




A Python/Zig optimized and customizable implementation for the ρ_{DCCA} and DMC_x^2 methods

Fernando Ferraz Ribeiro 
Universidade Federal da Bahia

Gilney Figueira Zebende 
Universidade Estadual
de Feira de Santana

Abstract

This paper presents the **Zebende**, a Python package written in Python and Zig, that calculates the *DFA*, *DCCA* ρ_{DCCA} and the DMC_x^2 . The package presents an optimized algorithm that significantly improves the calculations speed. A comparison with other packages that calculates the .The package is also the first to implement the DMC_x^2 coefficient for any number of series.

Keywords: ρ_{DCCA} , DMC_x^2 , optimization, Python, Zig.

1. introduction

The ρ_{DCCA} (Zebende 2011) is a widely used coefficient that measures the cross-correlation between two non-stationary time series. It's an extension of the Detrended Fluctuation Analysis (*DFA*) (Peng, Buldyrev, Havlin, Simons, Stanley, and Goldberger 1994) and the Detrended Cross-correlation Analysis (*DCCA*) (Podobnik and Stanley 2008): while the *DFA* calculates the self-affinity and long-memory properties of a time series data, and the *DCCA* analyses power-law cross correlations between two different non-stationarity time series, the ρ_{DCCA} coefficient quantifies this cross-correlation in simple values ranging from -1 to 1 , where -1 indicates a perfect anti-correlation between the series, 1 a perfect correlation and zero (0) no correlation at all.

The Detrended Multiple Cross-Correlation Coefficient (Zebende and Silva 2018) (DMC_x^2) is a generalization of the ρ_{DCCA} coefficient that correlates one time series (dependent variable) a number of time series (independent variables). The DMC_x^2 values ranges from zero (0), indicating no correlation to 1 , meaning perfect correlation or anti-correlation between the dependent and the independent variables.

This paper presents the **Zebende** Python package, an implementation of the *DFA*, *DCCA*, ρ_{DCCA} , DMC_x^2 and utility functions related to the methods. In section 2 the steps for calculating the ρ_{DCCA} and DMC_x^2 are presented and discussed. Section 3 shows how this library was implemented, the optimization technics and the recommended steps to use the library. In Section 4 the **Zebende** package is compared with other packages for Python and R that calculates the ρ_{DCCA} in terms of performance and usability, leading to the conclusions in Section 5.

2. Algorithms of the coefficients

The algorithms that calculates the ρ_{DCCA} uses the *DFA* and the *DCCA* steps. The DMC_x^2 coefficient uses the ρ_{DCCA} coefficient and, consequently, also embraces the *DFA* and the *DCCA*. The *DFA* method is described in six steps:

1. Taking a time series $\{x_i\}$ with i ranging from 1 to N , the integrated series X_k is calculated by $X_k = \sum_{i=1}^k [x_i - \langle x \rangle]$ with k also ranging from 1 to N ;
2. the X_k series is divided in $N - n$ boxes of size n (time scale), each box containing $n + 1$ values, starting in i up to $i + n$;
3. for each box, a polynomial (usually of degree 1) best fit is calculated, getting $\tilde{X}_{k,i}$ with $i \leq k \leq (i + n)$;
4. in each box is calculated: $f_{DFA}^2(n, i) = \frac{1}{1+n} \sum_{k=i}^{i+n} (X_k - \tilde{X}_{k,i})^2$
5. for all the boxes of a time scale, the *DFA* is calculated as:

$$F_{DFA}(n) = \sqrt{\frac{1}{N-n} \sum_{i=1}^{N-n} f_{DFA}^2(n, i)};$$

6. for a number of different timescales (n), with possible values $4 \leq n \leq \frac{N}{4}$ the F_{DFA} function is calculated to find a relation among $F_{DFA} \times n$

The *DCCA* method is very similar to the *DFA* calculations, with the difference of analyzing two series while the *DFA* evaluate properties of a single time series. It's also a six steps process:

1. Taking two time series with the same length $\{x\alpha_i\}$ and $\{x\beta_i\}$ with i ranging from 1 to N , the integrated series $X\alpha_k$ and $X\beta_k$ are calculated by $X_k = \sum_{i=1}^k [x_i - \langle x \rangle]$ for each series, with k also ranging from i to N ;
2. $X\alpha_k$ and $X\beta_k$ series are divided in $N - n$ boxes of size n (time scale), each box containing $n + 1$ values, starting in i up to $i + n$;
3. for each box, a polynomial (usually of degree 1) best fit is calculated, getting $\widetilde{X\alpha}_{k,i}$ and $\widetilde{X\beta}_{k,i}$, for series $\{x\alpha_i\}$ and $\{x\beta_i\}$ respectively, with $i \leq k \leq (i + n)$;
4. in each box is calculated: $f_{DCCA}^2(n, i) = \frac{1}{1+n} \sum_{k=i}^{i+n} (X\alpha_k - \widetilde{X\alpha}_{k,i}) \times (X\beta_k - \widetilde{X\beta}_{k,i})$

5. for all the boxes of a time scale, the $DCCA$ is calculated as:

$$F_{DCCA}^2(n) = \frac{1}{N-n} \sum_{i=1}^{N-n} f_{DCCA}^2(n, i);$$

6. for a number of different timescales (n), with possible values $4 \leq n \leq \frac{N}{4}$ the F_{DCCA}^2 function is calculated to find a relation among $F_{DCCA}^2 \times n$

Comparing the algorithms, the first three are basically identical, the only difference is that the $DCCA$ method apply those steps to two series. The step four of the DFA can be considered an analogous of the variance, replacing the average subtraction (in the variance) for the values obtained by the polinomial fit (estimated seires); and the equivalent step of the $DCCA$ is, in teh same terms, compared to the covariance between the two series. Step five calculates the square root of the mean of the values calculates in the previous step (f_{DFA}^2 in each box for the DFA , in the $DCCA$, the mean of the values evaluated each box is calculated in stead. The last step, in both cases, is more a reminder to repeat the respective previous operations for a number of difference time scales.

The ρ_{DCCA} is measured using Eq. 1. Considering the relation between DFA and variance and $DCCA$ and covariance, the ρ_{DCCA} resembles Pearson correlation for a time scale n .

$$\rho_{DCCA}(n) = \frac{F_{DCCA}^2(x\alpha, x\beta)(n)}{F_{DFA}(x\alpha)(n) \times F_{DFA}(x\beta)(n)} \quad (1)$$

The DMC_x^2 is a generalization of the ρ_{DCCA} that calculates the correlation between one time-series $\{Y\}$, as the dependent variable, and a number j of time-series $\{X_1\}$, $\{X_2\}$, $\{X_3\}$, ..., $\{X_j\}$ defined as independent variables. The coefficient is expressed mathematically as:

$$DMC_x^2 \equiv \rho_{Y, X_i}(n)^T \times \rho^{-1}(n) \times \rho_{Y, X_i}(n) \quad (2)$$

In Eq. 2, $\rho^{-1}(n)$ represent the inverse of a matrix populated by all possible combinations of ρ_{DCCA} between independent variables. In Eq. 3, value $\rho_{X_1, X_2}(n)$, for instance, is the ρ_{DCCA} for independent variables X_1 and X_2 calculated with time scale n , occupying position ρ_{12} of the matrix. Two fundamental characteristics: the first is that the main diagonal values are all ones, since it's position in the matrix denotes the calculation of a correlation between a series and itself. Second, the matrix is symmetric in relation to the main diagonal, as the ρ_{DCCA} is evaluated a commutative expression.

$$\rho^{-1}(n) = \begin{pmatrix} 1 & \rho_{X_1, X_2}(n) & \rho_{X_1, X_3}(n) & \dots & \rho_{X_1, X_j}(n) \\ \rho_{X_2, X_1}(n) & 1 & \rho_{X_2, X_3}(n) & \dots & \rho_{X_2, X_j}(n) \\ \vdots & \vdots & \vdots & \dots & \vdots \\ \rho_{X_j, X_1}(n) & \rho_{X_j, X_2}(n) & \rho_{X_j, X_3}(n) & \dots & 1 \end{pmatrix}^{-1} \quad (3)$$

At last Eq. 4 represent the transposed vector of the $\rho_{Y, X_i}(n)$ between the depended variable $\{Y\}$ and each $\{X_i\}$ independent variable for a given time scale n .

$$\rho_{Y, X_i}(n)^T = [\rho_{Y, X_1}(n), \rho_{Y, X_2}(n), \dots, \rho_{Y, X_j}(n)] \quad (4)$$

The ρ_{DCCA} and the DMC_x^2 should be evaluated in a number of time scales (n) to analyze the characteristics of each coefficient.

3. Zebende package: implementation and optimization

The implementation of the **Zebende** package follows some well defined goals:

1. Enhance performance;
2. avoid redundant calculations;
3. make the outputs compatibles with other data analyses tools (including data manipulation, machine learn and statistical packages);
4. create a customizable and modular set of tools;
5. facilitate package evolution and maintenance;
6. deliver an easy to use package.

The Python language was chosen because it's one of the most used languages in the data analyses field and have a great support for statistical tools and machine learning algorithms. There are a plethora of tools to load and manipulate data (**Pandas**, **Polar**, **PySpark** ...), execute statistical analyzes (**Numpy**, **SciPy**, **StatsModels** ...), machine learning (**Pytorch**, **TensorFlow**, **Scykit Learn**...) and data visualization((**Matplotlib**), **Seaborn**, **Plotly**) among other related applications.

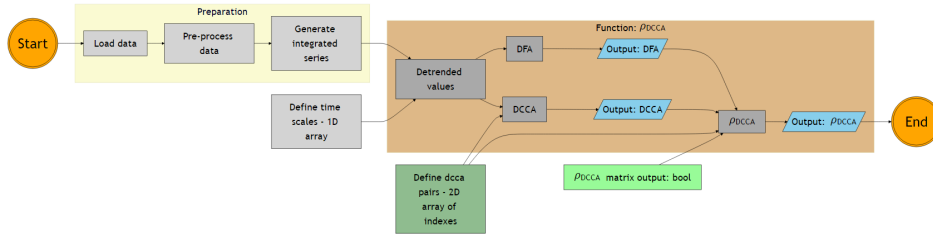


Figure 1: Calculating ρ_{DCCA} with **Zebende** package - Simplified flowchart

The first draft of the code was written in pure Python, to rapidly prototype the way users will interact with the package. Figures 1 and 2 presents simplified flowcharts illustrating how to use the package and how the main functions (ρ_{DCCA} and DMC_x^2) works.

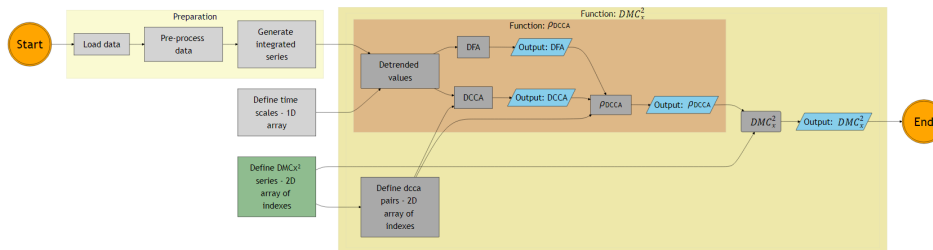


Figure 2: Calculating DMC_x^2 with **Zebende** package - Simplified flowchart

The preparation steps are the same in both functions. First the data is loaded, and should be analyzed by the researchers. Base on the data characteristics, the set should be treated to

ensure the methods requirements in the "Pre-processing" stage. The package functions expects data as ma matrix with the columns as the series and the lines as time steps. Columns unused in the indented research should also be dropped for better performance os the algorithms in this step. The more common way to do that is to use a data manipulation package. To proceed to the next step, the data table should be in the form of a **Numpy** 2D array, and this could be done with any data manipulation **Python** package. The next step is to calculate the integrated series. The package provides a function, named `integrated_series()`, to calculate that. The code example below show how to load the libraries (using **Pandas** as the data manipulation packages and loading a `.csv` file as a generic example), convert to **Numpy** array and generate the integrated series.

```
# importing packages
import numpy as np
import pandas as pd
import zebende as zb

data = pd.read_csv('path_to_the_file.csv') # loading data
# Pre-processing data
# ...
data = data.to_numpy(data) #converting data to Numpy array
int_data = zb.integrated_series(data) # calculating the integrated series
```

The option of taking out the integrated series generation from the main functions to an independent one was inspired by [Peng *et al.* \(1994\)](#) work, where the way of calculating integrated series was different from the one that is widely used in more recent years. The integration of the series is essentially a pre-processing step, and this approach makes easy to explore alternative ways to integrate the series or compare [Peng *et al.* \(1994\)](#) approach to the current one in different scenarios.

The input and output structures of each function are displayed below:

```
def p_dcca(
    input_data: NDArray[float64],
    tws: NDArray[int64] | NDArray[float64],
    DCCA_of: ndarray | Literal['all'] = "all",
    P_DCCA_output_matrix: bool = False
) -> tuple[NDArray[float64], # DFA
           NDArray[float64], # DCCA
           NDArray[float64] # P_DCCA
          ]

def dmcx2(
    input_data: NDArray[float64],
    tws: NDArray[int64] | NDArray[float64],
    dmcx2_of: ENUM_DMCx2_of | NDArray[float64] | list = 'all-full'
) -> tuple[ NDArray[float64], # DFA
           NDArray[float64], # DCCA
```

```

        NDArray[float64],    # P_DCCA
        NDArray[float64],    # DMC
    ]

```

The first two inputs are the same for functions `p_dcca()` and `dmcx2()`: `input_data` receive the integrated series and the `tw` receives an 1D array representing the time scales (box size) described in the algorithms on Section 2. The `input_data` is a 2D array of 64 bits floating point data. The `tw` accepts integers and, for convenience reasons, also floating points. Since the size of the boxes needs to be integers, in case of floating points, the values will be converted to integers by ignoring the decimal values(truncating). This two inputs are colored in light gray in Figures 1 and 2, indicating mandatory inputs.

With the mandatory steps explained, some very important optional inputs should be addressed. Starting with the ρ_{DCCA} function (dark green node in Figure 2) represents the input `DCCA_of` of the function. This input requires a 2D array of integers, each row is a pair of index, related to the `data_input` matrix. For example: if the `input_data` receives a four columns matrix, with index ranging from 0 to 3, that is intended to calculate de *DFA* for all the series but the ρ_{DCCA} between series of index 0 and 1 and also for series 2 and 3, the `DCCA_of` input should receive the array `[[0,1], [2,3]]`. If no value (or the string 'all') is given, the function will calculate all possible combinations of *DCCA* calculations between all the series respecting the index values order, as below:

```
[[0,1], [0,2], [0,3], [1,2], [1,3], [2,3]]
```

It's important to understand the calculation steps, the role of the `DCCA_of` array and how it fits in the goals of the package implementation. The code below is part of the Python implementation of the ρ_{DCCA} function,

```

for n_index, n in enumerate(tws): # for each time scale
    # temporary allocation arrays
    f2dfa_n = np.full(shape=(data.shape[0] - n, data.shape[1]),
        fill_value=np.nan, dtype=data.dtype)
    dcca_n = np.full(shape=(data.shape[0] - n, DCCA_of.shape[0]),
        fill_value=np.nan, dtype=data.dtype)
    detrended_mat = np.full(shape=(n + 1, data.shape[1]),
        fill_value=np.nan, dtype=data.dtype)
    for i in range(data.shape[0] - n): # for each box
        detrended_series( # inputs
            time_steps[i : i + (n + 1)], # arr_x
            data[i : i + (n + 1), :], # mat_y
            detrended_mat, # output
        )
        f2dfa_n[i] = np.power(detrended_mat, 2).mean(axis=0)
        for j, pair in enumerate(DCCA_of): # for each DCCA pair
            dcca_n[i, j] = (detrended_mat[:, pair[0]] * detrended_mat[:, pair[1]]
                ).mean(axis=0)
    F_DFA_arr[n_index, :] = np.sqrt(f2dfa_n.mean(axis=0))

```

```

DCCA_arr[n_index, :] = dcca_n.mean(axis=0)
# calculation of P_DCCA
P_DCCA_output_function(n_index, DCCA_of, F_DFA_arr, DCCA_arr, # Inputs
P_DCCA_arr) # Output

```

The first `for` loop in the code operates over the values of the `tw`s input, asserting that the step 6 of the *DFA* and *DCCA* methods in Section 2. Three temporary arrays are allocated for each time scale. The first, `f2dfa_n`, to store the calculations of the step 4 of the *DFA*. The number of lines of this array correspond to the number of boxes in the current time scale ($N - n$) and the columns is the number of series in the analysis. Array `dcca_n`, holds the values calculated in the step 4 of the *DCCA*. The number of lines also correspond to the number of boxes but the number of columns equals the number of pairs (rows) in the `DCCA_of` input. The `detrended_mat` array has a number of lines is the number of points in a time scale box ($n + 1$) and column count also equal to the number of time series. This last array is very important to the implementation goals.

In each box, the `detrended_series` function execute a one degree polynomial fit, subtract the values of the serie with the value given by the interpolated curve and stores this values in the `detrended_mat`. After the *DFA* is calculated. With all the detrended values calculated, another loop, for all the rows in the `DCCA_of`, uses the `detrended_mat` and the *DFA* arrays to calculate the *DCCA*. This approach avoid redundant calculation. If the `DCCA_of` is `[[0, 1], [0, 2], [1, 2]]`, the detrended and *DFA* values for each serie are calculated just once. If in a set of four series, a research intend to calculate the *DCCA* between the first serie and all the others along with the two last series, inputting `DCCA_of = [[0, 1], [0, 2], [0, 3], [2, 3]]`. In this case, the `detrended_mat` will be calculated just once for each series and the *DCCA* between series `[[1, 2], [1, 3]]` will be skipped, revealing a very customizable algorithm.

The function named `P_DCCA_output_function` in the code above, is a pointer to other functions. One that outputs the ρ_{DCCA} results in the form of a table (rows for each time scale and columns for each `DCCA_of` pair) and the other outputs it the form of a 3D matrix, where each level is the matrix in Equation 3 for one of the time scales. This behavior is driven by the `P_DCCA_output_matrix` input (represented as the light green node in Figure 1), where `False` means table output and `True` matrix output. This is very convenient for calculating the DMC_x^2 . There are two utility functions to transform a table output in a matrix one (`p_dcca_simple_to_matrix`) and also the other way around (`p_dcca_matrix_to_simple`).

The `dmcx2` function runs the `p_dcca()` with `P_DCCA_output_matrix` set to `True`. There is no `DCCA_of` input for the DMC_x^2 function, instead there is a `dmcx2_of` parameter. This input receives a 2D matrix where each line represents the indexes of the series to be used in Equation 2. The first value of each row is the index of the serie used as the dependent variable, the others, the independent ones. There are two literal strings that can be used, for convenience as inputs for this parameter: `'all-full'`, that generates a 2D array with every series as the dependent variable against all the others; and `'first-full'`, with only one row, having the index zero series as the dependente variable in relation with all the others. The `'all-full'` option is operated by the function `dmc_of_all_as_y`.

There is a restriction to the `dmcx2_of` array: the independent values must be in crescent order. There is a check in the `dmcx2` that raises an error if the values are out of order, and suggests the use of a utility function named `ordering_x_dmcx2_of`. The code below shows an example of a manually defined `dmcx2_of` array with 2 elements out of order in the second row,

the `AssertionError` output, the way of fixing it with the `ordering_x_dmcx2_of` function and finally the outputted ordered array.

```
dmcx2_of = np.array([ [0, 1, 2, 3],
                      [1, 0, 3, 2],
                      [2, 0, 1, 3],
                      [3, 0, 1, 2] ])

dfa , dcca, pdcca, dmc = zb.dmcx2(int_data, tws, dmcx2_of=dmcx2_of)

AssertionError:
dmcx2_of x values out of order:
use zebende.ordering_x_dmcx2_of(dmcx2_of)
to fix it before passing the dmcx2_of values to zebende.dmcx2() function"

dmcx2_of = ordering_x_dmcx2_of(dmcx2_of)
print(dmcx2_of)

[[0 1 2 3]
 [1 0 2 3]
 [2 0 1 3]
 [3 0 1 2]]
```

4. Results

5. Summary and discussion

References

- Peng CK, Buldyrev SV, Havlin S, Simons M, Stanley HE, Goldberger AL (1994). “Mosaic Organization of DNA Nucleotides.” **49**(2), 1685–1689.
- Podobnik B, Stanley HE (2008). “Detrended cross-correlation analysis: A new method for analyzing two nonstationary time series.” *Physical Review Letters*, **100**(8). ISSN 00319007. doi:10.1103/PhysRevLett.100.084102. 0709.0281.
- Zebende GF (2011). “DCCA cross-correlation coefficient: Quantifying level of cross-correlation.” *Physica A: Statistical Mechanics and its Applications*, **390**(4), 614–618. ISSN 03784371. doi:10.1016/j.physa.2010.10.022. URL <http://dx.doi.org/10.1016/j.physa.2010.10.022>.
- Zebende GF, Silva AM (2018). “Detrended Multiple Cross-Correlation Coefficient.” *Physica A*, **510**, 91–97. ISSN 0378-4371. doi:10.1016/j.physa.2018.06.119.

Affiliation:

Fernando Ferraz Ribeiro
Universidade Federal da Bahia
Faculty of Architecture
Universität Innsbruck
Universitätsstr. 15
6020 Innsbruck, Austria
E-mail: fernando.ribeiro@ufba.br
and
Centro Universitário Senai-Cimatec