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 tha **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 ρ_{DCCA}. The package is also the first to implement the DMC_x^2 coefficient in a package and the first algorithm to calculates it for any number of time series.

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

1. introduction

The Detrended Cross-correlation Coefficient (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-stationary 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_{DCCCA}, DMC_x^2 and utility functions related to the methods. In section 2 the steps for calculating the ρ_{DCCCA} 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 $\widetilde{X}_{k,i}$ with $i \leq k \leq (i + n)$;

4. in each box is calculated: $f_{DFA}^2(n, i) = \frac{1}{1+n} \sum_{k=i}^{i+n} (X_k - \widetilde{X}_k)^2$

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: $f_{DCCA}^2(n, i) = \frac{1}{1+n} \sum_{k=i}^{i+n} (X\alpha_k - \widetilde{X\alpha}_{k,i}) \times (X\beta_k - \widetilde{X\beta}_k)$

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

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 polynomial fit (estimated series); and the equivalent step of the $DCCA$ is, in the same terms, compared to the covariance between the two series. The technique of fitting a curve, interpreted as a trend inside each box, and subtracting the value estimated by the trend from the actual value in the integrated series $(X_k - \widetilde{X}_k)$ from now on will be called **detrended values**(DV). In the DFA algorithm, the $f^2_{DFA}(n,i)$ function is the mean of the square of the DV, in $DCCA$ calculations, the $f^2_{DCCA}(n,i)$ function evaluates the mean of the product of the DV of the two series in each box.

Step five of the DFA calculates the square root of the mean of the values calculates in the previous step for each box, in the $DCCA$, the mean of the values evaluated for 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 (n).

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.

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

The DMC^2_x 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 $\{X_1\}$, $\{X_2\}$, $\{X_3\}$, $\ldots$, $\{X_j\}$ defined as independent variables. The coefficient is expressed mathematically as:

$$DMC^2_x \equiv \rho_{Y,X_i}(n)^T \times \rho^{-1}(n) \times \rho_{Y,X_i}(n) \tag{2}$$

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 cross-correlation between a series and itself. Second, the matrix is symmetric in relation to the main diagonal, as the ρ_{DCCA} is a commutative operation.

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

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.

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

As the DFA and the $DCCA$, ρ_{DCCA} and 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. manage multiple time series inputs;

5. operate the DMC_x^2 for any number of series;

6. create a customizable and modular set of tools;

7. facilitate package evolution and maintenance;

8. 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. The 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 data related applications.
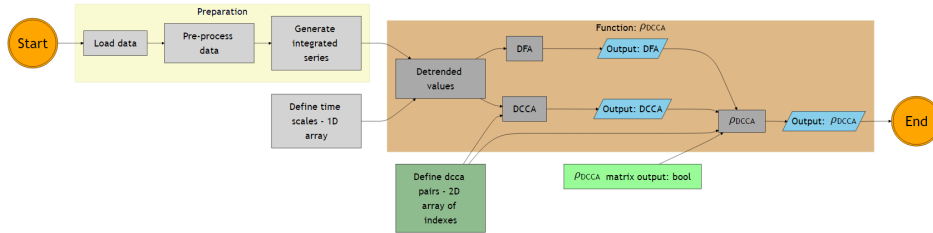

Figure 1: Calculating ρ_{DCCA} with **Zebende** package - Simplified flowchart

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.

The preparation steps are the same in both functions. First the data is loaded, and should be analyzed by the researchers. Based 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 a matrix with the columns as the series and the lines as time steps. Columns unwanted in the indented research should also be dropped for better performance of the
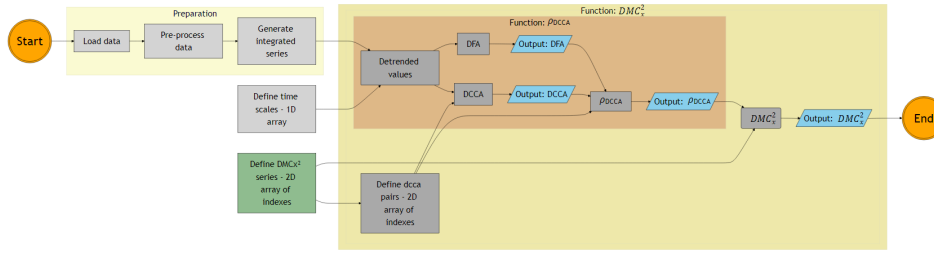
Figure 2: Calculating DMC_x^2 with **Zebende** package - Simplified flowchart

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. Any data manipulation Python package can export a table as a **Numpy** array. 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 methods (`p_dcca()` and `dmcx2()`) 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 most used one in different scenarios, or even embrace new proposals for the series integrating step.

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 `tws` 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 `tws` 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]]`

3.1. Implementation of the Detrended Cross-correlation Coefficient function

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 pure Python implementation of the ρ_{DCCA} function,

```
for n_index, n  in enumerate(tws): # for each time scale
  # temporaty 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 `tws` input, ensuring that step 6 of the DFA and $DCCA$ methods, presented in Section 2, is being carried out. In other words, the calculations will be applied, sequentially, to every single value in the `tws` array. Three temporary arrays are allocated and resized for each time scale. The first, `f2dfa_n`, is used to store the calculations of the step 4 of the DFA (f_{DFA}^2). The number of lines of this array correspond to the number of boxes in the current time scale ($N - n$, resized for each time scale n) and the columns is the number of series in the analysis (same size for every value of n). Second, the array `dcca_n` holds the values calculated in the step 4 of the $DCCA$ (to calculate $f_{DCCA}^2(n, i)$). The number of lines also correspond to the number of boxes(resized for each n) but the number of columns equals the number of pairs (rows) in the `DCCA_of` input (same size in each n). The `detrended_mat` array has a number of lines equal to the number of points in a time scale box ($n + 1$, resized for each n) and column count also equal to the number of time series (same size in each n). The first two temporary arrays will store data for all the boxes in the time scale, the last one will be used in each box and will have the values replaced in the next one, until the last box of the time scale. Than all the arrays will be cleared and recreated with shapes calculated with the new value of n.

After the allocation of the temporary arrays, the second `for` loop operates in every box for a certain time scale n. In each box, the `detrended_series()` function execute a one degree polynomial fit, subtract the values of the series with the value given by the interpolated curve(DV) and stores this values in the `detrended_mat`. After that, the **mean of the square of each DV in the box** ($f_{DFA}^2(n, i)$) is calculated in the current box for every time series and stored in the `f2dfa_n` array in the line associated with the current box, and the column related to each time series.

The next nested `for` loop operates over the `DCCA_of` array. For each line (pair) in the `DCCA_of` array, the $f_{DCCA}^2(n, i)$ is evaluated by multiplying the corresponding DV from the two series boxes in the current pair, getting the mean, and stores it in the `dcca_n` temporary array.

After all the pairs are calculated, the algorithm goes back to the each box `for` loop and head to the next box for the current time scale. when all the all the boxes are calculated, the final results for the DFA, $DCCA$ and ρ_{DCCA} are evaluated and saved in the output vectors `F_DFA_arr`, `DCCA_arr` and `P_DCCA_arr` respectively.

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()`).

3.2. Implementation of the Detrended Multiple Cross-correlation Coefficient function

The `dmcx2()` function runs the `p_dcca()` with `P_DCCA_output_matrix` set to `True` in the background. 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. For each row, the first elements is the index of the series used as the dependent variable, the others, the index of 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 dependent variable in relation with all the others. The `'all-full'` option is conducted by calling `dmc_of_all_as_y()`, also available as a utility function.

With a given `dmcx2_of` an array with all the necessary pairs for the $DCCA$ and ρ_{DCCA} is automatically generated and used for the background `p_dcca()` function to calculate da matrix as in Eq. 3. The `dcca_of_from_dmcx2_of()`, also can be used as an utility function, receives the `dmcx2_of` as input and returns the `dcca_of` array. With the matrix ρ_{DCCA} assembled, two internal functions, that can also be used as utility functions, calculates the DMc_x^2 for all the lines in the `dmcx2_of` matrix, for all the time scales. The first function, `dmcx2_from_p_dcca_matrix()`, that receives the ρ_{DCCA} and the `dmcx2_of` array, is presented in the code below.

```
def dmcx2_from_p_dcca_matrix(P_DCCA_arr: NDArray[np.float64],
    dmcx2_of: NDArray[np.float64]) -> NDArray[np.float64]:
  # DMCx2 output matrix
  DMCx2_arr = np.full(shape=(P_DCCA_arr.shape[2], dmcx2_of.shape[0]),
  fill_value=np.nan, dtype=P_DCCA_arr.dtype)

  for n_index in range(P_DCCA_arr.shape[2]):
      P_DCCA_arr_2D = P_DCCA_arr[:, :, n_index]
      for j,  dmcx2_of_1D in enumerate(dmcx2_of):
          DMCx2_arr[n_index, j] = dmcx2_from_p_dcca_matrix_2d(P_DCCA_arr_2D,
                                                      dmcx2_of_1D)
  return DMCx2_arr
```

The `dmcx2_from_p_dcca_matrix()` function executes a for loop over the ρ_{DCCA} that separates the this 3D matrix in to the 2D matrices for each time scale. Nested in this loop, another `for` extracts each line of the `dmcx2_of` and passes, the extracted matrix and the line vector to the `dmcx2_from_p_dcca_matrix_2d()`, displayed here.

```
def dmcx2_from_p_dcca_matrix_2d(P_DCCA_arr_2D: NDArray[np.float64],
            dmcx2_of_1D: NDArray[np.float64]) -> NDArray[np.float64]:
  y_index = dmcx2_of_1D[0:1]
  x_indexes = dmcx2_of_1D[1:]

  mat_x = P_DCCA_arr_2D[np.ix_(x_indexes, x_indexes)]
  vec_y = P_DCCA_arr_2D[np.ix_(x_indexes, y_index)]

  return vec_y.T @ np.linalg.inv(mat_x) @ vec_y
```

The `dmcx2_from_p_dcca_matrix_2d()` function uses **Numpy** methods to prepare the data to apply Eq. 2. The list of indexes is divided in `y_index`, holding an one item array whit the index of the dependent variable time series, and `x_indexes`, containing the indexes of the independent ones. The **Numpy** `np.ix_()`, although it's not a very known function of the library constructs index arrays that will use the cross product from a series of 1D arrays as inputs. It's a very convenient way to extract a sub matrix and a vector from the ρ_{DCCA} matrix. The matrix, as assembled by the `p_dcca()` function, will always need to have extractions of a 2D matrix and a vector, as we can see in Eq. 2. The code below is an example of the extraction process.

```
import numpy as np
arr = np.array([[1,2,3,4],
                [2,1,5,6],
                [3,5,1,7],
                [4,6,7,1]])
print(arr)

[[1 2 3 4]
 [2 1 5 6]
 [3 5 1 7]
 [4 6 7 1]]
```

An illustrative `arr` matrix is defined as 4×4 with all ones in the main diagonal and symmetric integer values in the other cells. Since the ρ_{DCCA} ranges from -1 to 1, those values should be interpreted as place holders.

```
sub_mat_index = np.ix_([1,2,3], [1,2,3])
print("index combination:\n", sub_mat_index)
print("extracted matrix:\n",arr[sub_mat_index])

index combination:
(array([[1],
       [2],
       [3]]), array([[1, 2, 3]]))
extracted matrix:
[[1 5 6]
 [5 1 7]
 [6 7 1]]
```

Above, a sub matrix, holding the positions 1 to 3 is extracted using `np.ix_()` function, and below, the extraction of the vector, first as a line and then as a column vector.

```
sub_mat_vec = np.ix_([0], [1,2,3])
print("line vector:\n", arr[sub_mat_vec])
sub_mat_vec = np.ix_([1,2,3],[0])
print("column vector:\n", arr[sub_mat_vec])


line vector:
[[2 3 4]]
column vector:
[[2]
[3]
[4]]
```

In the **Zebende** package, the vector is extracted as a column for better coherence with the DMC_x^2 theory. The method also works for dependent variable different of index 0. The resulting matrix will preserve the diagonal as 1, the symmetry regarding the main diagonal and the order of elements in the column vector respected. the code below extract the index 1 series as the dependent and the others as independent.

```
sub_mat_index = np.ix_([0,2,3], [0,2,3])
print("index combination:\n", sub_mat_index)
print("extracted matrix:\n",arr[sub_mat_index])
sub_mat_vec = np.ix_([0,2,3],[1])
print("column vector:\n", arr[sub_mat_vec])


index combination:
(array([[0],
      [2],
      [3]]), array([[0, 2, 3]]))
extracted matrix:
[[1 3 4]
[3 1 7]
[4 7 1]]
column vector:
[[2]
[5]
[6]]
```

The idea of separating the calculations of the $\rho DCCA$ calculations in three distinct functions aims to different workflows. The `dmcx2()` function calculates and outputs all the prerequisites, as the DFA, $DCCA$, ρ_{DCCA} along with the DMC_x^2. This is the most practical way of getting all this calculations conducted. But the task can also be divided in two: first use the `p_dcca()` function to generate DFA, $DCCA$, ρ_{DCCA} outputs, analyze the outputs and then get the DMC_x^2 using `dmcx2_from_p_dcca_matrix()`.

Function `dmcx2_from_p_dcca_matrix_2d()` can be used to more customizable applications. Imagine a use case where only the ρ_{DCCA} anti-correlation, inside a certain range, in relation to the dependent variable, with the `dmcx2_0f` set to `all-full`. From previous ρ_{DCCA} studies is known that this coefficient can vary from positive to negative and vice versa in different time scales. This implies that the ρ_{DCCA} matrix should be analyzed in every time scale from every line of the `dmcx2_of`, also implies that the `dmcx2_of` may not be a matrix in the since that the rows may have different. The **Numpy** ND Array could not hold that. Many different workarounds could be proposed for that situation. In the implementation of this package, function `dmcx2_from_p_dcca_matrix_2d()` allow the user to make a custom code that extracts the ρ_{DCCA} matrix for the current time scale and extracts the elements that fit the rules from each `dmcx2_of` row.

3.3. Zig implementation

The **Python** implementation successfully reflects the implementation goals presented in Sec. 3 but the performance could benefit with the integration of a low-level language. **Zig** was chosen to enhance the algorithms performance. It's a low-level language that gain popularity in recent years and provides performance similar to **C** and **Fortran** in some scenarios (Kacs, Lee, Zarins, and Brown 2024).

The interest for using **ZIg** for this project also relies on the cross-compiling capabilities of the **Zig** compiler. For a small research group maintaining a package could be challenging and the ability to to compile all releases for all platforms in a single machine is a great advantage. The **Zig** compiler can generate binaries for Windows, Linux and MacOS from a single machine. The **Zig** compiler is also very fast, and the language is very easy to learn, with a syntax that is very similar to **C**.

The implementation focus on writing technics ρ_{DCCA} function in **Zig** exposed as a **C** Application Binary Interface (ABI), called by **Python** using the **ctypes** package. The ρ_{DCCA} function is the most computational expensive part of the package, and the performance gain was expected to be significant.

The output arrays are allocated in the **Python** side and passed as pointers to the **Zig** function together with the series matrix, `tws` and `DCCA_of` arrays. The **Zig** function receives the pointers and the size of the arrays as inputs, and the results are stored in the same memory space. Before passing to **Zig** must be assured that the arrays are contiguous, using the `np.ascontiguousarray()` function. The boolean parameter `P_dcca_matrix_output` is also passed to the **Zig** function, to determine if the output of the ρ_{DCCA} should be a table or a matrix.

The **Zig** implementation follows the same steps as the **Python** one, but repspecting languages differences. In **Python**, using **Numpy**, all the time series calculations occur in the same line of code, using the **Numpy** broadcasting capabilities. In **Zig**, the calculations are made in a nested `for` loop.

3.4. Algorithm optimization

Although the **Zig** implementation presents a significant performance gain, the algorithm still can be optimized. The strategy is to focus on avoiding repeated calculations in the process. The code below shows the calculations of the polynomial fit before the optimization.

```
pub fn lin_ls_fit(win: []f64, time: []f64) [2]f64 {
    var x_sum: f64 = 0;
    var y_sum: f64 = 0;
    var xy_sum: f64 = 0;
    var x2_sum: f64 = 0;
    for (win, time) |w, t| {
        x_sum += t;
        y_sum += w;
        xy_sum += t * w;
        x2_sum += pow(f64, t, 2);
    }
    const n: f64 = @as(f64, @floatFromInt(time.len));
    //slope
    const slope: f64 = (((n * xy_sum) - (x_sum * y_sum)) /
        ((n * x2_sum) - (pow(f64, x_sum, 2))));
    //inter
    const inter: f64 = ((y_sum - (slope * x_sum)) /
        (n));
    //result
    return [_]f64{ slope, inter };
}
```

The code above calculates the slope and the intercept of a linear least squares fit. The function runs on every box for every time scale. In the optimized version, for consecutive boxes, the value of the sums from the previous box is used to calculate the next one without a loop for every item in the box. This technic proved to be very efficient in increasing performance. the expressions below show the effectiveness of the optimized algorithm.

$$\forall \, 1 < i \le (N - n), \; \sum_{k=i}^{i+n} T_k = \left(\sum_{j=i-1}^{(i+n)-1} T_j \right) \; - \; T_{i-1} \; + \; T_{i+n} \tag{5}$$

$$\forall \, 1 < i \le (N - n), \; \sum_{k=i}^{i+n} T_k^2 = \left(\sum_{j=i-1}^{(i+n)-1} T_j^2 \right) \; - \; T_{i-1}^2 \; + \; T_{i+n}^2 \tag{6}$$

$$\forall \, 1 < i \le (N - n), \; \sum_{k=i}^{i+n} S_k = \left(\sum_{j=i-1}^{(i+n)-1} S_j \right) \; - \; S_{i-1} \; + \; S_{i+n} \tag{7}$$

$$\forall \, 1 < i \le (N - n), \; \sum_{k=i}^{i+n} (S_k \times T_k) = \left(\sum_{j=i-1}^{(i+n)-1} (S_j \times T_j) \right) - (S_{i-1} \times T_{i-1}) + (S_{i+n} \times T_{i+n}) \tag{8}$$

The expression in Eq. 7 states that the sum of the values in a box is equal to the sum of the previous box minus the first value of the previous box plus the last value of the current box. According to that, when a box is calculated, the sum and the first value are saved in temporary variables. To calculate the next sum, the previous sum is subtracted from

the temporary variable and the new value is added. The larger the box, the greater the performance gain.

Also, to optimize the calculations from one time scale to the next, the first sums for the first box are saved in a temporary variable. Considering the current time scale as tws_prev and the consecutive as $tws_current$. To calculate the current value, the algorithm takes the sums from the tws_prev and add the values with indexes that exceed the tws_prev size. The code was also rewritten with structs, to hold the temporary values and the functions for better readability. The code below shows the optimized version of polynomial fitting calculations.

```
fn shiftWindow( self: *MainOperator,
                n: usize, win_start: usize,
                F_DFA_ptr: *allowzero [*c]f64) void {
    self.time_window = self.time[win_start ..][ .. (n + 1)];
    // print("win_start {}\n", .{win_start});
    if (win_start != 0) {
        // updating sum_x
        self.current.sum_x = self.current.sum_x - self.left_x + self.time_window[n];
        // updating sum x^2
        self.current.sum_x2 = self.current.sum_x2 -
        pow(f64, self.left_x, 2) + pow(f64, self.time_window[n], 2);
        // updating y and y*x for every serie
        for (self.series, 0..) |serie, sr_index| {
            serie.current.sum_y = serie.current.sum_y -
            serie.left_y + serie.serie[win_start + n];

            serie.current.sum_xy = serie.current.sum_xy -
            (self.left_x * serie.left_y) +
            (self.time_window[n] * serie.serie[win_start + n]);
            serie.left_y = serie.serie[win_start];
            self.detrended(serie, win_start, &F_DFA_ptr.*[sr_index]);
        }
    } else { // win_start == 0
        self.current.window_len = n + 1;

        for (self.previous.window_len..self.current.window_len) |i| {
            self.previous.sum_x += self.time_window[i];
            self.previous.sum_x2 += pow(f64, self.time_window[i], 2);

            for (self.series) |serie| {
                serie.previous.sum_y += serie.serie[i];
                serie.previous.sum_xy += self.time_window[i] * serie.serie[i];
            }
        }

        // updating current sum values
        self.current.sum_x = self.previous.sum_x;
```

```
        self.current.sum_x2 = self.previous.sum_x2;

        for (self.series, 0..) |serie, sr_index| {
            serie.current.sum_xy = serie.previous.sum_xy;
            serie.current.sum_y = serie.previous.sum_y;

            serie.left_y = serie.serie[win_start];
            self.detrended(serie, win_start, &F_DFA_ptr.*[sr_index]);
        }
    }
    self.left_x = self.time_window[0];
}
```

In the code above, the `MainOperator` is a structure that holds the time series(y), the time window, the sums and the temporary values. The `shiftWindow()` function is a method of the `MainOperator` structure. The function receives the size of the box, the index of the first element of the box, and a pointer to the F_{DFA}^2 array. The function is called for every box in every time scale. The function is responsible for updating the sums of the time series, the sums of the DV and the DV matrix.

The three implementations, pure Python, Zlg and optimized Zig, where tested and the results are presented in the next section.

4. Results

5. Summary and discussion

References

Kacs D, Lee J, Zarins J, Brown N (2024). "Pragma driven shared memory parallelism in Zig by supporting OpenMP loop directives." In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 930–938. doi:10.1109/SCW63240.2024.00132.

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 Achitecture
Universität Innsbruck
Universitätsstr. 15
6020 Innsbruck, Austria
E-mail: `fernando.ribeiro@ufba.br`
and
Centro Universitário Senai-Cimatec