4] J. Ziv, "Coding Theorems for Individual Sequences," *IEEE Transactions on Information Theory*, vol. 24, no. 4, pp. 405–412, Jul. 1978.
- [5] A. Lempel and J. Ziv, "On the Complexity of Finite Sequences," *IEEE Trans. Inform. Theory*, vol. 22, no. 1, pp. 75–81, Jan. 1976.
- [6] P. Grassberger, "Toward a quantitative theory of self-generated complexity," *Int J Theor Phys*, vol. 25, no. 9, pp. 907–938, Sep. 1986.
- [7] D. P. Feldman and J. P. Crutchfield, "Discovering Noncritical Organization: Statistical Mechanical, Information Theoretic, and Computational Views of Patterns in One-Dimensional Spin Systems," *Santa Fe Institute Working Paper 98-04-026*, 1998.
- [8] J. P. Crutchfield and D. P. Feldman, "Regularities unseen, randomness observed: Levels of entropy convergence," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 13, no. 1, pp. 25–54, Mar. 2003.
- [9] J. P. Crutchfield, "Between order and chaos," *Nature Phys*, vol. 8, no. 1, pp. 17–24, Jan. 2012.
- [10] O. Melchert and A. K. Hartmann, "Analysis of the phase transition in the two-dimensional Ising ferromagnet using a Lempel-Ziv string-parsing scheme and black-box data-compression utilities," *Phys. Rev. E*, vol. 91, no. 2, p. 023306, Feb. 2015.
- [11] P. M. B. Vitányi, F. J. Balbach, R. L. Cilibrasi, and M. Li, "Normalized Information Distance," in *Information Theory and Statistical Learning*, F. Emmert-Streib and M. Dehmer, Eds. Boston, MA: Springer US, 2009, pp. 45–82.
- [12] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, 1993.
- [13] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, ser. Texts in Computer Science. Cham: Springer International Publishing, 2019.

- [14] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi, “The Similarity Metric,” *IEEE Trans. Inform. Theory*, vol. 50, no. 12, pp. 3250–3264, Dec. 2004.
- [15] E. Estevez-Rams, R. L. Serrano, C. A. J. Nunes, and B. Aragon-Fernandez, “Lempel-Ziv complexity analysis of one dimensional cellular automata,” *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 25, no. 12, p. 123106, Dec. 2015.
- [16] T. Ito, S. L. Brincat, M. Siegel, R. D. Mill, B. J. He, E. K. Miller, H. G. Rotstein, and M. W. Cole, “Task-evoked activity quenches neural correlations and variability across cortical areas,” *PLOS Computational Biology*, vol. 16, no. 8, p. e1007983, 2020.
- [17] E. Estevez-Rams, D. Estevez-Moya, K. Garcia-Medina, and R. Lora-Serrano, “Computational capabilities at the edge of chaos for one dimensional systems undergoing continuous transitions,” *Chaos*, vol. 29, p. 043105, 2019.
- [18] D. Estevez-Moya, E. Estevez-Rams, and H. Kantz, “Complexity and transition to chaos in coupled adler-type oscillators,” *Phys. Rev. E*, vol. 107, p. 044212, 2023.
- [19] E. Estevez-Rams, D. Estevez-Moya, and B. Aragon-Fernandez, “Phenomenology of coupled nonlinear oscillators,” *Chaos*, vol. 28, pp. 23110–23121, 2018.
- [20] D. Zambella and P. Grassberger, “Complexity of forecasting in a class of simple models.” *Complex Systems*, vol. 2, pp. 269–303, 1988.