Efficiently discovering ternary lagged correlations

Rensen, Jolan J.R.

Award date:
2021

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Datamanagement

Efficiently discovering ternary lagged correlations

Master Thesis

Jolan J. R. Rensen

Supervisor:
dr. O. Papapetrou

Version: 1.1

Eindhoven, November 2021

Abstract

In this thesis, we investigate the challenge of efficiently finding ternary lagged correlations. The discovery of ternary correlations describes the search for highly correlated groups of three time series. Ternary 'lagged' correlations are similar, but they also allow for each time series to be shifted in time relative to the others in the group. We propose a solution that supports sliding windows and expands upon the Matrix Profile calculations and Mueen's Algorithm for Similarity Search [13] to allow for the detection of lagged ternary correlations. To make the calculations as efficient as possible, many optimizations are proposed. The solution comes in the form of an algorithm, which works efficiently for large window sizes, and an approximation technique, which can be applied in some specific cases, to further improve efficiency.

Preface

At the end of 2020, I decided to do a seminar project at the Databases group. While not a data analyst at heart, I did enjoy a course called Data Engineering, taught by my supervisor Odysseas Papapetrou. I felt like the part I enjoyed most in that course, the programming part, could definitely be applied to more projects in the same group. One of the suggested seminar projects included the concept of combining correlation in groups with lag between time series. This concept immediately sparked some ideas, so, the seminar project I chose to do directly led to the thesis you are reading now.

I would like to thank Odysseas Papapetrou a lot since he provided very useful guidance and tips. Without this, I probably would not have known where to start in the vast landscape that is Data Engineering. Aside from him, I would also like to thank my fellow students who were also writing their thesis and had Odysseas as their supervisor too. They were able to provide some key ideas that ended up shaping the thesis better.

Of course, I would also like to thank Odysseas, Renata Medeiros de Carvalho, and Nikolay Yakovets for being on the assessment committee.

Finally, I would like to thank my family and friends for just being there and listening to me rambling about the thesis itself or any of its sidetracks that I put myself on while working on it.

Jolan Rensen

Contents

Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1
1.1 Outline . . . 2

2 Preliminaries 3
2.1 Definitions . . . 3
2.2 Problem definition . . . 5
2.2.1 Main problem: Efficiently discovering ternary lagged correlations . . . 5
2.2.2 Sub-problem: How can we efficiently discover the strength of ternary lagged correlation within time series? . . . 6
2.2.3 Sub-problem: Can we skip window placements to find lagged correlations faster? . . . 6

3 Related work 7
3.1 Matrix Profile . . . 7
3.2 Others . . . 9

4 How can we efficiently discover the strength of ternary lagged correlation within time series? 11
4.1 Using STAMP for cross-correlation . . . 12
4.1.1 Cross comparison for two time series . . . 12
4.1.2 Pearson correlation instead of Euclidean distance . . . 12
4.2 Adding the third time series . . . 14
4.2.1 For-loop structure . . . 14
4.2.2 MASS-Pearson with aggregation . . . 17
4.3 Optimizations for STAMP 3D . . . 23
4.3.1 Vertical optimizations . . . 23
4.3.2 Horizontal optimizations . . . 24
4.3.3 Vertical- and horizontal optimizations overview . . . 26
4.3.4 Lag bound . . . 27
4.4 Group-by-Key Method . . . 27

5 Can we skip window placements to find lagged correlations faster? 29
5.1 Straight line skipping . . . 29
5.2 Motif skipping . . . 31
5.3 Marking which window positions to skip . . . 31
5.4 Straight line skipping algorithm . . . 33
5.5 Motif skipping algorithm . . . 35

5.6 Simplification . . . 37
5.7 Applying skipping to STAMP-3TS-Pearson . . . 38

6 Experiments **39**
6.1 Experimental setup . . . 39
6.1.1 Computer . . . 39
6.1.2 Dataset . . . 39
6.1.3 Implementation . . . 40
6.1.4 Summary of results . . . 40
6.2 First problem . . . 41
6.2.1 Optimized versus naive version . . . 41
6.2.2 Effectiveness of lag bound in STAMP-Pearson-3TS . . . 43
6.2.3 Sliding Means and -Standard Deviation with Aggregation . . . 44
6.3 Second problem . . . 45
6.3.1 Straight line skipping per window size . . . 46
6.3.2 Motif skipping per window size . . . 46
6.3.3 Straight line skipping with simplification . . . 48
6.3.4 Motif skipping with simplification . . . 49
6.3.5 Effectiveness of window skipping in STAMP-Pearson-3TS . . . 51

7 Conclusions **55**
7.1 Conclusion . . . 55
7.2 Future work . . . 56
7.2.1 Regarding first sub-problem . . . 56
7.2.2 Regarding second sub-problem . . . 57

Bibliography **59**

List of Figures

2.1 A small section of all window positions considered in a sliding window procedure for two time series. . . . 4
2.2 Highest correlation: 0.974118301573819 Red: Forex, EURSEK window: [2018-08-12 15:00:00 .. 2018-09-02 10:00:00] Orange: Forex, EURNOK window: [2018-08-04 19:00:00 .. 2018-08-25 14:00:00] Blue: Forex, EURHKD window: [2018-08-13 03:00:00 .. 2018-09-02 22:00:00] Correlation of agg(Red, Blue) and Orange 6

3.1 Example of self-pair-comparison[1] 7
3.2 Example of Matrix Profile[1] 7
3.3 Example of an iteration in STAMP supplying MASS with a time series T (green) and a query Q (blue). 8
3.4 3D motif comparison using mSTAMP [12] 9

4.1 Example of an iteration in STAMP with three time series (T_A, T_B, and T_C) supplying MASS with the whole time series T_A (green) and two queries Q_B (blue) from time series T_B and Q_C (red) from time series T_C. 15
4.2 Sliding dot product QT between time series T and query Q (window size of 2) . . 18
4.3 Sliding dot product QT_a between time series T, aggregated with a one-dimensional $\mathbf{Q_a}$, and query Q (window size of 2) 18
4.4 Example to show the derivation of Formula 4.6, so it can be used to get QT_a using QT from Figure 4.2. 19
4.5 $m = 2$, $l = 1$ 27

5.1 Windows T_{A_1} and T_{B_2} are highlighted in green. 29
5.2 Time series (blue) and accompanying Pearson Matrix Profile (red) calculated with a window size of 5. 31
5.3 Windows containing any removed (gray) data points are skipped (red example), while others are still calculated (green example). 32
5.4 Blue: Straight piece longer than $2m - 1$ where two values can be removed. Green: Window which when calculated covers the entire straight piece. 32
5.5 Blue: Straight piece longer than m where values can be removed. Green: Window which when calculated covers the entire straight piece. 32
5.6 Blue: Straight piece longer than m where values can be removed. Green: Window which when calculated covers the entire straight piece. 33
5.7 Blue: Original time series. Red: Ramer-Douglas-Peucker simplified time series. . . 37

6.1 Each data point is the time of the correlation calculation of a single group of 3 time series. 42
6.2 Each data point is the time of the correlation calculation of a single group of 3 time series. 42
6.3 Each data point is the time of the correlation calculation of a single group of 3 time series. 43

6.4 Each data point is the time of the correlation calculation of a single group of 3 time series. . . . 44
6.5 Each data point represents the time to calculate the sliding means and -stds for one time series. . . . 45
6.6 Each data point represents the time to calculate the sliding means and -stds for one time series. . . . 45
6.7 Each data point represents the number of window placements saved for a single time series. . . . 46
6.8 Each data point represents the number of window placements saved for a single time series. . . . 47
6.9 Each data point represents the average number of window placements saved for all time series. . . . 47
6.10 Each data point represents the average number of window placements saved for all time series. . . . 48
6.11 Each data point represents the average number of window placements saved for all time series. . . . 49
6.12 Each data point represents the average number of window placements saved for all time series. . . . 49
6.13 Each data point represents the average number of window placements saved for all time series. Matrix Profile precision: 0.9 50
6.14 Each data point represents the average number of window placements saved for all time series. Matrix Profile precision: 0.9 50
6.15 Each data point represents the running time of STAMP-Pearson-3TS with a group of 3 time series with the same number of saved window positions. Approximate exponential trend lines are visible. See Table 6.1 51
6.16 Each data point represents the running time of STAMP-Pearson-3TS with a group of 3 time series. . . . 51
6.17 Each data point is the time of the correlation calculation of a single group of 3 time series. . . . 52
6.18 Each data point is the time of the correlation calculation of a single group of 3 time series. . . . 53

List of Tables

4.1 Overview of the structure of the algorithms in this chapter. 12
4.2 2D matrix example 28

6.1 Amount of data points per window size, first dataset 51

Chapter 1

Introduction

There already exists a lot of research in detecting (cross-)correlations and similarities between pairs of time series [13, 17, 16]. One way this research is extended is by introducing correlation in groups, also called multiple-, or multivariate correlation [7]. This has applications in many fields, from biologists finding links between the color, mineral composition, and botanical order of honey [4], to climate scientists finding relations between the sea level pressure at three different points on the globe [11]. Another way pair-wise correlation is extended is by introducing lag. It is useful since correlations between two time series might not always exist unless they are shifted in time relative to each other. This method is also widely applicable, from finding strong connections between the number of people reported to believe in conspiracy theories during a COVID-19 wave and the number of people reporting to adhere to social distancing guidelines the following wave [2], to finding time-lagged correlation between prediction methods regarding the growth of sugarcane [5].

While those two techniques are useful in their own rights, for this thesis we imagined it to be beneficial to combine the two. In more concrete terms, this combination could mean finding correlations in the stock market where a spike in the value of two stocks today could influence another stock a month later. For example, it could detect correlations between spending money on advertising and finding a spike in sales a week later. Alternatively, it could detect a high correlation between the value of airplane tickets in March and the value of sunscreen and inflatables in July. Just like calculating 'normal' correlations between three or more variables is trivially harder than calculating correlations between just two, calculating 'lagged' correlations with groups of time series is also harder than with pairs. Naive solutions can easily become storage- and computationally heavy. Compared to 'normal' correlation detection in groups, lag essentially adds another dimension to the problem.

An option for comparing two time series is by having a sliding window [13, 6] go over both time series and then consider all possible placements of both windows to find the highest correlating subsequences. If m is the window size and n is the number of items in each time series, this method already has $(n - m + 1)^2$ options to calculate the best correlation per pair of time series. Doing the exact same thing for groups of x time series would thus give $(n - m + 1)^x$ options to consider per group. As can be seen, this does not scale well, so a more efficient solution is needed (for instance, by leaving out options that would not give sensible solutions) and, to make this process viable for data analysis, the solution needs to be as efficient as possible as well. Because time series are rarely random data and often contain certain patterns, there might be many speed gains to be found regarding this aspect.

In this thesis, we propose a new way of finding lagged correlations in groups of three to make it a more viable data analysis solution. Our investigation is split up into two separate sub-problems, which can be combined to improve the speed and efficiency of analyzing data this way. See the Problem definition (Section 2.2) for these sub-problems.

1.1 Outline

This thesis is split up as follows: First, we explain a couple of definitions and we lay out the problem statement in Chapter 2. Next, we look at related work in Chapter 3 and explain how it can be applied to the research. Then, in Chapter 4 and Chapter 5 the research sub-problems are extensively explained and a solution is proposed for discovering ternary lagged correlations. Afterward, the proposed solutions are tested using real-life data using experiments in Chapter 6 before providing a conclusion and future work in the final chapter, Chapter 7.

Chapter 2

Preliminaries

Before getting into detail, in this chapter, we would first like to go over some re-occurring definitions and notations in Section 2.1. We also explain the problem definition of discovering ternary lagged correlations and how this is split up into two research sub-problems in Section 2.2.

2.1 Definitions

Time series Often denoted as T of size n. It represents a string of values defined by the change of a value over time. It can be seen as a list, array, or vector and might be described in this thesis as any of those interchangeably.

Query Also called a 'subsequence'. Often denoted as Q of size m (window size). It represents a smaller subsection of a time series. It is often gained by taking the values within a window on the time series. It can also be seen as a list, array, or vector. A collection of multiple queries is denoted as $\mathbf{Q}$.

Pearson Correlation Often denoted by 'ρ', is a very common correlation measure. It can be used to measure how much linear dependence there exists between two series of values. The measure results in $+1.0$ when there exists an exact linear correlation between two series, while -1.0 denotes an exact but inverse linear correlation. The formula for Pearson correlation can be found below at Equation 2.1, where μ and σ represent the mean and standard deviation respectively. Pearson correlation also has a strong connection to Z-normalized Euclidean distance. For more information about this, see Section 4.1.2.

$$\rho(X, Y) = \frac{\sum (X - \mu_X)(Y - \mu_Y)}{\sigma_X \sigma_Y} \tag{2.1}$$

Sliding window This is the technique of sliding an m sized window along a time series by moving it from start to end one step at a time. Each window position in the time series can be called a 'query'. A sliding window technique over more than one time series is also possible. In this case, each time series has its own window and the windows move in such a way that all the possible combinations of window placements in the time series are visited exactly once. See Figure 2.1 for an example.

Lag The lag between two windows is simply the difference in time (or distance) between their starting positions.

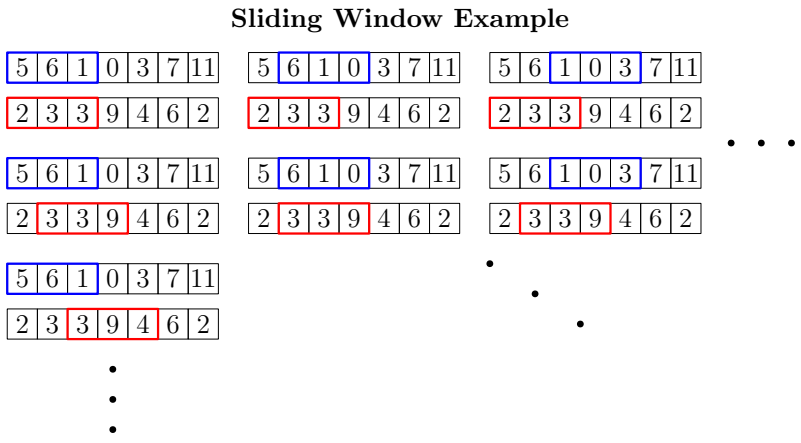

Figure 2.1: A small section of all window positions considered in a sliding window procedure for two time series.

Lag bound It is also called 'enveloping' and is often denoted by the value l. It can be used to limit the window placements across multiple time series to within a distance of l of each other. To clarify: When lag bound is implemented, then for any window, all the other windows are within l distance of it. More about lag bound and how it is applied can be found in Section 4.3.4.

Multivariate cross-correlation Correlation is usually measured in pairs, but it can also be calculated for small groups. Almost all of the correlations discussed in the thesis are Pearson correlations in groups of three. To get sensible results for three time series (because Pearson correlation normally only works between two time series) two of the three time series need to be aggregated together and then compared to the remaining one. The aggregation method, by default, is the mean. The best result of the three ways to aggregate the three time series is reported together with correlation value of the three time series. This is described in Section 4.2.1 as well.

Z-normalization This is a normalization technique for series of values to make it easier to compare and find trends between multiple series without scaling issues. For the values in a time series T it is calculated like Equation 2.2, where μ_T and σ_T represent the mean and standard deviation of T respectively.

$$T[i] = \frac{T[i] - \mu_T}{\sigma_T} \tag{2.2}$$

Whenever we talk about normalized Euclidean distance regarding the Matrix Profile, Z-normalization is what is inferred.

Motif A motif, in the Matrix Profile context, is a repeating pattern in a time series. It can consist of two or more instances. STAMP [13] can be used to create a Matrix Profile list from which motifs can relatively easily be discovered. See Section 3.1 for more info on motifs and the Matrix Profile in general.

Notation There are a couple of notes to give on the notation in the algorithms and across the rest of the thesis:

- Values are always denoted by a lowercase letter, like n. (Time) series, lists, or arrays of values are denoted by a capital letter, like T, and are used interchangeably, since they behave the same. Higher-dimensional structures, such as lists of lists or two-dimensional arrays are shown as a bold uppercase letter, such as $\mathbf{Q}$.

- A ".." range is considered inclusive. For instance "1..3" would refer to the range containing the numbers "1, 2, 3".
- The $X[i..j]$ notation represents a slice taken from a series of values X from index i to index j (inclusive). So essentially, $X[i..j] = X[i, i+1, ..., j-1, j]$.
- For all series, lists, arrays, and other objects that can be accessed using indices, indices start at 0.
- "$\cdot$" is considered the dot product.
- Operations on series, lists, and arrays are considered element-wise. For instance: $A + B$, where both A and B are similarly sized, one-dimensional arrays, would result in a one-dimensional array where each element from A is added up with an element from B at the same index. This means that AB is considered the pair-wise product of A and B.
- Building onto that, simple operations on series, lists, and arrays are also considered element-wise. For example, $A + 1$ would result in a new array similar to A but with each element being 1 larger.
- "agg(A, B)" is considered an aggregation by element-wise average. Essentially this can thus be rewritten as $\frac{A+B}{2}$ using the notations laid out before.
- $\sum X$ is used to represent the sum of each element in series of values X. In short: $\sum X = \sum_{x \in X} x$.

  This notation can also work for a collection of arrays or two dimensional lists like $\mathbf{Y}$: $\sum \mathbf{Y} = \sum_{Y \in \mathbf{Y}} Y$. The result of this would be a one-dimensional array.
- Some arguments of algorithms have a blue part. For example: "Algorithm(1, 2, $A = 3$, $B = 5$)". These arguments are named arguments. We use this notation for clarification; to show how the argument given is used in the called algorithm. In most cases, this means that the blue value does not have to be calculated anymore inside the called algorithm as it has been calculated already.

2.2 Problem definition

In this section, we give the problem statement of the thesis, as well as the two sub-problems which we use to tackle it.

2.2.1 Main problem: Efficiently discovering ternary lagged correlations

The goal of the thesis is to find an efficient approach to discover ternary lagged correlations. In more detail: We want to find an algorithm that can give the window positions inside each of three time series and the aggregation manner which will give the best (either positive or negative) Pearson correlation value of all window positions in as little time as possible. Using this algorithm, we can then find the best combinations of three time series in the data set to report to a data analyst.

An example result that the algorithm should be able to give can be seen in Figure 2.2. In the figure, the window positions that give the highest correlation are highlighted and have a small amount of lag relative to each other. The data comes from the second dataset described in Section 6.1.2.

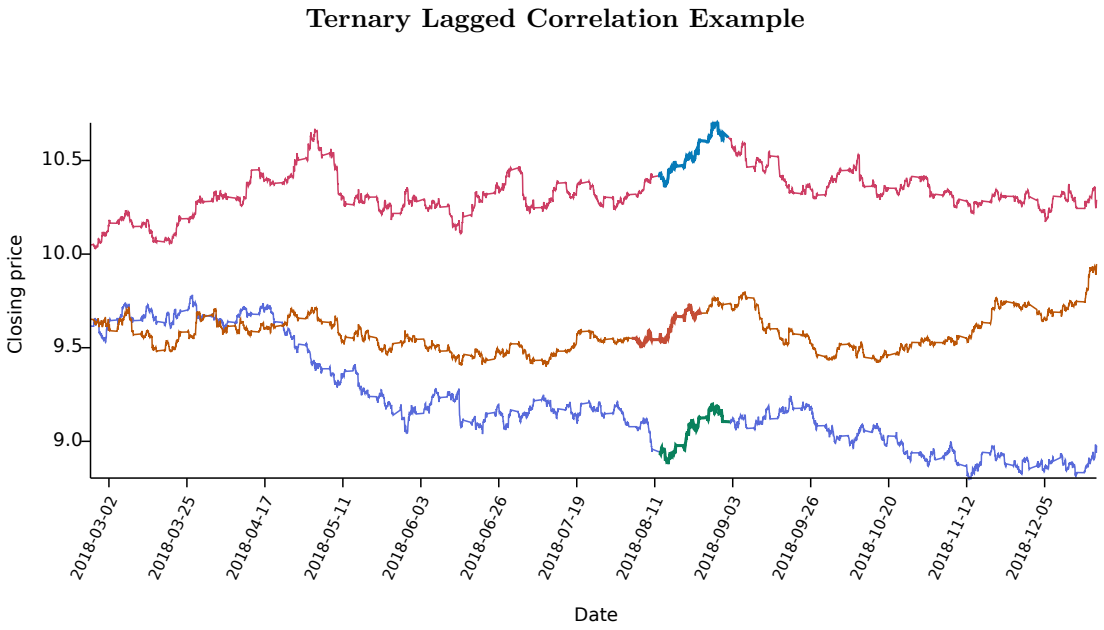

Figure 2.2: Highest correlation: 0.974118301573819

Red: Forex, EURSEK
window: [2018-08-12 15:00:00 .. 2018-09-02 10:00:00]

Orange: Forex, EURNOK
window: [2018-08-04 19:00:00 .. 2018-08-25 14:00:00]

Blue: Forex, EURHKD
window: [2018-08-13 03:00:00 .. 2018-09-02 22:00:00]

Correlation of agg(Red, Blue) and Orange

2.2.2 Sub-problem: How can we efficiently discover the strength of ternary lagged correlation within time series?

The first sub-problem aims to find an efficient approach to discover the best placements of a window in each time series and the best aggregation method to maximize the Pearson correlation for a group of three time series. This approach materializes in the form of an algorithm that can efficiently go over each possible window placement and then report the best one in terms of correlation. Aside from this, we also want to be able to find the best combinations of three time series from a data set of many. A small technique for this is also proposed. See Chapter 4.

2.2.3 Sub-problem: Can we skip window placements to find lagged correlations faster?

The second sub-problem tries to find an approach to find the best window placement while skipping as many window positions as possible. This approach can encompass some approximation and therefore differs from the first sub-problem. However, it might be able to make the approach of the first sub-problem work even better. See Chapter 5.

Chapter 3

Related work

In this chapter, we describe all the work that is related to this thesis and helped form it to how it currently stands. We first go over the Matrix Profile papers (Section 3.1) before touching on some others as well (Section 3.2).

3.1 Matrix Profile

Similar research regarding comparing time series is done by the collection of papers regarding the Matrix Profile. The first paper in this series [13] describes how to detect motifs (or similarities) and discords (anomalies) within a time series or between two different ones. The Matrix Profile (MP) described can be seen as a sparse distance-matrix consisting of the Euclidean distances between all possible pairs of n-sized subsequences of both time series. An example of the formation of one row in the matrix (based on the examples from the STUMPY website[1]) can be seen below in Figure 3.1.

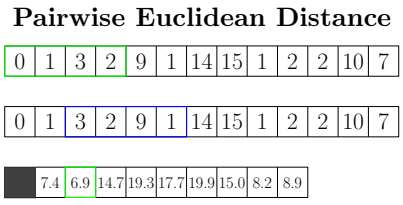

Figure 3.1: Example of self-pair-comparison[1]

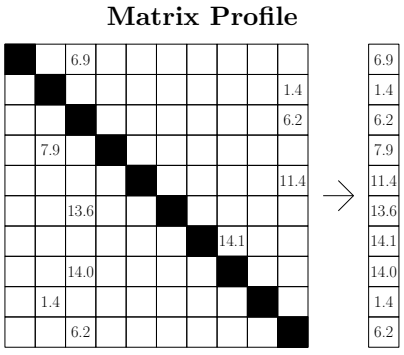

Figure 3.2: Example of Matrix Profile[1]

The bottom-most row in the figure is formed by sliding the blue (window) box along the series and calculating the Euclidean distance between the numbers in the green box and the blue box. Of this row, only the lowest value will be saved. For the Matrix Profile, this is currently 6.9. Another noteworthy thing is the black box in the first spot. This results from comparing a time series with itself and is left out of the row because the result would be trivial (namely 0). After forming the full matrix, it looks like Figure 3.2. The one-dimensional list can be seen as the result of calculating the Matrix Profile using time series T, or $MP(T, T)$. As can be seen, storage for a Matrix Profile is only $O(n)$, as only the list on the right is saved in storage together with an accompanying list of indices. These indices can be used to find where, in the time series, this smallest distance was found.

[1] https://stumpy.readthedocs.io/en/latest/Tutorial_The_Matrix_Profile.html

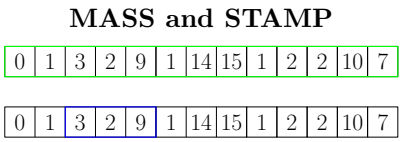

Figure 3.3: Example of an iteration in STAMP supplying MASS with a time series T (green) and a query Q (blue).

Creating a Matrix Profile using the STAMP algorithm [13] (which runs in $O(n^2 \log n)$) is roughly done by calculating the Euclidean distance between all the pairs. In practice, this is optimized with Mueen's Algorithm for Similarity Search (MASS) using Fast Fourier Transform- and sliding dot product calculations, described in the paper [13]. MASS essentially calculates all distances between one window position in one time series with all the window positions in the other time series very efficiently. When this is put in a for-each loop of all window positions, STAMP is the result. A visual example of an iteration in STAMP can be seen in Figure 3.3.

While the MP is useful for similarity- and anomaly detection, it needs to be tweaked to work for correlation calculation. Pearson correlation, for example, yields a value between -1 and 1, where 1 means a high positive correlation and -1 means a high negative correlation. Conversely, the Euclidean distance between two sequences is an unbounded value larger than 0, where it being closer to 0 means the sequences are more alike. Luckily, since the MP uses Z-normalized (see Formula 2.2) Euclidean distance, it is straightforward to convert MASS to return Pearson correlation instead of Euclidean distance. This conversion method is described in Section 4.1.2.

All in all, the Matrix Profile promises high efficiency and low storage usage for pairwise time series comparisons, which could come in handy when calculating correlations between multiple time series as well.

Another benefit of adopting the Matrix Profile is that there are many optimizations done around it. In the second paper [17], a second algorithm is described, namely STOMP (which runs in $O(n^2)$). Where STAMP is an anytime algorithm [13], meaning the Matrix Profile can be constructed asynchronously per row, STOMP constructs the Matrix Profile in an ordered fashion. This implies that it can use data from the previous iteration (or -row) to speed up calculations. The algorithm does lose the ability to be used for approximations as now the entire Matrix Profile always needs to be calculated instead of just one single row.

Another optimization is described in the eleventh paper [16], in the form of the algorithms SCRIMP and SCRIMP++. This algorithm adopts the running time of STOMP ($O(n^2)$) while also being an anytime algorithm like STAMP. According to the paper, it converges much faster than both other algorithms, which would make it more useful in interactive scenarios. It does need to be noted that the optimizations in place specifically target motif discovery, which might not apply to correlation discovery directly.

The last relevant paper in the Matrix Profile collection is the sixth [12]. In the paper, the algorithm mSTAMP is described. The algorithm creates a multi-dimensional Matrix Profile and can detect auto-motifs (a repeating pattern discovered within a single time series) that happen simultaneously for multiple time series. An example of what this algorithm can detect is shown in Figure 3.4.

While the idea of a multi-dimensional Matrix Profile can definitely be used for correlation detection, the way mSTAMP detects motifs at the same time across different might not be that useful. This is because for most correlations, the interest lies more in comparing a time series directly with another (so cross-correlation) instead of with itself (auto-correlation).

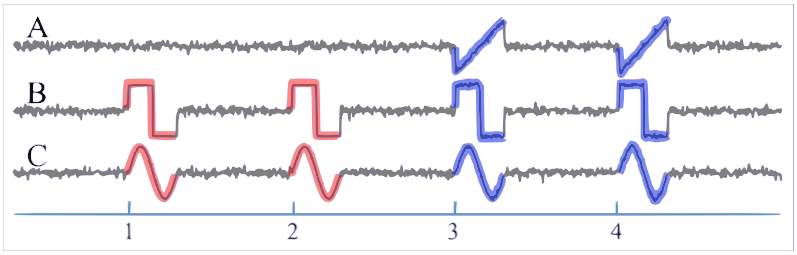

Figure 3.4: 3D motif comparison using mSTAMP [12]

3.2 Others

The thesis is mainly built around the Matrix Profile papers, combining some very common correlation concepts like Pearson correlation with aggregation of time series with the Matrix Profile algorithms. There exist other papers that use similar concepts, but the way they apply those concepts differs from this thesis. For instance, there are papers that consider lagged correlations using a sliding window approach, but only in pairs or with a correlation measure other than Pearson [6]. While cross-correlation measures like DCCA can provide useful information about the relation between time series [6, 8, 15], Pearson correlation is still one of the most used correlation measures. Other papers do consider finding correlations in groups of series but focus on a different form of lag, such as lag found by dynamic time warping [1], using different time scales [14], or they do not consider lag at all, instead focusing just on streaming and static data [7].

Chapter 4

How can we efficiently discover the strength of ternary lagged correlation within time series?

To find the group of three time series with the best lagged correlation, we first need to check all window positions in each possible group and find the correlation there. The number of possible groups is defined by $\binom{n}{3} = \frac{n!}{3!(n-3)!}$, where n is number of time series in the data set. Since this number gets larger quickly, checking all the window positions inside each group should be as efficient as possible. To still provide valuable information while not delaying the data analysts too much, in this chapter we propose an efficient technique, based on the Matrix Profile calculations, that can give the best correlation and the starting positions of the window in each time series where it holds.

This technique is explained starting with the MASS and STAMP algorithms from the Matrix Profile papers, adapting them for ternary correlations and improving their performance. First, we explain how to calculate lagged cross-correlation (Section 4.1) using the previously mentioned algorithms as a base. Next, we lay out a relatively simple version of ternary lagged correlation using a straightforward but inefficient for-loop structure (Section 4.2). The structure contains a lot of algorithms for which an overview is provided in Table 4.1. Finally, we give a set of optimizations that are applied to the straightforward version to vastly increase the efficiency (Section 4.3).

The proposed algorithms can only be used for a single group of three time series. We suspect it to be even more interesting to find the best combinations of three time series in a dataset of many time series. To efficiently calculate and go over all possible groups in a time series dataset in a distributed fashion, we propose a small but helpful technique in Section 4.4.

Name and number	Description
STAMP-Pearson-3TS (Alg. 3)	Initial algorithm for finding the best ternary lagged correlation. Traverses time series T_C. Calls STAMP-Pearson-3TS-Sub.
STAMP-Pearson-3TS-Sub (Alg. 4)	Secondary algorithm for finding the best ternary lagged correlation. Traverses time series T_B. Calls MASS-Pearson-3TS.
MASS-Pearson-3TS (Alg. 5)	Third algorithm for finding the best ternary lagged correlation. Splits calculations for three aggregation methods. Calls MASS-Pearson and MASS-Pearson-Agg.

MASS-Pearson (Alg. 1)	Finds best correlation for a whole time series and a query of another. Calls ComputeSlidingMeanStd.	MASS-Pearson-Agg (Alg. 6)	Same as MASS-Pearson but with aggregation of a query from a third time series. Calls ComputeSliding-MeanStdWithAgg.
Compute-Sliding-MeanStd (Alg. 2)	Computes the sliding means and standard deviations of a given time series.	Compute-Sliding-MeanStd-WithAgg (Alg. 7)	Same as ComputeSlidingMeanStd, but with aggregation of a query from another time series for each window position.

Table 4.1: Overview of the structure of the algorithms in this chapter.

4.1 Using STAMP for cross-correlation

STAMP is designed for self-similarity using Euclidean distance. In this section, we explain that it can also be used for cross-comparison and that Euclidean distance can be mapped to Pearson correlation. This shows that using STAMP (and MASS in its for-loop) can be used as a base in the search of lagged cross-correlations.

4.1.1 Cross comparison for two time series

The STAMP (Scalable Time series Anytime Matrix Profile) algorithm is the first algorithm described to make the Matrix Profile (MP) by the paper [13]. For the MP, the algorithm computes a self-similarity join matrix profile and only takes a single time series as input. However, it can also compute a general similarity join by accepting two different time series. This means that STAMP and MASS can also be used to find similarities between (or cross) time series, which is what we are interested in. There is no actual modification for the algorithm necessary to achieve this.

4.1.2 Pearson correlation instead of Euclidean distance

MASS [13] produces a Euclidean distance profile between a time series T and a query Q, which is a piece of another-, or the same time series. This distance profile is essentially an array containing the distance of Q to each subsequence in T. While this is useful in its own right, we are interested in Pearson correlation, not distance. Luckily, since it is using Z-normalized Euclidean distance, this conversion is relatively simple.

A conversion method between Z-normalized Euclidean distance and Pearson correlation is described in paper [9] and it works by applying Formula 4.1 to convert it to the Pearson correlation measure:

$$corr(X,\ Y) = 1 - \frac{1}{2m} d^2(\hat{X},\ \hat{Y}) \quad [9]. \tag{4.1}$$

In the formula, X and Y represent two time series, $\hat{X}$ and $\hat{Y}$ represent the Z-normalized versions of the time series. m represents the window- / query size and $d()$ represents the distance function.

This conversion can directly be applied to the distance formula of MASS. The original is given as Formula 4.2:

$$D = \sqrt{2m\left(1 - \frac{\mathrm{QT} - m\mu_Q M_T}{m\sigma_Q \Sigma_T}\right)} \quad [13]. \tag{4.2}$$

Here, D is the distance profile. m is again the window- / query size. QT consists of the dot product between the query Q and all subsequences in time series T (see paper [13] for the algorithm). μ_Q and σ_Q represent the mean and standard deviation of query Q. Finally, M_T and Σ_T represent the means and standard deviations of all subsequences in time series T.

Modified to give the "Pearson correlation profile", it looks like Formula 4.3:

$$P = \frac{\mathrm{QT} - m\mu_Q M_T}{m\sigma_Q \Sigma_T}. \tag{4.3}$$

We introduce MASS-Pearson (Algorithm 1) as a modified MASS [13] algorithm, replacing the Euclidean distance with Pearson correlation and introducing a reducer function r to only report the value and index of the best window position in T instead of the entire array of results.

To calculate the best lagged cross-correlation in pairs (so between two time series), Algorithm 1 can be put in an algorithm similar to STAMP where it would be called for each query Q sliding across one of the two time series.

Algorithm 1: MASS-Pearson(T, Q, r)
Input: T: time series. Q: query of window size. r: MIN or MAX reducer function, returns the index and value of best element.
Output: Index of best position of the window in T. Value of the correlation between Q and that window.

$m \leftarrow |Q|$ // window size
$QT \leftarrow$ SlidingDotProducts(Q, T) // see [13]
$\mu_Q, \sigma_Q \leftarrow$ ComputeMeanStd(Q) // computes the mean and standard deviation of Q
M_T, $\Sigma_T \leftarrow$ ComputeSlidingMeanStd(T, m) // Algorithm 2
$P \leftarrow$ CalculatePearsonProfile(QT, μ_Q, σ_Q, M_T, Σ_T, m) // uses Formula 4.3
return $r(P)$

4.1.2.1 Regarding ComputeSlidingMeanStd

The ComputeSlidingMeanStd algorithm is mentioned in the Matrix Profile paper [13] but not explicitly given. It is, however, explained. Compared to calculating the standard deviation and mean for each subsection in time series T, the difference in speed is gained by keeping a cache of sums and squares. For the sums, as the window slides along the time series, at each iteration, we subtract the value that has just left the window and add the value that has just entered the window. For the squares, it is comparable, but then the values are squared. To see our version of the algorithm in full, refer to Algorithm 2. The algorithm starts by initializing the sums cache with the sum of the first window, and squares cache with the sum of squares (the dot product). Then, for each iteration, it uses those two caches to calculate the current mean and standard deviation and stores them in the M_T- and Σ_T array, ending the iteration by updating both caches. After running the algorithm, each element in the arrays can be described by the equations in Formulas 4.4 and 4.5.

$$M_T[i] = \frac{\sum T[i..i+m]}{m} \tag{4.4}$$

$$\Sigma_T[i] = \sqrt{\frac{\sum \left(T[i..i+m] - M_T[i]\right)^2}{m}} \tag{4.5}$$

A note on notation: $X[i..j]$ represents a slice taken from array X from index i to index j (inclusive). So essentially, $X[i..j] = X[i, i+1, ..., j-1, j]$. $\sum X$ is used to represent the sum of

each element in array X. In short: $\sum X = \sum_{x \in X} x$. For more details on the notation, refer back to the notation part in Section 2.1.

Algorithm 2: ComputeSlidingMeanStd(T, m)
Input: T: time series. m: window size.
Output: Sliding mean- and sliding standard deviation array.

$n \leftarrow |T|$
M_T, $\Sigma_T \leftarrow$ empty arrays of size $n - m + 1$
sums $\leftarrow \sum T[0..m-1]$ // initially the sum of first window
squares $\leftarrow T[0..m-1] \cdot T[0..m-1]$ // initially the sum of squares of first window (dot product)

for all $i \in (0..n-m)$ **do**
  $\mu \leftarrow sums/m$
  $M_T[i] \leftarrow \mu$
  $\Sigma_T[i] \leftarrow \sqrt{squares/m - \mu^2}$

  if $i < n - m$ **then**
    sums $\leftarrow sums - T[i] + T[i+m]$
    squares $\leftarrow squares - T[i]^2 + T[i+m]^2$
  end if
end for

return M_T, Σ_T

4.2 Adding the third time series

The goal of the thesis is to find lagged correlations in groups of three time series. While MASS-Pearson (and a Pearson-modified STAMP) can be used to find lagged correlations in pairs, we need to put in some changes to find correlations in larger groups. We want to keep MASS-Pearson as efficient as possible, using the fast SlidingDotProducts algorithm from the Matrix Profile paper [13]. Still, we also need to account for the aggregation that needs to occur for calculating Pearson correlation in groups.

In the following section (Section 4.2.1) we go over a straightforward for-loop structure similar to STAMP, but modified by us for three time series. We visit the algorithm top-down, meaning we start with three whole time series. We slide a window over one and we call the next algorithm for each position. This goes on until we have one whole time series and two queries of the others. At this stage (Algorithm 5), the aggregation, needed for Pearson, has to take place. While for one aggregation method, MASS-Pearson (Algorithm 1) suffices, for the other methods, further modification is necessary. These changes are proposed in Section 4.2.2, which describes how MASS-Pearson needs to be modified to make it work.

4.2.1 For-loop structure

First of all, we need to have a basic idea of how the algorithm should work. STAMP with two time series works like Figure 3.3, where a sliding window is moved along one of the two time series, and each iteration, the values inside the window are compared to all the window positions in the other time series using MASS(-Pearson). With three time series, the idea is to have two sliding windows over two time series, like in Figure 4.1, and for each iteration to calculate the Pearson correlations for all three possible aggregation methods.

This way, MASS-Pearson still gets one whole time series and one query to process. The way the aggregation works is explained in Section 4.2.2. Having two separate sliding windows is great for having a better overview of how the algorithm works, but it evades the speed benefits SlidingDotProducts and ComputeSlidingMeanStd could give. More on how these can still be

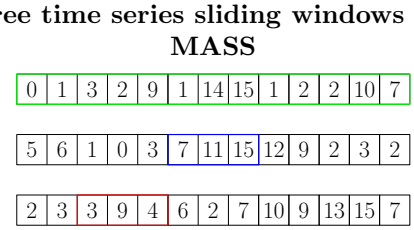

Figure 4.1: Example of an iteration in STAMP with three time series (T_A, T_B, and T_C) supplying MASS with the whole time series T_A (green) and two queries Q_B (blue) from time series T_B and Q_C (red) from time series T_C.

applied can be found in Section 4.3. For a complete overview of the structure of the algorithms in this chapter in relation to each other, see Table 4.1.

The basic structure of the algorithm consists of two loops around MASS-Pearson-3TS (Algorithm 5). The first loop occurs in STAMP-Pearson-3TS (Algorithm 3), where a window is slid over time series T_C and for each position, the query inside the window is given to STAMP-Pearson-3TS-Sub (Algorithm 4) together with T_A and T_B. This piece of the algorithm reports the best positions of the windows in all three time series, the aggregation method used, and the correlation value.

Algorithm 3: STAMP-Pearson-3TS(T_A, T_B, T_C, m, r)
Input: T_A, T_B, T_C: time series. m: window size. r: MIN or MAX reducer function, returns the indices, best aggregation method, and value of best window placements.
Output: Indices of best positions of the window in T_A, T_B, and T_C. Aggregation method used. Value of the best correlation found.

```
initialize bestCorrelation, bestAgg, bestIndices based on r
T_C ← AllSubsequences(T_C, m) // give all subsequences in time series of window size
for all i_C ∈ (0..|T_C| − 1) do
    indices, agg, corr ← STAMP-Pearson-3TS-Sub(T_A, T_B, T_C[i_C], m, r) // Algorithm 4
    if corr is better than bestCorrelation according to r then
        update bestCorrelation, bestAgg, bestIndices
    end if
end for
return bestCorrelation, bestAgg, bestIndices
```

A similar thing happens in STAMP-Pearson-3TS-Sub (Algorithm 4), but then a window is slid over time series T_B and both queries for T_B, and T_C and the time series T_A are then supplied to MASS-Pearson-3TS (Algorithm 5). This piece of the algorithm reports the best positions of the

windows in T_A and T_B given Q_C, the aggregation method used, and the correlation value.

Algorithm 4: STAMP-Pearson-3TS-Sub(T_A, T_B, Q_C, m, r)
Input: T_A, T_B: time series. Q_C: query of window size. m: window size. r: MIN or MAX reducer function, returns indices, best aggregation method, and value of best window placements.
Output: Indices of best positions of the window in T_A, and T_B. Aggregation method used. Value of the best correlation found.

```
initialize bestCorrelation, bestAgg, bestIndices based on r
T_B ← AllSubsequences(T_B, m) // give all subsequences in time series of window size
for all i_B ∈ (0..|T_B| − 1) do
  indices, agg, corr ← MASS-Pearson-3TS(T_A, T_B[i_B], Q_C, r) // Algorithm 5
  if corr better than bestCorrelation according to r then
    update bestCorrelation, bestAgg, bestAIndex, bestBIndex, bestCIndex
  end if
end for
return bestCorrelation, bestAgg, bestIndices
```

So far, the algorithm pieces are fairly straightforward, but MASS-Pearson-3TS (Algorithm 5) is a bit more interesting. This algorithm piece reports the best window position inside T_A given Q_B and Q_C, the best of three aggregation methods and, of course, the correlation value. MASS-Person(-Agg) (Algorithm 1, 6) needs to be run with all aggregation options for the three time series to get these values.

Since Pearson correlation can only be calculated in a meaningful way between two series, we need to aggregate the queries of two of the three time series, compare it to the other query in all three possible ways and then report the best one. So, to be more exact, the algorithm needs to report the best of $corr(agg(Q_A, Q_B), Q_C)$, $corr(Q_A, agg(Q_B, Q_C))$, and $corr(agg(Q_A, Q_C), Q_B)$. For aggregation, the element-wise-average method is used. This aggregation method also proves handy inside MASS-Pearson-Agg (Algorithm 6).

However, since we want to use MASS-Pearson, which takes a whole time series and (in our case) two smaller queries, the aggregation becomes more challenging. The second of the aggregation methods, which in MASS-Pearson-3TS (Algorithm 5) looks like T_A and $agg(Q_B, Q_C)$, works fine. Since Q_B and Q_C have the same length, they can be aggregated by average and, together with T_A, be supplied to MASS-Person (Algorithm 1).

Although, for the other two cases, the aggregation cannot be done at this stage and needs to be dealt with at a lower level, inside a further modified MASS-Pearson algorithm that can handle

these 'aggregation queries'. More on this in Section 4.2.2.

Algorithm 5: MASS-Pearson-3TS(T_A, Q_B, Q_C, r)
Input: T_A: time series. Q_B, Q_C: queries of window size. r: MIN or MAX reducer function, returns index, best aggregation method, and value of best element.
Output: Index of best position of the window in T. Aggregation method used. Value of the correlation between Q and that window.

// agg(T_A, Q_B) and Q_C
$P_A \leftarrow$MASS-Pearson-Agg(T_A, [Q_B], Q_C, r) // Algorithm 6

// T_A and agg(Q_B, Q_C)
$P_B \leftarrow$MASS-Pearson(T_A, $\frac{Q_B+Q_C}{2}$, r) // Algorithm 1

// agg(T_A, Q_C) and Q_B
$P_C \leftarrow$MASS-Pearson-Agg(T_A, [Q_C], Q_B, r) // Algorithm 6

// join the arrays elementwise taking the MIN or MAX of the 3 depending on r
// keeping track of which aggregation method the result belonged to
$P \leftarrow$ TakeElementwiseBest(P_A, P_B, P_C, r)
return $r(P)$

4.2.2 MASS-Pearson with aggregation

As described above, while aggregating two queries of the same length works fine, it is a bit more cumbersome when each subsection of a full time series needs to be aggregated with a query. For these cases, MASS-Pearson (Algorithm 1) needs to be modified into MASS-Pearson-Agg (Algorithm 6). This new version can take any number of aggregation queries $\mathbf{Q_a}$ which are aggregated with time series T for each of its subsequences before running the normal MASS-Pearson calculations. Allowing any number of aggregation queries means this algorithm can also work for MASS-Pearson algorithms with more than three time series in the future.

To make sure all queries from $\mathbf{Q_a}$ are applied everywhere the subsequences of T are accessed, MASS-Pearson (Algorithm 1) needs to change in two places: At the SlidingDotProducts- and at the ComputeSlidingMeanStd algorithm.

After these changes, the result can be found at Algorithm 6.

A note for notation: Some arguments of algorithms have a blue part. These arguments are named arguments. We use this notation to show how the argument given is used in the called algorithm if it is otherwise unclear. See the notation section, Section 2.1, for more information about notations.

Algorithm 6: MASS-Pearson-Agg(T, $\mathbf{Q_a}$, Q, r)
Input: T: time series. $\mathbf{Q_a}$: array of query arrays to aggregate T with. Q: query of window size. r: MIN or MAX reducer function, returns index and value of best element.
Output: Index of best position of the window in T. Value of the correlation between Q and that window.

$QT \leftarrow$ SlidingDotProducts(Q, T) // see [13]
$Q_{aE} \leftarrow$ ElementwiseProducts($\mathbf{Q_a}$)
$QT_a \leftarrow \frac{QT+(Q_{aE} \cdot Q)}{|\mathbf{Q_a}|+1}$
μ_Q, $\sigma_Q \leftarrow$ ComputeMeanStd(Q)
M'_T, $\Sigma'_T \leftarrow$ ComputeSlidingMeanStdWithAgg(T, $m = |Q|$, $\mathbf{Q_a}$) // Algorithm 7
$P \leftarrow$ CalculatePearsonProfile(QT_a, μ_Q, σ_Q, M'_T, Σ'_T, $m = |Q|$) // uses Formula 4.3
return $r(P)$

4.2.2.1 SlidingDotProducts with aggregation

The SlidingDotProducts [13] algorithm takes a time series T and a query Q and returns a new array QT where each element is a dot product between a subsection of T (of given window size and thus also $|Q|$) and Q. A general example of this can be seen in Figure 4.2. We want to modify it to make sure the result will be an array where each element is the dot product between Q and a subsection of T, aggregated with all aggregation queries in $\mathbf{Q_a}$. In Figure 4.3 an example can be seen where $\mathbf{Q_a}$ consists of just one aggregation query. As can be seen in the figure, each value from a subsection in T is first aggregated by average using $\mathbf{Q_a}$ before being used in the dot product with Q.

Original SlidingDotProducts behavior

T : | a | b | c | d | e |

Q : | A | B |

QT : | $aA + bB + ...$ | $bA + cB + ...$ |

| $cA + dB + ...$ | $dA + eB + ...$ |

Figure 4.2: Sliding dot product QT between time series T and query Q (window size of 2)

Aggregated SlidingDotProducts behavior

T : | a | b | c | d | e |

Q : | A | B | $\mathbf{Q_a}$: | α | β |

QT_a : | $A \cdot \frac{a+\alpha}{2} + B \cdot \frac{b+\beta}{2} + ...$ | $A \cdot \frac{a+\alpha}{2} + B \cdot \frac{b+\beta}{2} + ...$ |

| $A \cdot \frac{a+\alpha}{2} + B \cdot \frac{b+\beta}{2} + ...$ | $A \cdot \frac{a+\alpha}{2} + B \cdot \frac{b+\beta}{2} + ...$ |

Figure 4.3: Sliding dot product QT_a between time series T, aggregated with a one-dimensional $\mathbf{Q_a}$, and query Q (window size of 2)

It is clear what QT_a should look like and, luckily, it is also possible to get to this result without modifying SlidingDotProducts itself. QT can be converted to QT_a using Formula 4.6.

$$QT_a = \frac{QT + (Q_{aE} \cdot Q)}{|\mathbf{Q_a}| + 1} \tag{4.6}$$

In the Formula 4.6, the aggregated sliding dot product result QT_a is calculated using the 'normal' sliding dot product result QT, calculated by SlidingDotProducts [13]. The array of aggregation queries that need to be applied is $\mathbf{Q_a}$ and Q_{aE} is the result of taking the element-wise product of all the aggregation queries in $\mathbf{Q_a}$. To show the formula works, a small derivation is shown in Figure 4.4, this time, using a two-dimensional $\mathbf{Q_a}$.

Firstly, you can see time series T and query Q on the left-hand side of the figure. The window size is again 2. In this example, $\mathbf{Q_a}$ consists of two aggregation queries divided by a bold line. Hence $|\mathbf{Q_a}|$ is also 2. QT is again calculated by taking the dot product between Q in all its possible positions in T. QT_a calculated similarly, but now again, the values of T are first aggregated by average with all aggregation queries in $\mathbf{Q_a}$. We rewrite all values in QT_a to fit inside a single fraction.

Next, we fill in Formula 4.6 on the right-hand side of the figure using the same values to see if we get the same result. Q_{aE} is, again, the element-wise product of $\mathbf{Q_a}$ and you can see the result of what that looks like in the figure. If we next take the dot product of Q_{aE} and Q, we get a single value, as shown in the figure. Finally, we add that value to each value in QT and divide those by 3, which the value of $|\mathbf{Q_a}| + 1$ in the current example. As can be seen, the result we get is the same as the rewritten QT_a, showing that QT_a can also be calculated using QT.

4.2.2.2 Modifying ComputeSlidingMeanStd for aggregation

The second place where the calculation needs to change is at ComputeSlidingMeanStd. However, while for the sliding means M_T it might be possible to apply the aggregation afterward, for the sliding standard deviations Σ_T this is very difficult and perhaps impossible. Hence we need to modify ComputeSlidingMeanStd itself. See Section 4.1.2.1 and Algorithm 2 for the original ComputeSlidingMeanStd algorithm.

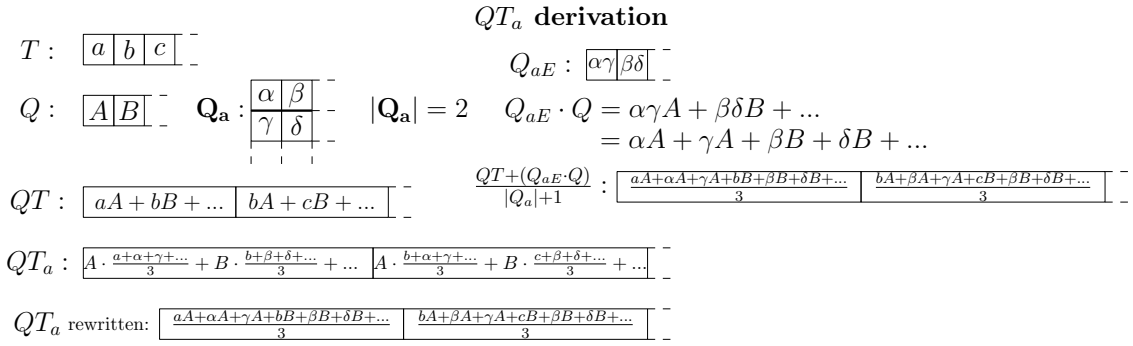

Figure 4.4: Example to show the derivation of Formula 4.6, so it can be used to get QT_a using QT from Figure 4.2.

Compared to the original ComputeSlidingMeanStd output (M_T and Σ_T), where each value in the mean- and standard deviation array adheres to Formulas 4.4 and 4.5, we want the new ones (M'_T and Σ'_T) to adhere to Formula 4.7 and 4.8, respectively. In short, each window taken from time series T first needs to be aggregated by average using all aggregation queries in $\mathbf{Q_a}$ before calculating the mean and standard deviation.

To clarify $\sum \mathbf{Q_a}$, seen in the formulas: This represents the element-wise sum of each aggregation query in $\mathbf{Q_a}$. In other words, each array in $\mathbf{Q_a}$ is aggregated into one by summing up the elements at the same index in each array. The same element-wise addition happens when this result is added to a slice of T in both formulas. For more clarification of some of the notation used, see the notation part in Section 2.1.

$$M'[i] = \frac{\sum \frac{T[i..i+m] + \sum \mathbf{Q_a}}{|\mathbf{Q_a}|+1}}{m} \tag{4.7}$$

$$\Sigma'[i] = \sqrt{\frac{\sum \left(\frac{T[i..i+m] + \sum \mathbf{Q_a}}{|\mathbf{Q_a}|+1} - M'[i] \right)^2}{m}} \tag{4.8}$$

We want to be able to calculate the sliding means and -standard deviations without having to query the time series T for each possible window because then most elements would need to be queried m times during the calculation. Instead, we want to only have to query each element one or two times at most to keep the running time constant and use a sums- and squares cache, similar to the non-aggregated variant of ComputeSlidingMeanStd. For the aggregated means M'_T, the fraction can be split up to reuse the calculation for the non-aggregated means M_T, for which we already know it can be calculated entirely using those caches. The calculation to show how this split-up can be done is shown in Formula 4.9.

$$
\begin{aligned}
M'[i] &= \frac{\frac{\sum T[i..i+m] + \sum \mathbf{Q_a}}{|\mathbf{Q_a}|+1}}{m} \\
&= \frac{\frac{\sum (T[i..i+m] + \sum \mathbf{Q_a})}{|\mathbf{Q_a}|+1}}{m} && \text{First summing and then dividing is the same as the other way around} \\
&= \frac{\frac{\sum (T[i..i+m] + \sum \mathbf{Q_a})}{m}}{|\mathbf{Q_a}|+1} && \text{Swapping division order} \\
&= \frac{\frac{\sum T[i..i+m] + \sum\limits_{Q_a \in \mathbf{Q_a}} \sum Q_a}{m}}{|\mathbf{Q_a}|+1} && \text{Distributing sums over each element, since sum(A + B) = sum(A) + sum(B)} \\
&= \frac{\frac{\sum T[i..i+m]}{m} + \sum\limits_{Q_a \in \mathbf{Q_a}} \frac{\sum Q_a}{m}}{|\mathbf{Q_a}|+1} && \text{Distributing division by } m \text{ over each element}
\end{aligned}
\tag{4.9}
$$

This means the same sum-cache solution (which provides $\sum T[i..i+m]$) can also be applied to calculate the aggregated sliding means. This can also be done without losing speed since $\frac{\sum T[i..i+m]}{m}$ is the only piece of the formula that is dependent on i. Since this is the same as $M[i]$, we know this can use the cache. The rest can be pre-calculated once and reused for all is.

For the aggregated standard deviations Σ'_T, we would also like to reuse the sum- and squares-cache, $\sum T[i..i+m]$ and $\sum T[i..i+m]^2$ respectively, if possible. The downside is that the formula for the standard deviation becomes rather large when aggregation is applied. However, in this expanded form, which is calculated and shown in Formula 4.10, we can look at each individual term and see how the caches can be applied to ensure we do not have to query full windows from time series T itself.

$$
\begin{aligned}
\Sigma'[i] &= \sqrt{\frac{\sum\left(\frac{T[i..i+m]+\sum \mathbf{Q_a}}{|\mathbf{Q_a}|+1} - M'[i]\right)^2}{m}} \\
\Sigma'[i] &= \sqrt{\frac{s(i)}{m}} && \text{Splitting up formula} \\
s(i) &= \sum\left(\frac{T[i..i+m]+\sum \mathbf{Q_a}}{|\mathbf{Q_a}|+1} - M'[i]\right)^2 \\
&= \sum\left(\left(\frac{T[i..i+m]+\sum \mathbf{Q_a}}{|\mathbf{Q_a}|+1}\right)^2 - \frac{T[i..i+m]+\sum \mathbf{Q_a}}{|\mathbf{Q_a}|+1}2M'[i] + M'[i]^2\right) && \text{Distributing square} \\
&= \sum\left(\frac{T[i..i+m]^2 + 2T[i..i+m]\sum \mathbf{Q_a} + (\sum \mathbf{Q_a})^2}{(|\mathbf{Q_a}|+1)^2}\right. && \text{Distributing squares further} \\
&\quad \left. - \frac{2M'[i]T[i..i+m] + 2M'[i]\sum \mathbf{Q_a}}{|\mathbf{Q_a}|+1} + M'[i]^2\right) \\
&= \sum\left(\frac{T[i..i+m]^2}{(|\mathbf{Q_a}|+1)^2} + \frac{2T[i..i+m]\sum \mathbf{Q_a}}{(|\mathbf{Q_a}|+1)^2} + \frac{(\sum \mathbf{Q_a})^2}{(|\mathbf{Q_a}|+1)^2}\right. && \text{Splitting fractions} \\
&\quad \left. - \frac{2M'[i]T[i..i+m]}{|\mathbf{Q_a}|+1} - \frac{2M'[i]\sum \mathbf{Q_a}}{|\mathbf{Q_a}|+1} + M'[i]^2\right) \\
&= \underbrace{\frac{\sum T[i..i+m]^2}{(|\mathbf{Q_a}|+1)^2}}_{(a)} + \underbrace{\frac{2\sum(T[i..i+m]\sum \mathbf{Q_a})}{(|\mathbf{Q_a}|+1)^2}}_{(b)} + \underbrace{\frac{\sum(\sum \mathbf{Q_a})^2}{(|\mathbf{Q_a}|+1)^2}}_{(c)} && \text{Merging down sum} \\
&\quad - \underbrace{\frac{2M'[i]\sum T[i..i+m]}{|\mathbf{Q_a}|+1}}_{(d)} - \underbrace{\frac{2M'[i]\sum(\sum \mathbf{Q_a})}{|\mathbf{Q_a}|+1}}_{(e)} + \underbrace{mM'[i]^2}_{(f)}
\end{aligned}
\tag{4.10}
$$

For each of the queries, $(|\mathbf{Q_a}|+1)^2$ and $|\mathbf{Q_a}|+1$ can be pre-calculated. This value is not dependent on i and can thus be considered a constant.

The first term, (a), consists of $\sum T[i..i+m]^2$, which can be obtained by the squares cache.

The second term, (b), is impossible to do with sum- and square caches since it contains a pairwise product between $T[i..i+m]$ and $\sum \mathbf{Q_a}$. However, summing the pairwise product is the same as the dot product: $\sum(T[i..i+m]\sum \mathbf{Q_a}) = T[i..i+m]\cdot\sum \mathbf{Q_a}$. $\sum \mathbf{Q_a}$ stays constant for each i, but as i chances, $T[i..i+m]$ does slide along T. While using the sum- and square caches might be impossible, we can still use SlidingDotProducts [13] for some extra speed. It can be called once using T and $\sum \mathbf{Q_a}$ and the values calculated can simply be queried using i when this term is calculated.

The third term, (c), has no components dependent on i, which means the entire component can be pre-calculated and reused for each i.

The fourth term, (d), contains $M'[i]$, which we already know can be calculated using just the caches. Plus, since the standard deviations and means are usually calculated simultaneously, the results of M' can simply be reused for calculating the standard deviations. It also contains $\sum T[i..i+m]$ which can be obtained by the sums cache.

The fifth term, (e), again contains $M'[i]$, which can be done using caches. It contains $\sum(\sum \mathbf{Q_a})$, which also does not depend on i and thus can be pre-calculated and reused.

Finally, the sixth term, (f), consists of just a constant and a value from $M'[i]$, which we already know can be given with caches.

Based on these findings, we propose Algorithm 7. The experiments, Section 6.2.3, show that the algorithm is slower than the non-aggregated variant (Algorithm 2), however, not by much.

Algorithm 7: ComputeSlidingMeanStdWithAgg(T, m, $\mathbf{Q_a}$)
Input: T: time series. m: window size. $\mathbf{Q_a}$: array of aggregation queries, each of size m, to aggregate T with.
Output: Sliding mean- and sliding standard deviation array.

$n \leftarrow |T|$
$q \leftarrow |\mathbf{Q_a}| + 1$
M'_T, $\Sigma'_T \leftarrow n - m + 1$ sized empty arras
$q_{a\mu s} \leftarrow \sum_{Q_a \in \mathbf{Q_a}} \frac{Q_a}{m}$ // each aggregation query's mean added up
$Q_{as} \leftarrow \sum \mathbf{Q_a}$ // all aggregation queries aggregated into one by addition
$Q_{asd} \leftarrow Q_{as} \cdot Q_{as}$ // sum of squares of Q_{as} (dot product)
$Q_{ass} \leftarrow \sum Q_{as}$ // sum of by-sum-aggregated queries
$TQ_{as} \leftarrow$SlidingDotProducts(Q_{as}, T) // see [13]

sums $\leftarrow \sum T[0..m-1]$ // initially sum of first window
squares $\leftarrow T[0..m-1] \cdot T[0..m-1]$ // initially sum of squares of first window (dot product)

for all $i \in (0..n-m)$ **do**
  $\mu \leftarrow sums/m$ // calculate mean normally
  $\mu'_a \leftarrow (\mu + q_{a\mu s})/q$ // apply aggregation to mean
  $M'_T[i] \leftarrow \mu'_a$

  $stdSum \leftarrow$ squares$/q^2$
    $+2TQ_{as}[i]/q^2$
    $-2\mu'_a$ sums$/q$
    $+Q_{asd}/q^2$
    $-2\mu'_a Q_{ass}/q$
    $+m\mu'^2_a$
  $\Sigma'_T[i] \leftarrow \sqrt{stdSum/m}$

  if $i < n - m$ **then**
    sums $\leftarrow sums - T[i] + T[i+m]$
    squares $\leftarrow squares - T[i]^2 + T[i+m]^2$
  end if
end for

return M'_T, Σ'_T

4.3 Optimizations for STAMP 3D

Calculating something as heavy as this triple-nested for-loop takes its toll time-wise. So it is all the more important to apply as many optimizations as possible to make the process more efficient. Luckily, due to the structure of the problem, there are a couple of possibilities to improve the algorithms. In this section, we split up the optimizations into vertical and horizontal: Vertical optimizations (Section 4.3.1) entailing parts of the problem being pre-calculated in a higher level of the for-loop structure and reused downwards wherever possible. Horizontal optimizations (Section 4.3.2) being where an iteration of a for-loop can provide information to the next iteration at the same for-loop level, like a cache for sliding dot product calculations. Afterward, we give a brief overview of all optimizations in place (Section 4.3.3) and we explain how lag bound can also help to optimize STAMP 3D (Section 4.3.4).

The for-loop structure of the algorithms can be shown as a more generalized version in STAMP-Pearson-3TS-Generalized (Algorithm 8). We will apply the optimizations to this version of the algorithm in this section because breaking it up into multiple separate algorithms (as done previously in this chapter) might over-complicate it. The end result can be seen in Algorithm 10.

Algorithm 8: STAMP-Pearson-3TS-Generalized(T_A, T_B, T_C, m, r)
Input: T_A, T_B, T_C: time series. m: window size. r: MIN or MAX reducer function, returns the indices, best aggregation method, and value of best window placements.
Output: Indices of best positions of the window in T_A, T_B, and T_C. Aggregation method used. Value of the best correlation found.

```
// STAMP-Pearson-3TS (Algorithm 3)
for all Q_C ∈ AllSubsequences(T_C, m) do

  // STAMP-Pearson-3TS-Sub (Algorithm 4)
  for all Q_B ∈ AllSubsequences(T_B, m) do

    // MASS-Pearson-3TS (Algorithm 5)

    P_A ←MASS-Pearson-Agg(T_A, [Q_B], Q_C, r) // Algorithm 6

    P_B ←MASS-Pearson(T_A, (Q_B+Q_C)/2, r) // Algorithm 1

    P_C ←MASS-Pearson-Agg(T_A, [Q_C], Q_B, r) // Algorithm 6

  end for
end for
```

4.3.1 Vertical optimizations

All vertical optimizations consist of efficiently calculating a set of values in a higher level of the for-loop structure for reuse downwards.

Here are some that are implemented and tested: Sliding means and -standard deviations (Section 4.3.1.1) and Sliding dot products (Section 4.3.1.2).

4.3.1.1 Sliding means and -standard deviations

At many points in the algorithm, having a precalculated sliding means and -standard deviations array can come in handy and save calculations that would otherwise be repeated. At the highest level, STAMP-Pearson-3TS (Algorithm 3), the sliding means and -standard deviations are calculated of all three time series.

The calculation of P_B in MASS-Pearson-3TS (Algorithm 5) can benefit directly from the sliding means and -standard deviations of time series T_A: M_{T_A} and Σ_{T_A}. These would otherwise

be calculated inside MASS-Pearson (Algorithm 1) itself anyways, so this saves a lot of work.

The calculation of P_C in MASS-Pearson-3TS (Algorithm 5), where a subsection Q_B of time series T_B is used, can benefit from the sliding means and -standard deviations of time series T_B: M_{T_B} and Σ_{T_B}. Inside Mass-Pearson-Agg (Algorithm 6), the mean and standard deviation of Q_B, μ_{Q_B} and σ_{Q_B}, would be calculated anyway, but now it can simply take these values using the efficiently precalculated ones in M_{T_B} and Σ_{T_B}.

Very similarly, the calculation of P_A in MASS-Pearson-3TS (Algorithm 5), where a subsection Q_C of time series T_C is used, can benefit from the sliding means and -standard deviations of time series T_C: M_{T_C} and Σ_{T_C}. This removes the obligation of having to calculate μ_{Q_C} and σ_{Q_C} inside Mass-Pearson-Agg (Algorithm 6).

All calculations of ComputeSlidingMeanStdWithAgg (Algorithm 7) can benefit from having the sliding means of the time series T, M_T, precalculated. This is because the aggregated sliding means M'_T can be calculated like $M'_T[i] = \frac{M_T[i] + \sum_{Q_a \in \mathbf{Q_a}} \frac{Q_a}{m}}{|\mathbf{Q_a}|+1}$ (as derived from Functions 4.9 and 4.4). This means that the calculation of P_A in MASS-Pearson-3TS (Algorithm 5) can also benefit from having M_{T_A} precalculated, since ComputeSlidingMeanStdWithAgg (Algorithm 7) is used inside Mass-Pearson-Agg (Algorithm 6) to calculate the sliding means and -standard deviations of time series T_A aggregated with query Q_B: $M'_{T_AQ_B}$ and $\Sigma'_{T_AQ_B}$.

The calculation of P_C in MASS-Pearson-3TS (Algorithm 5) can benefit from having the aggregated sliding means and -standard deviations of time series T_A and query Q_C: $M'_{T_AQ_C}$ and $\Sigma'_{T_AQ_C}$. These can be calculated once for each window in time series T_C at the for-loop level of STAMP-Pearson-3TS-Sub (Algorithm 4) and reused downwards. As a bonus, like before, M_{T_A} can again be used to save some time when calculating.

Finally, the STAMP-Pearson-3TS algorithm usually is run for many groups of three. A single time series occurs in multiple groups. Calculating the sliding means and -standard deviations for each time series each time STAMP-Pearson-3TS is called is a waste of computing power. The algorithm can benefit a lot from having those calculated once for each time series before and have these values supplied to the algorithm itself.

4.3.1.2 Sliding dot products

Both the calculation of P_A and P_C in MASS-Pearson-3TS (Algorithm 5) can benefit from the dot product between the current query Q_B and query Q_C because in these two cases, this corresponds to the $(Q_{aE} \cdot Q)$ part in MASS-Pearson-Agg (Algorithm 6), meaning it does not have to be recalculated. This value can be efficiently calculated using SlidingDotProducts [13] between time series T_B and query Q_C, Q_CT_B, at the STAMP-Pearson-3TS-Sub (Algorithm 4) for-loop level and provided to the level below one at a time for each query Q_B.

The calculation of P_A in MASS-Pearson-3TS (Algorithm 5) can benefit from having the sliding dot products between query Q_C and time series T_A, Q_CT_A, precalculated. This value represents QT in Mass-Pearson-Agg (Algorithm 6) and thus prevents the value from being recalculated multiple times. It can be calculated once for each query Q_C at the STAMP-Pearson-3TS-Sub (Algorithm 4) for-loop level and provided downwards.

4.3.2 Horizontal optimizations

Horizontal optimizations consist of providing information from each iteration to the next iteration on the same for-loop level. These optimizations do take away the ability for the algorithm to be split up into multiple threads. This is because the previous iteration needs to happen before the current to get the information in time. Without horizontal optimizations, each iteration could, in theory, run in parallel. The one horizontal optimization which is tested and implemented is the sliding dot products cache, for which the concept is explained in Section 4.3.2.1 and its application in Section 4.3.2.2.

4.3.2.1 Sliding dot products cache concept

Having the result of SlidingDotProducts(Q_{i-1}, T)[13] is highly beneficial for calculating the result of SlidingDotProducts(Q_i, T) more efficiently. The speed gain is best seen when the window size is relatively large. The concept of having a sliding dot products cache is also explored in the second Matrix Profile paper [17] in the STOMP algorithm. This concept redesigned by us to work for our 3TS variant as well. To explain how, we give a small mathematical explanation in Formula 4.11.

Say we have a time series T and two adjacent queries Q_{i-1} and Q_i:

$$\begin{aligned} T &= [a, b, c, d, e] \\ Q_{i-1} &= [\alpha, \beta, \gamma] \\ Q_i &= [\beta, \gamma, \delta] \end{aligned}$$

We can then see that the sliding dot products results, $Q_{i-1}T$ and Q_iT, have some overlap:

$$\begin{aligned} Q_{i-1}T &= [\alpha a + (\beta b + \gamma c),\ \alpha b + (\beta c + \gamma d),\ \alpha c + \beta d + \gamma e] \\ Q_iT &= [\beta a + \gamma b + \delta c,\ (\beta b + \gamma c) + \delta d,\ (\beta c + \gamma d) + \delta e] \end{aligned}$$

This means that each iteration (except for the first) each element in QT can be expressed using an element from the previous QT cache, the previous query Q_{i-1}, query Q_i, and the time series T:

$$Q_iT[i] = Q_{i-1}T[i-1] - Q_{i-1}[0]\ T[i-1] + Q_i[m-1]\ T[i+m-1] \tag{4.11}$$

This idea is applied to create UpdateQTCache (Algorithm 9). As can be seen, for the first iteration, when the cache is still empty, the algorithm simply uses SlidingDotProducts [13] to initialize the cache. For the next iterations, from back to front, each value in the cache is updated, finishing off with calculating the first value using a normal dot product.

Algorithm 9: UpdateQTCache(QT_{cache}, Q, T, Q_{prev})
Input: QT_{cache}: $n - m + 1$ sized cache array that will be updated. T: time series of size n. Q: query of window size. Q_{prev}: previous query (only if QT_{cache} is not empty).

```
if QT_cache is empty then
    QT_cache ← SlidingDotProducts(Q, T) [13]
else
    // Note: for-loop moves downwards
    for all i ∈ (n − m .. 1) do
        QT_cache[i] ← QT_cache[i − 1] − Q_prev[0] T[i − 1] + Q[m − 1] T[i + m − 1]
    end for
    QT_cache[0] ← Q · T[0 .. m − 1]
end if
```

4.3.2.2 Applying sliding dot products cache

The calculation of P_C in MASS-Pearson-3TS (Algorithm 5) can take advantage of the sliding dot products cache of time series T_A and the queries Q_B from time series T_B: Q_BT_{Acache}. The value of QT inside MASS-Pearson-Agg (Algorithm 6) can simply be set to QT_{cache} after running UpdateQTCache (Algorithm 9) on it in the same algorithm. The previous query Q_{Bprev} can be

stored at the for loop level of STAMP-Pearson-3TS-Sub (Algorithm 4) and provided to the lower algorithms, while Q_BT_{Acache} can be stored at the top-most level.

The sliding dots products between time series T_A and query Q_C, Q_CT_A, and T_B and query Q_C, Q_CT_B, at the STAMP-Pearson-3TS-Sub (Algorithm 4) level in use by the vertical optimization at Section 4.3.1.2 can both also implement the new sliding dot products cache. By storing the dot products cache of both, Q_CT_{Acache} and Q_CT_{Bcache}, again at the top for-loop level, as well as the previous query Q_{Cprev}, the calculations can be sped up quite a bit.

4.3.3 Vertical- and horizontal optimizations overview

With all these optimizations applied to STAMP-Pearson-3TS-Generalized (Algorithm 8), we propose STAMP-Pearson-3TS-Generalized-Opt (Algorithm 10).

A note for notation: Some arguments of algorithms have a blue part. These arguments are named arguments. We use this notation to show how the argument given is used in the called algorithm. In most cases this means that the blue value does not have to be calculated anymore inside the called algorithm as it has been calculated already. See the notation section, Section 2.1, for more information about notations.

Algorithm 10: STAMP-Pearson-3TS-Generalized-Opt(T_A, T_B, T_C, m, r, M_{T_A}, Σ_{T_A}, M_{T_B}, Σ_{T_B}, M_{T_C}, Σ_{T_C})
Input: T_A, T_B, T_C: time series. m: window size. r: MIN or MAX reducer function, returns the indices, best aggregation method, and value of best window placements. Σs and Ms: Precalculated sliding means and -standard deviations for each time series.
Output: Indices of best positions of the window in T_A, T_B, and T_C. Aggregation method used. Value of the best correlation found.

```
// STAMP-Pearson-3TS (Algorithm 3)
Q_C T_Acache, Q_C T_Bcache, Q_B T_Acache ← Empty arrays of (respective) size |T_*| − m + 1
Q_Cprev ← NIL
for all Q_C, i_C ∈ AllSubsequencesWithIndex(T_C, m) do

    // STAMP-Pearson-3TS-Sub (Algorithm 4)
    Q_C T_A ← UpdateQTCache(Q_C T_Acache, Q_C, T_A, Q_Cprev)
    Q_C T_B ← UpdateQTCache(Q_C T_Bcache, Q_C, T_B, Q_Cprev)
    M'_{T_A Q_C}, Σ'_{T_A Q_C} ← ComputeSlidingMeanStdWithAgg(T_A, m, [Q_C], M_{T_A})
    Q_Bprev ← NIL
    for all Q_B, i_B ∈ AllSubsequencesWithIndex(T_B, m) do

        // MASS-Pearson-3TS (Algorithm 5)
        P_A ←MASS-Pearson-Agg(T_A, [Q_B], Q_C, r, Q_aE · Q = Q_C T_B[i_B], QT = Q_C T_A,
        μ_Q = M_{T_C}[i_C], σ_Q = Σ_{T_C}[i_C], M_T = M_{T_A}) // Algorithm 6

        P_B ←MASS-Pearson(T_A, (Q_B+Q_C)/2, r, M_T = M_{T_A}, Σ_T = Σ_{T_A}) // Algorithm 1

        P_C ←MASS-Pearson-Agg(T_A, [Q_C], Q_B, r, Q_aE · Q = Q_C T_B[i_B], μ_Q = μ_{Q_B}, σ_Q = σ_{Q_B},
        M'_T = M'_{T_A Q_C}, Σ'_T = Σ'_{T_A Q_C}, QT_cache = Q_B T_Acache, Q_Bprev = Q_Bprev) // Algorithm 6

        Q_Bprev ← Q_B
    end for
    Q_Cprev ← Q_C
end for
```

4.3.4 Lag bound

Lag bound, also called enveloping, can be used to limit the maximum distance that windows can be apart during calculations. Depending on the domain, lag bound might be applied to greatly limit the number of calculations necessary to get a satisfactory result. For instance, it might not be very meaningful to find a correlation between a stock in January and another in September.

In this section, we first explain the concept of lag bound and the effect it has (Section 4.3.4.1). Afterward, we give some notes we faced when integrating it (Section 4.3.4.2).

4.3.4.1 Concept

The concept of applying lag bound, denoted by l, to the 3TS STAMP Pearson algorithms consists of 'cutting off' pieces from the other time series while sliding the window over one. See Figure 4.5 for an example.

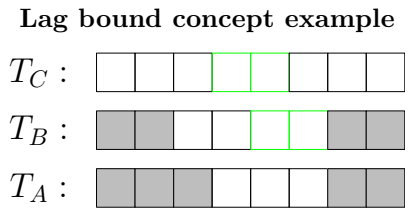

Figure 4.5: $m = 2$, $l = 1$

In the time series T_C, the window (in green) is slid over the entire time series, but at each position, all slots in time series T_B that are more than the lag bound away are cut off (colored gray). The window in these time series will not use the gray slots. This makes time series T_B considerably smaller and removes a lot of positions.

Now, when the window is slid over the remaining piece of time series T_B, every slot that is more than the lag bound away from the window in time series T_C ánd the window in time series T_B is cut off, leaving, in this example, just 3 slots or 2 window positions in time series T_A that have to be considered.

Implementing this thus lowers the total amount of window positions from $(n - m + 1)^3 = 343$ to just 43 in this specific example.

While it might be possible to provide a long and cumbersome formula to give how many positions are saved for each time series length, window size, and lag bound value, it will be a lot clearer to show how much time it can save when using it the calculations. This is done in the experiments section, Section 6.2.2.

4.3.4.2 Note on integration

The combination of lag bound with the STAMP-Pearson-3TS algorithms consists of providing lower for-loop levels with smaller pieces of the time series. This will ensure that fewer window positions will be used there. Due to the time series being shorter and potentially starting at another index than originally, the new starting index must be passed on to the lower for-loop levels as well. This is to make sure that the correct values are taken from M_{T_B}, Σ_{T_B}, M_{T_A}, Σ_{T_A}, $M'_{T_AQ_C}$, $\Sigma'_{T_AQ_C}$ (Section 4.1.2.1), Q_CT_A (Section 4.3.1.2), Q_CT_{Bcache}, and Q_BT_{Acache} (Section 4.3.2.2).

4.4 Group-by-Key Method

The chapter, up to here, describes how the lagged correlation within a group of three can be calculated. What it does not describe yet, however, is how these groups of three can efficiently

be generated in a data engineering toolkit, such as Apache Spark[1], when given a dataset of many time series. In this section, we propose a technique to do so in an efficient and distributed manner.

To do this, we can take advantage of the `groupByKey` function, present in most toolsets, and generate all possible groups of any desired group size. Our method works by building an imaginary matrix of '*group size*' dimensions of length '*amount of time series in the set*' for each dimension. After this, each slot is given a unique index, while those which are not needed are omitted. This includes self-comparisons or those slots that already are paired up.

Next, for each time series in the set, the same amount of key-value pairs are created, namely for each row, column, etc. in the table of that time series (see Table 4.2 for an example in 2D). The key in this pair is the number from the table and the value is (a reference to) the time series itself.

Now to find the groups of time series, simply perform `groupByKey` on all the key-value pairs, which can be done very efficiently and in a distributed fashion in a toolkit like Apache Spark[2]. This will group all the values (time series) for pairs with the same key.

After using this method, the groups formed can also be distributed evenly across the workers, which can divide the upcoming calculations effectively.

To be specific, the only values accepted for the matrix (which form this pyramid shape) are those for which the following holds, where the tuple I represents a multidimensional index for the matrix (both I and the matrix are assumed to start their index at 0, which corresponds best to the final implementation):

$$\text{Index } I \text{ is allowed in the matrix } \textbf{iff } \forall_{j\in\{0,\dots,|I|-1\}} \; \forall_{k\in\{j+1,\dots,|I|)\}} \; I_j - I_k > 0 \tag{4.12}$$

The value for the specific index is defined using the following formula, again where I is a tuple representing a multidimensional index and g represents the desired group size:

$$value(I, g) = \sum_{j\in\{0,\dots,|I|-1\}} I_j \cdot g^j \tag{4.13}$$

The result of creating a table using those two formulas can be seen in Table 4.2, where there are 5 time series in the set and groups of 2 are to be found (hence it being a 2D matrix). The keys for the time series C are highlighted.

	A	B	C	D	E
A					
B	5				
C	**10**	**11**			
D	15	16	**17**		
E	20	21	**22**	23	

Table 4.2: 2D matrix example

The two formulas (4.12, 4.13) were formed by first looking at a 2D example, as seen in the table (4.2) and then seeing what it would look like in 3D and so on. The formulas are also tested and proven to work for higher dimensions, but this is difficult to show on paper and prove theoretically.

Since this method already evenly divides the work so well, as is found when using it implemented for Apache Spark[2], it is easy to not look further at other methods. That said, the method does have its downsides. For instance, when there are a lot of time series and groups of large size need to be calculated, a lot of work and storage may still be required to save all the indices formed. Whether there exist faster or more efficient methods is outside the scope of this thesis.

[1] https://spark.apache.org/
[2] https://spark.apache.org/

Chapter 5

Can we skip window placements to find lagged correlations faster?

One way to further increase the efficiency of finding lagged correlations is by skipping calculations. Since we are only interested in the best correlation between three time series at certain window positions, any positions which would result in an as-good or worse correlation can be skipped.

In this chapter, we explain what we discovered trying to save calculations by skipping certain window positions. Firstly, we look at skipping straight lines (Section 5.1), next skipping repeating patterns or motifs (Section 5.2). We also look at how the windows to be skipped can be marked (Section 5.3) and apply this to both techniques to provide algorithms (Sections 5.4 and 5.5). After that, we apply simplification to improve both algorithms at the cost of accuracy (Section 5.6). Finally, we explain how this skipping of window positions can be applied to the (optimized) STAMP-Pearson-3TS algorithms of the previous chapter (Section 5.7).

5.1 Straight line skipping

When a time series contains a straight piece larger than the window size, all window placements where the window is completely inside the straight line will result in the same correlation value. We show an example of this using two generic time series T_A and T_B in Figure 5.1. T_A consists of a straight line and T_B can contain any values.

Two generic time series

T_A :	$a+1b$	$a+2b$	$a+3b$	$a+4b$	$a+5b$
T_B :	c	d	e	f	g

Figure 5.1: Windows T_{A_1} and T_{B_2} are highlighted in green.

Using a window size of 3, there are three possible window positions in T_A: $T_{A_1} = [a+1b, a+2b, a+3b]$ (highlighted in green), $T_{A_2} = [a+2b, a+3b, a+4b]$, and $T_{A_3} = [a+3b, a+4b, a+5b]$. We now calculate the Pearson correlation for all these window positions and $T_{B_2} = [d, e, f]$ in Formula 5.1.

In terms of notation: Simple operations on series, lists, and arrays are considered element-wise. For example, $A+1$ would result in a new array similar to A but with each element being 1 larger. For more information on notation, see Section 2.1.

$$pearson(X,Y) = \frac{\sum(X-\overline{X})(Y-\overline{Y})}{\sqrt{\sum(X-\overline{X})^2}\sqrt{\sum(Y-\overline{Y})^2}} = \frac{\sum(X-\overline{X})(Y-\overline{Y})}{\sigma_X\sigma_Y}$$

$$\begin{aligned}
\overline{T_{A_1}} &= \frac{a+1b+a+2b+a+3b}{3} = a+2b \\
\sigma_{T_{A_1}} &= \sqrt{(a+1b-(a+2b))^2+(a+2b-(a+2b))^2+(a+3b-(a+2b))^2} = \sqrt{2}b \\
\overline{T_{A_2}} &= \frac{a+2b+a+3b+a+4b}{3} = a+3b \\
\sigma_{T_{A_2}} &= \sqrt{(a+2b-(a+3b))^2+(a+3b-(a+3b))^2+(a+4b-(a+3b))^2} = \sqrt{2}b \\
\overline{T_{A_3}} &= \frac{a+3b+a+4b+a+5b}{3} = a+4b \\
\sigma_{T_{A_3}} &= \sqrt{(a+3b-(a+4b))^2+(a+4b-(a+4b))^2+(a+5b-(a+4b))^2} = \sqrt{2}b
\end{aligned}$$

$$\overline{T_{B_2}} = \frac{def}{3}$$

$$\begin{aligned}
pearson(T_{A_1},T_{B_2}) &= \frac{\sum(T_{A_1}-\overline{T_{A_1}})(T_{B_2}-\overline{T_{B_2}})}{\sqrt{2}b\sigma_{T_{B_2}}} \\
&= \frac{\sum([a+1b,a+2b,a+3b]-a-2b)([d,e,f]-\frac{def}{3})}{\sqrt{2}b\sigma_{T_{B_2}}} \\
&= \frac{\sum[-b,0,b][d-\frac{def}{3},e-\frac{def}{3},f-\frac{def}{3}]}{\sqrt{2}b\sigma_{T_{B_2}}}
\end{aligned}$$

$$\begin{aligned}
pearson(T_{A_2},T_{B_2}) &= \frac{\sum(T_{A_2}-\overline{T_{A_2}})(T_{B_2}-\overline{T_{B_2}})}{\sqrt{2}b\sigma_{T_{B_2}}} \\
&= \frac{\sum([a+2b,a+3b,a+4b]-a-3b)([d,e,f]-\frac{def}{3})}{\sqrt{2}b\sigma_{T_{B_2}}} \\
&= \frac{\sum[-b,0,b][d-\frac{def}{3},e-\frac{def}{3},f-\frac{def}{3}]}{\sqrt{2}b\sigma_{T_{B_2}}}
\end{aligned}$$

$$\begin{aligned}
pearson(T_{A_3},T_{B_2}) &= \frac{\sum(T_{A_3}-\overline{T_{A_3}})(T_{B_2}-\overline{T_{B_2}})}{\sqrt{2}b\sigma_{T_{B_2}}} \\
&= \frac{\sum([a+3b,a+4b,a+5b]-a-4b)([d,e,f]-\frac{def}{3})}{\sqrt{2}b\sigma_{T_{B_2}}} \\
&= \frac{\sum[-b,0,b][d-\frac{def}{3},e-\frac{def}{3},f-\frac{def}{3}]}{\sqrt{2}b\sigma_{T_{B_2}}}
\end{aligned} \tag{5.1}$$

As can be seen, the resulting Pearson correlation is exactly the same for all three window positions, meaning that if we calculate just one, we would not have to calculate the others anymore.

The downside to this approach, however, is that it only works if the window size is smaller than the straight line piece in the time series, which we did not encounter often, aside from data with a lot of interpolation. We speculate that most datasets, especially stock data, fluctuate a lot

from one point in time to the next. Aside from that, large window sizes are also more commonly used, since they can sometimes indicate better whether there is a correlation between time series.

5.2 Motif skipping

Since long straight pieces do not occur often in time series of real-world data, we decided to also take a look at other repeating patterns. If the same pattern occurs multiple times in the same time series, all window positions fully inside this pattern only need to be checked once since the correlation would be the same inside a different instance of this pattern. To detect these repeating patterns or motifs, we once again look at the Matrix Profile [13]. This time, however, we do not need to modify it, since self-similarity is exactly what the Matrix Profile was built for. In Figure 5.2 we give an example of a small time series with repeating patterns to show how the (Pearson) Matrix Profile detects them. We did choose to use a Pearson result instead of the Euclidean distance, since this normalizes the values between -1.0 and 1.0, making them easier to work with.

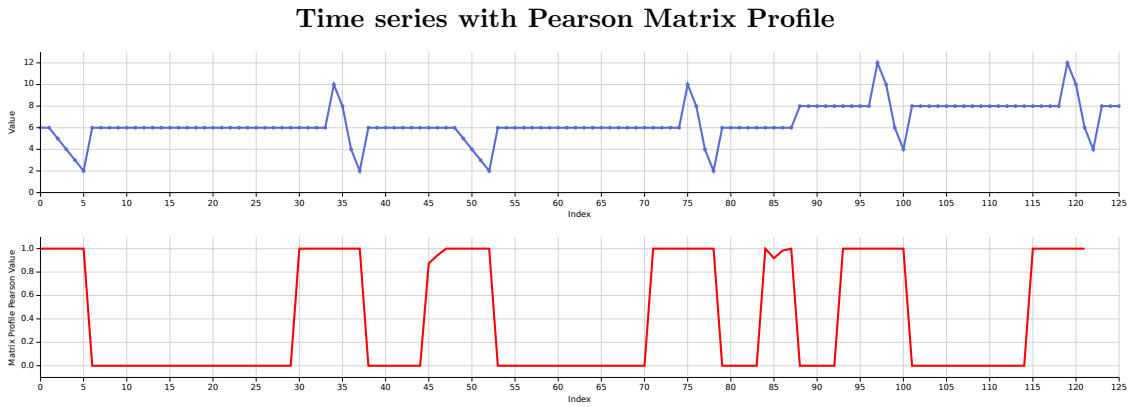

Figure 5.2: Time series (blue) and accompanying Pearson Matrix Profile (red) calculated with a window size of 5.

As can be seen, the Pearson Matrix Profile becomes 1.0 each time a repeating pattern occurs in the time series and else 0.0 (there is a small error at index 85, which can occur). Aside from this, the Matrix Profile also supplies a list of window indices which, for each item in the list, points at another window index in the time series where the pattern is similar. This can be any other instance of the same motif or even the same instance in some cases.

Using this indices list and the Pearson Matrix Profile we can utilize the clustering UnionFind algorithm to build a list of repeating patterns where, for each, only one instance needs to be checked to find the best correlation.

It does need to be noted that the amount of skippable window placements depends on the window size. This can also be seen in the relevant experiments, Section 6.3.2. However, when a data analyst is trying to find a good window size, it could be interesting to determine the window size based on how many positions can be saved using this approach.

5.3 Marking which window positions to skip

One way to mark which window positions to skip without having to store extra indices is by simply removing parts of the data in the time series itself. In practice, this is done by setting them to NaN, as that value can still be stored in a primitive double array. If a window contains a NaN value, it is skipped in the calculations. An example of this can be seen in Figure 5.3. With the sliding windows, however, it is not that straightforward which positions of the window can

be skipped when a repeating part or straight line occurs. This is why we provide some further explanation.

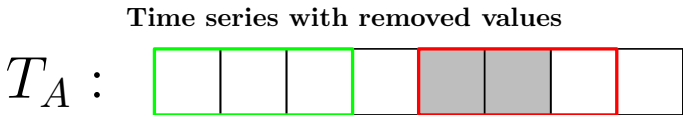

Figure 5.3: Windows containing any removed (gray) data points are skipped (red example), while others are still calculated (green example).

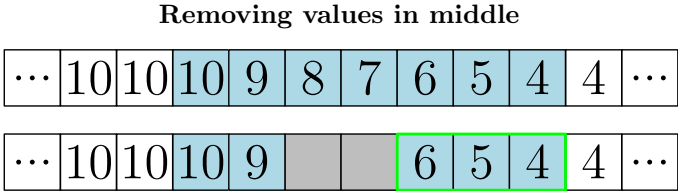

Figure 5.4: Blue: Straight piece longer than $2m - 1$ where two values can be removed. Green: Window which when calculated covers the entire straight piece.

Note that the examples, given in this section, only cover the straight line pieces skipping, but they also apply to the motif skipping. It might however be easier to see how it works with straight lines.

In Figure 5.4, you can see part of a time series with a straight line inside (blue). Of that line, only one window position needs to be kept to make sure the best correlation can still be found if it exists anywhere in this line piece. So, we keep the last m (window size) values of the piece (seen in green). We can, however, not remove the rest of the piece entirely. This is because window positions like $[10, 10, 9]$ can still result in the best correlation. This is why $m - 1$ values at the start of the piece need to be kept. As a result of this, only for each section larger than $2m - 1$ can values be removed. Although, the positions saved for each of these sections are $m + r - 1$. r being the number of consecutively removed elements.

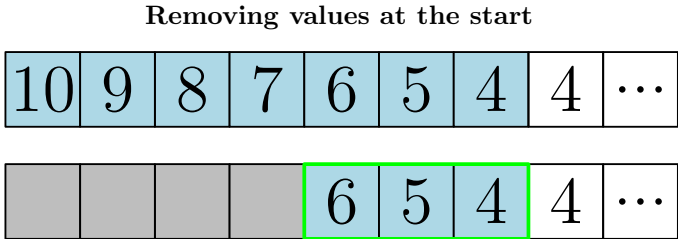

Figure 5.5: Blue: Straight piece longer than m where values can be removed. Green: Window which when calculated covers the entire straight piece.

If a section touches the beginning or end of the time series, more positions can be saved, since those $m - 1$ values are not needed. See Figure 5.5 and 5.6. Depending on the size of the piece, all but m window positions can be skipped.

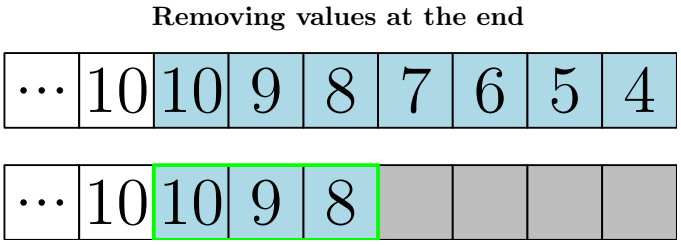

Figure 5.6: Blue: Straight piece longer than m where values can be removed. Green: Window which when calculated covers the entire straight piece.

5.4 Straight line skipping algorithm

Now that we know which positions to skip and how to mark them, we can provide an algorithm for the line skipping approach, explained in Section 5.1. Algorithm 11 scans over the time series T, keeping track of the current slope. If the slope ends, the algorithm checks whether the slope touched the start or end. If the slope touches the start of the time series, all but the last m values inside the slope are set to NaN. If the slope touches the end, all but the first m values inside are set to NaN. Finally, if the slope touches neither, all but the first $m-1$ and last m values inside are set to NaN.

Algorithm 11: MarkStraightLinesSkippable(T, m)
T: Time series, m: Query or window size

```
s_start ← 0
s_end ← 1 // inclusive
Δ ← T[0] − T[1] // angle of current slope

for all i ∈ (1 .. |T| − 1) do
  // difference between current and next value, ∞ for final round
  δ ← if (i = |T| − 1) ∞ else T[i] − T[i + 1]

  if δ = Δ then
    // section continues
    s_end ← i + i
  else
    // section ends, check if this section was long enough to cut something out
    size ← s_end − s_start
    if size ≥ m then

      // if slope touches end of time series, set end to NaN, else set middle of slope to NaN
      if s_end = |T| − 1 then
        for all j ∈ (s_start + m .. s_end) do
          T[j] ← NaN
        end for
      else
        for all j ∈ (s_start + m − 1 .. s_end − m) do
          T[j] ← NaN
        end for
      end if

      // if slope touches start of time series, set start to NaN
      if s_start = 0 then
        for all j ∈ (s_start .. s_start + m − 2) do
          T[j] ← NaN
        end for
      end if
    end if

    // start new slope
    s_start ← i
    s_end ← i + 1
    Δ ← δ
  end if
end for
```

5.5 Motif skipping algorithm

Applying the marking and skipping of positions in the time series using motif detection (described in Section 5.2) also requires a bit more explanation. Algorithm 12 shows how we applied it in practice.

The algorithm takes, aside from the time series T and the window size m, also a Matrix Profile precision p. This precision is the value the Matrix Profile needs to be above for the corresponding section to be marked as a motif, i.e. a part that is repeated elsewhere in the series.

When run, the algorithm first uses the normal Matrix Profile [13] algorithm modified to give Pearson correlation on the supplied time series. After collecting motif instances (subsequences in the MP with values all larger than p), UnionFind (Algorithms 13 and 14) is used to find clusters of motif instances that point at each other. Often, motif instances point at multiple other instances. Hence, only the one it points at most is recognized as a 'link' of a cluster.

After forming the clusters of motif instances (each cluster representing a motif), all but one instance of each motif need to be partially set to NaN. We make sure that if a motif instance touches the beginning or end of the time series it has the priority of being partially set to NaN, as that saves the most window placements. One notable difference from the straight line skipping algorithm is that for every piece that is marked to be skipped, only $m - 1$ values need to be kept at both ends of the piece (unless it is situated at either end of the time series). In the straight line skipping algorithm, the m values were used to represent the entire skipped section. For motif skipping, the section is represented elsewhere, in another instance of the same motif.

Algorithm 12: MarkRepeatingMotifsSkippable(T, m, p)
T: Time series, m: Query or window size,
p: precision (in terms of correlation, so at most 1.0, usually 0.9)

```
// Using [13] with MP values converted to Pearson
mpp, indices ← MatrixProfilePearson(T, m, MAX)
motifInstances ← All non-overlapping, as long as possible subsections of mpp that are entirely ≥ p

Filter motifInstances to only keep the ones that are at least of size 2m − 1 or, if they touch the start or end, at least of size m.

// Use UnionFind to find clusters of motif instances
clusterIndices ← Array of size |motifInstances| filled with −1
for all i ∈ (0 .. |motifInstances| − 1) do
  // First collect the indices the motif indices are pointing at using the indices array
  pointingAt ← motifInstances[i].map(it ⟶ indices[it])

  // Motif instances often point at multiple others. We try to find the other instance it points at most
  bestJ, bestCount ← −1
  for all j ∈ (0 .. |motifInstances| − 1) do
    // We're only interested in comparing motifs with others of same size
    if i = j then continue
    if |motifInstances[i]| ≠ |motifInstances[j]| then continue

    count ← count how many of pointingAt are in motifInstances[j]
    if count > bestCount then
      bestJ ← j
      bestCount ← count
    end if
  end for
  // Register the link between the current index and the one it points at most in clusterIndices using UnionFind
  if bestJ ≠ −1 then Union(clusterIndices, i, bestJ) // Algorithm 13
end for

motifs ← List of motifs (motif = list of motif instances) based on their index matching in clusterIndices[index]
Sort each motif in motifs such that the motif instances touching the start or back of time series are not first // makes sure the most highest number of values are removed

// Remove all but one occurrence of motif from the time series
for all motif ∈ motifs do
  for all i ∈ (1 .. |motifs| − 1) do
    // Take out parts of motif based on its position in T
    if motif[i].last() = |T| − 1 then
      Set all but first m − 1 of motif to NaN in T.
    else
      Set all but first m − 1 and last m − 1 of motif to to NaN in T .
    end if
    if motif[i].first() = 0 then
      Set all but last m − 1 of motif to NaN in T.
    end if
  end for
end for
```

Algorithm 13: Union(P, x, y)
P: Parent integer array (initially filled with -1), x: integer to match up with y, y: integer to match up with x

x must be different from y
$xSet \leftarrow$Find(P, x) // Algorithm 14
$ySet \leftarrow$Find(P, y) // Algorithm 14

$P[xSet] \leftarrow ySet$

Algorithm 14: Find(P, i)
P: Parent integer array (initially filled with -1), i: integer for which to find the cluster
returns: the index of the cluster i belongs to

if $P[i] = i \mid P[i] = -1$ **then**
  return i
else
  return Find(P, $P[i]$)
end if

5.6 Simplification

As can be seen in the experiments (Sections 6.3.1 and 6.3.2), both the straight line skipping (Section 5.1) and motif skipping (Section 5.2) could be more effective in practice, especially for larger window sizes. This is likely because real data is more often highly volatile and thus long straight pieces and identical motifs are uncommon.

One possible way to tackle this is by simplifying the data before searching for repeating patterns to skip and searching for lagged correlations. Note that the result of this is not exact anymore; it becomes an approximation of the best lagged correlation. There are many simplification algorithms that are applicable here, including the Ramer-Douglas-Peucker [10, 3] algorithm. This algorithm works by removing points from the graph while the resulting graph is still similar enough (the Hausdorff distance compared to the original is smaller than a certain ε). An example of the effect of this algorithm can be seen in Figure 5.7.

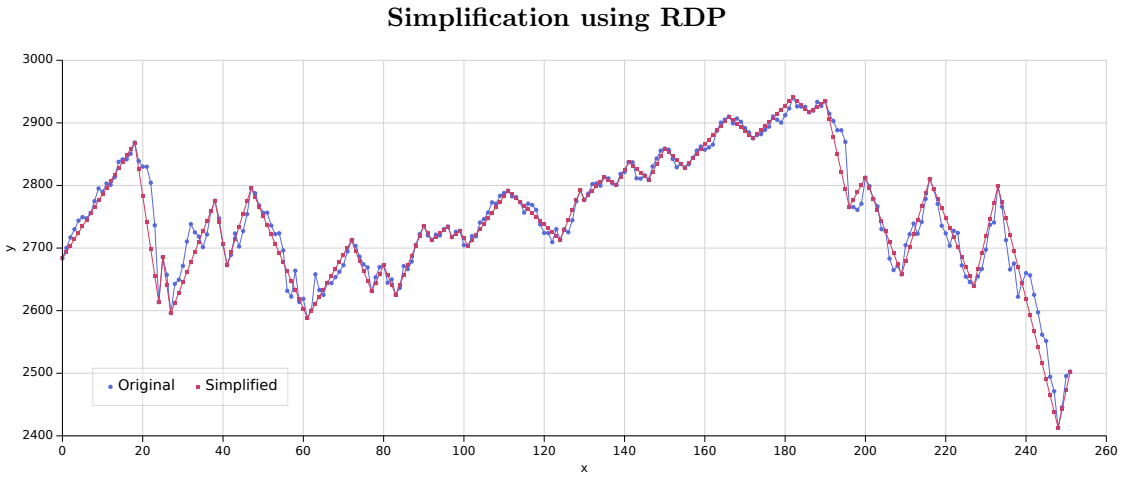

Figure 5.7: Blue: Original time series. Red: Ramer-Douglas-Peucker simplified time series.

This algorithm is both useful for the straight line skipping, since straight lines are more common

by removing points, and also for motif skipping, since simplifying the graph means there is less noise and thus the Matrix Profile will produce clearer results.

Results of applying these simplifications can be found in the experiments (Sections 6.3.3 and 6.3.4).

5.7 Applying skipping to STAMP-3TS-Pearson

To use the window skipping in the context of this thesis, they have to be able to be applied to the STAMP-3TS-Pearson algorithms from the previous chapter. This does come with some hurdles, as the algorithm is already very optimized.

Firstly, when calculating the lagged ternary correlation, the skipping of windows cannot be applied to time series T_A, since that time series is used in its entirety in the Fast Fourier Transform part of the SlidingDotProducts algorithm [13]. The negative impact of this can however be minimized by ordering the three time series on the amount of NaN values contained in them. The time series with the most values removed becomes T_C, the next T_B and the one with the fewest NaN's will have to be replaced by its original version (with all values intact) and become T_A.

The (aggregated) sliding means and -standard deviations (Section 4.1.2.1) also suffer in performance from having pieces of the time series set to NaN since they use a sliding window to save calculations. If they encounter a NaN value, they have to start over and calculate the mean and standard deviation using all the values in the window again (instead of simply updating the sliding mean and standard deviation using a single new one). Depending on how many pieces are set to NaN, the decision can be made to give the sliding means and -standard deviations the unmodified time series, as that will result in a shorter running time. In the experiments, this decision is always made.

The same holds for the sliding dot product optimization (Section 4.3.2.1), which also profits from the calculations of the previous window to calculate the current. For this, in the experiments, the decision is also always made to provide an unmodified time series.

Finally, we need to look at the combination of the lag bound optimization (Section 4.3.4) and window skipping. This fix is only relevant for the straight-line skipping. When marking the middle of a straight piece NaN, the straight line skipping algorithm usually leaves the last m values to represent the correlation values for the entire straight piece. However, since lag bound can cut off the time series at any point, it might be the case that these last m values are (partially) cut off from the time series, meaning that a part of the time series is left unprocessed. A simple fix for this is to add a single value from the original time series to the first $m - 1$ values already present at the start of the section. This makes sure that the correlation value of the entire straight piece can still be calculated.

A similar fix is needed for the motif skipping. If a motif instance representing a skipped one is cut off by the lag bound, one of the motif instances inside the lag bound must be restored for the result to be valid.

Chapter 6

Experiments

To demonstrate the effectiveness of the proposed solutions to the two research sub-problems, we provide several experiments.

In this chapter, we first go over the experimental setup (Section 6.1), which includes the technical details and tools used to implement the algorithms and run the experiments. Next we give a brief summary of the results in Section 6.1.4. The experiments regarding the first problem can be found in Section 6.2. This includes the experiments themselves, as well as an interpretation of the results. The same can be found for the second problem in Section 6.3.

6.1 Experimental setup

To explain how the experiments are executed, in this section we go over the specifications of the device used to run the experiments (Section 6.1.1) and how the data for the experiments is gathered (Section 6.1.2). Next, we go over how the experiments and algorithms are implemented and the tools used (Section 6.1.3).

6.1.1 Computer

All experimental tests are run on a Lenovo Thinkpad W541 with an Intel Core i7-4710MQ CPU (2.5 GHz x 8), 22 GiB of DDR3 RAM at 33 MHz (+8 GiB of swap memory on the SSD), Samsung 860 EVO SSD (3B6Q), running Arch Linux (kernel version 5.13.13.arch1-1).

6.1.2 Dataset

The dataset used to run the experiments consists of stock data in the year 2018. We use two differently pre-processed versions of the dataset.

The first consists of the closing prices of all stocks aggregated by average per day. The days that did not have much data (weekends, holidays) were dropped. After this, all the stocks that did not have an average value for each of these days were dropped too. Stocks that looked erroneous when plotted were dropped manually as well. The end result was a relatively small data set of 80 time series of size 252. However, while small, it contains realistic data (no interpolation) and it proves useful for testing the algorithms on a small scale and with same-length time series.

The second pre-processed version of the dataset aims to provide longer time series and of different lengths. It contains all days of the year and the data is aggregated by average closing price per hour. Missing values in the time series are linearly interpolated. This leads to 2182 time series of sizes ranging between 674 and 8760 (although, most are 8000+).

6.1.3 Implementation

The first part of the implementation is about reading and (pre-)processing the dataset. Apache Spark[1] was chosen to help doing this job. This library was chosen because it enables us to parse and manage large amounts of data easily, while also giving us the ability to distribute the work evenly across multiple threads, or even, multiple computers. The implementation of the algorithms in the thesis is not dependent on Spark itself. It can be run in any JVM (Java Virtual Machine) environment. We, however, use Spark for the experiments to parse the data, create all the possible groups and divide the work evenly in 8 threads.

The programming language chosen for Spark and the rest of the implementation was Jetbrains' Kotlin[2]. It can run on the JVM, similar to Java and Scala, and can thus interoperate freely with Kotlin, Scala, and Java libraries. The language was chosen for its conciseness, readability, features, and decreased amount of boilerplate code compared to more traditional languages, like Java.

Because Apache Spark at the time of writing only officially supports Scala, Java, R, and Python, the unofficial Kotlin Spark API by Jetbrains[3] was used to apply more of the advantages of the language to Spark. This includes, among other things, better support for lambda functions, overloaded operators, and support for Kotlin's data classes to represent types in Datasets. During this research, many contributions were made to this open-source project, helping it to a 1.0 release[4].

Many of the algorithms proposed in this research involve vector- or array calculations. For this purpose, the 64-bit floating-point arrays of Jetbrains' Viktor[5] Kotlin library were used. As described by the library authors, the library brings a couple of Numpy[6] features to Kotlin. It provides Intel SIMD acceleration for some operations, like the dot-product and other element-wise calculations, as well as some syntactical features.

For the sliding dot product part of the algorithm, a couple of FFT (Fast Fourier Transform libraries) were tested, many using native code, using a JNI (Java Native Interface), to speed up calculations. However, as most of the calculations involve arrays that are not that large, it turned out that keeping the arrays within the JVM was faster than copying them over and back from native arrays. The Java library JTransforms[7] was chosen to handle the FFT calculations because it is fast, open-source, and purely Java, meaning there is no speed loss for using a JNI to copy arrays to native ones.

Most plots in the paper are created using the Charts.kt[8] library using their provided student license. Other considered libraries were Plotly.kt[9] and Jetbrains' Lets-Plot for Kotlin[10]. Charts.kt was chosen for its more Kotlin-esque approach and descriptive, strongly-typed notation. The other two libraries are essentially Python ports and are promoted to be used in Jupyter notebooks. This results in code that prefers runtime failure above compilation, making debugging a lot harder. Charts.kt is built upon Data2Viz[11], an open-source multiplatform Kotlin data visualisation toolbox. During the research, contributions were also done, bringing SVG rendering capabilities to the project.

All relevant implementations, algorithms, and source code are made available at `https://github.com/Jolanrensen/EfficientlyDiscoveringTernaryLaggedCorrelations`.

6.1.4 Summary of results

First problem The experiments show that the STAMP-Pearson-3TS algorithm is faster than a naive algorithm (described in Section 6.2.1), but not for small window sizes. The length of

[1] `https://spark.apache.org/`
[2] `https://kotlinlang.org/`
[3] `https://github.com/JetBrains/kotlin-spark-api`
[4] `https://blog.jetbrains.com/kotlin/2021/07/kotlin-api-for-apache-spark-1-0-released/`
[5] `https://github.com/JetBrains-Research/viktor`
[6] `https://numpy.org/`
[7] `https://sites.google.com/site/piotrwendykier/software/jtransforms`
[8] `https://charts-kt.io/`
[9] `https://github.com/mipt-npm/plotly.kt`
[10] `https://github.com/JetBrains/lets-plot-kotlin`
[11] `https://github.com/data2viz/data2viz`

the time series influences the computation time similarly for both STAMP-Pearson-3TS and the naive algorithm. Lag bound has a very large impact on the running time (described in Section 6.2.2). The tighter it is, the shorter the running time. The running time of the sliding means and -standard deviation algorithms is shorter than the one with aggregation (described in Section 6.2.3), but this is likely a necessary sacrifice. Both versions are similarly affected by time series size and similarly unaffected by the window size.

Second problem The experiments show that the straight line skipping method (from Section 5.1) cannot skip a large number of window positions in all types of data (described in Section 6.3.1). This difference is clearly visible between the first dataset (where almost no window positions can be skipped) and the second dataset (where quite a few can be skipped). The number of window positions that can be skipped decreases as the window size increases. motif skipping (from Section 5.2) is slower than straight line skipping (as described in Section 6.3.2). It works best using a high precision value and it has a sweet spot for a window size in terms of how many window positions can be skipped. After the sweet spot, the number of window positions that can be skipped also decreases as the window size increases. RamerDouglasPeucker simplification (from Section 5.6) works to make straight line skipping faster in almost all cases (as described in Section 6.3.3). However, it does not work to make motif skipping faster (as described in Section 6.3.4). The more window placements can be skipped, the faster STAMP-Pearson-3TS runs (as described in Section 6.3.5). For the first dataset, where little to no window placements can be skipped using straight line window skipping, STAMP-Pearson-3TS without skipping is almost always the fastest option. For the second dataset, where more windows can be skipped, STAMP-Pearson-3TS is the fastest with skipping and simplification for larger window sizes. For smaller window sizes the native algorithm is the fastest.

6.2 First problem

The first problem, worked out in Chapter 4, aims to find an efficient method to find the strength of ternary lagged correlation for a group of time series. The method found comes in the form of the optimized STAMP-Pearson-3TS algorithm. To demonstrate the efficiency and strengths of this algorithm, we provide a couple of tests: The first tests compare the optimized STAMP-Pearson-3TS algorithm to a naive approach (Section 6.2.1). Next, we show the effect of introducing lag bound to the algorithm in Section 6.2.2, the running time of the Siding Means and -Standard Deviations algorithm with and without aggregation in Section 6.2.3.

6.2.1 Optimized versus naive version

To demonstrate the efficiency of the STAMP-Pearson-3TS algorithms, we measure how fast it can calculate the ternary lagged correlation compared to a naive approach.

The naive algorithm is constructed as follows: The algorithm consists of a triple for-loop to calculate all combinations of window positions to find the best Pearson correlation. It also has lag bound built-in, since this is not just an optimization, but also an analytical feature. Of course, STAMP-Pearson-3TS is very memory efficient. This includes reusing (Viktor) arrays and being largely functional, avoiding the creation of JVM objects. This philosophy is also adhered to in the naive version. Combining all these properties, comparing the naive- to the STAMP-Pearson-3TS algorithm now only comes down to the MASS-Pearson implementation and optimizations put in place, which are what makes STAMP-Pearson-3TS unique.

For all experiments, STAMP-Pearson-3TS uses all optimizations described in Section 4.3, except for the pre-calculation of the sliding means and -standard deviations of the time series (described at the bottom of Section 4.3.1.1), as this moves some essential calculations outside the algorithm, giving it an unfair advantage.

For the first experiment, we take 30 time series from the first dataset and we compare the running time for calculating the lagged ternary correlation for all 4060 possible groups of three.

The lag bound is set to 10 for both methods and the length of each time series is the same (252). We are searching for the highest correlation (instead of the lowest), but this has no effect on the running time. The window sizes are varied. The results can be seen in Figure 6.1.

Running Time per Window Size, STAMP-Pearson-3TS versus Naive, First Dataset

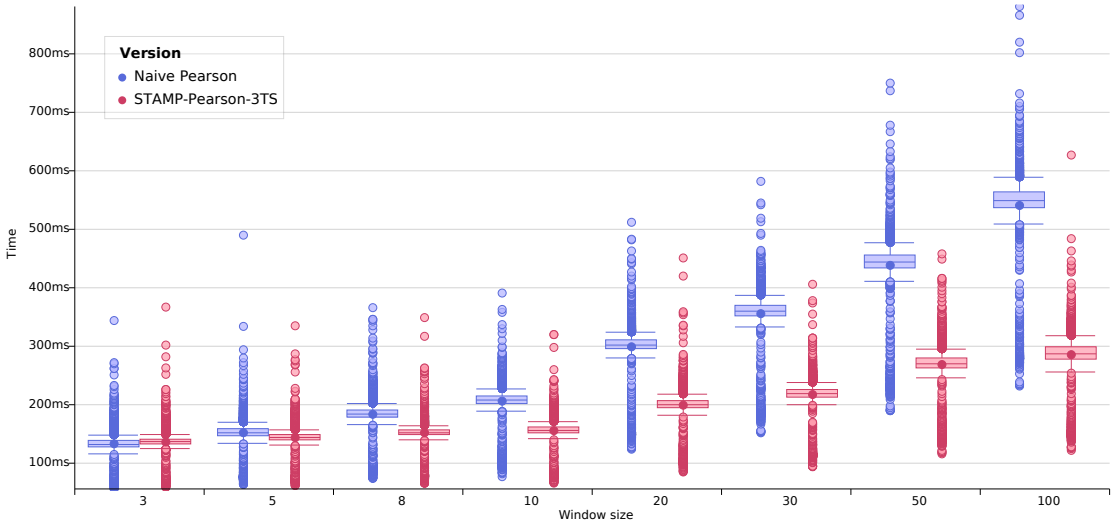

Figure 6.1: Each data point is the time of the correlation calculation of a single group of 3 time series.

For the second experiment, we take 7 time series from the second dataset. These specific time series vary in length from 8640 to 8735, forming 35 unique groups. The reason fewer time series are used is because of their long length, making the calculations take considerably longer. We again have a set lag bound of 10, are looking for the highest correlation, and vary the window size. See Figure 6.2.

Running Time per Window Size, STAMP-Pearson-3TS versus Naive, Second Dataset

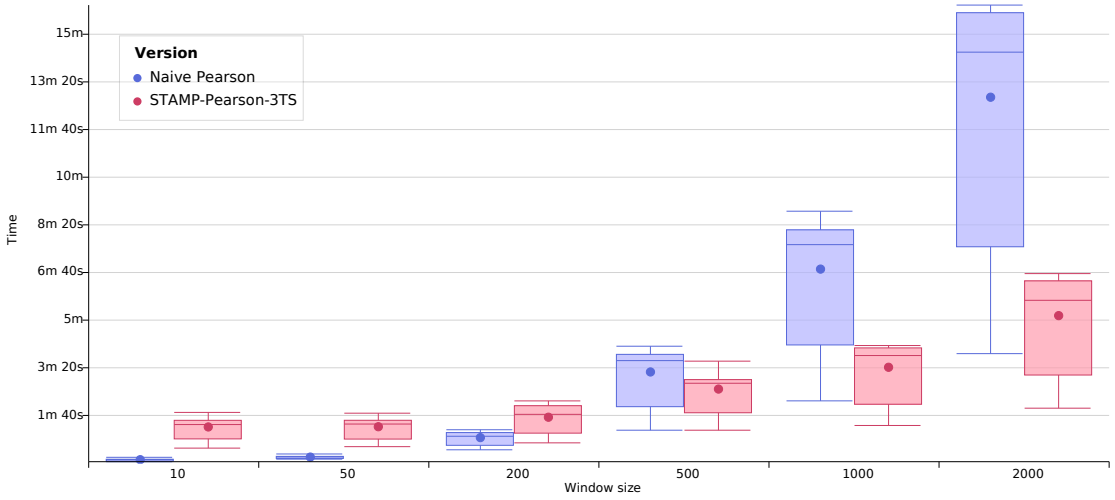

Figure 6.2: Each data point is the time of the correlation calculation of a single group of 3 time series.

The first two experiments (Figure 6.1 and 6.2) show that STAMP-Pearson-3TS is not always faster than the naive version. When the data consists of longer time series (such as in the second dataset), using a relatively small window size, the naive Pearson version performs faster on average. When the window size becomes larger, relative to the time series size, STAMP-Pearson-3TS seems to perform better. This can most likely be attributed to the amount of time it takes to reserve and cache all the sliding calculations. When the window size is large enough, however, the time saved by the sliding calculations outweighs it. So, STAMP-Pearson-3TS is less affected by window size than the naive version, but there is still a slight impact on performance. This makes sense since the sliding calculations need to be initialized. This initialization only happens once, but is still dependent on the window size, making the algorithm a slight bit slower.

The third experiment also uses the second dataset, but this time we are interested in how the time series length affects the running time. We again take 7 time series, but this time, we cut them off at different lengths. The window size is fixed at 500, the lag bound at 10, and we are again looking for the highest correlations. See Figure 6.3.

Running Time per Time Series Size, STAMP-Pearson-3TS versus Naive, Second Dataset

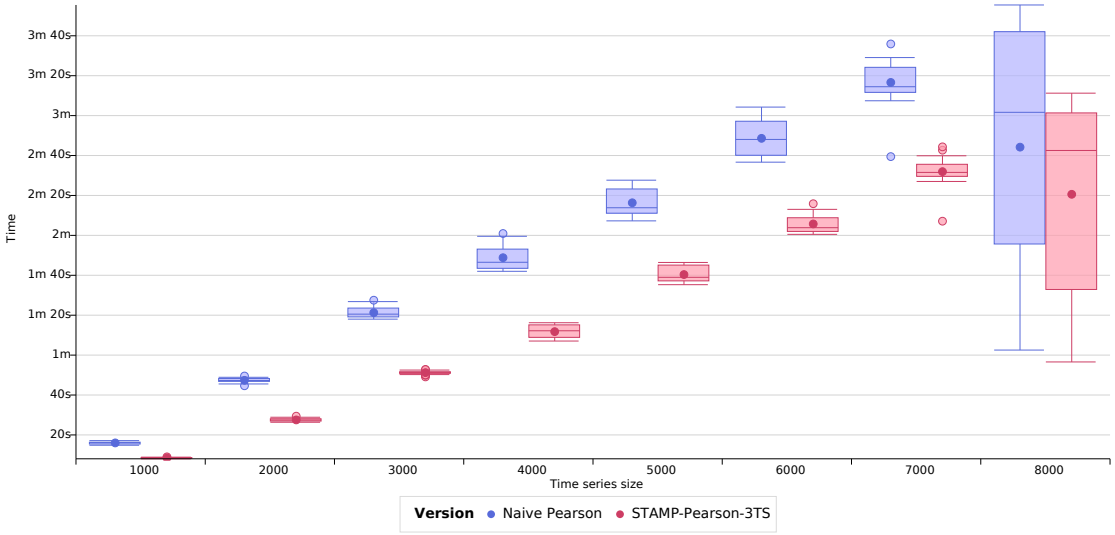

Figure 6.3: Each data point is the time of the correlation calculation of a single group of 3 time series.

Looking at the result we can see a comparison between the effect of time series length on the Naive version and on STAMP-Pearson-3TS. Since the window size was picked to be 500 and the second data set is used, STAMP-Pearson-3TS performs better, but this is likely the only reason. The best takeaway from this result is that STAMP-Pearson-3TS and the naive version are equally affected by the time series size. This result is to be expected since in both cases, the entire time series needs to be traversed many times in order to find the best correlation.

6.2.2 Effectiveness of lag bound in STAMP-Pearson-3TS

We want to show the effectiveness of adding a lag bound to the lagged ternary correlation calculations. For this, we use the first dataset and take again 30 time series of size 252 (resulting in 4060 groups). We use a window size of 10 and we vary the lag bound while looking for the highest correlation. The results are visible in Figure 6.4.

The results are as expected. The tighter the lag bound, the fewer combinations of window positions are considered, resulting in a much faster time. The challenge is to find the sweet spot

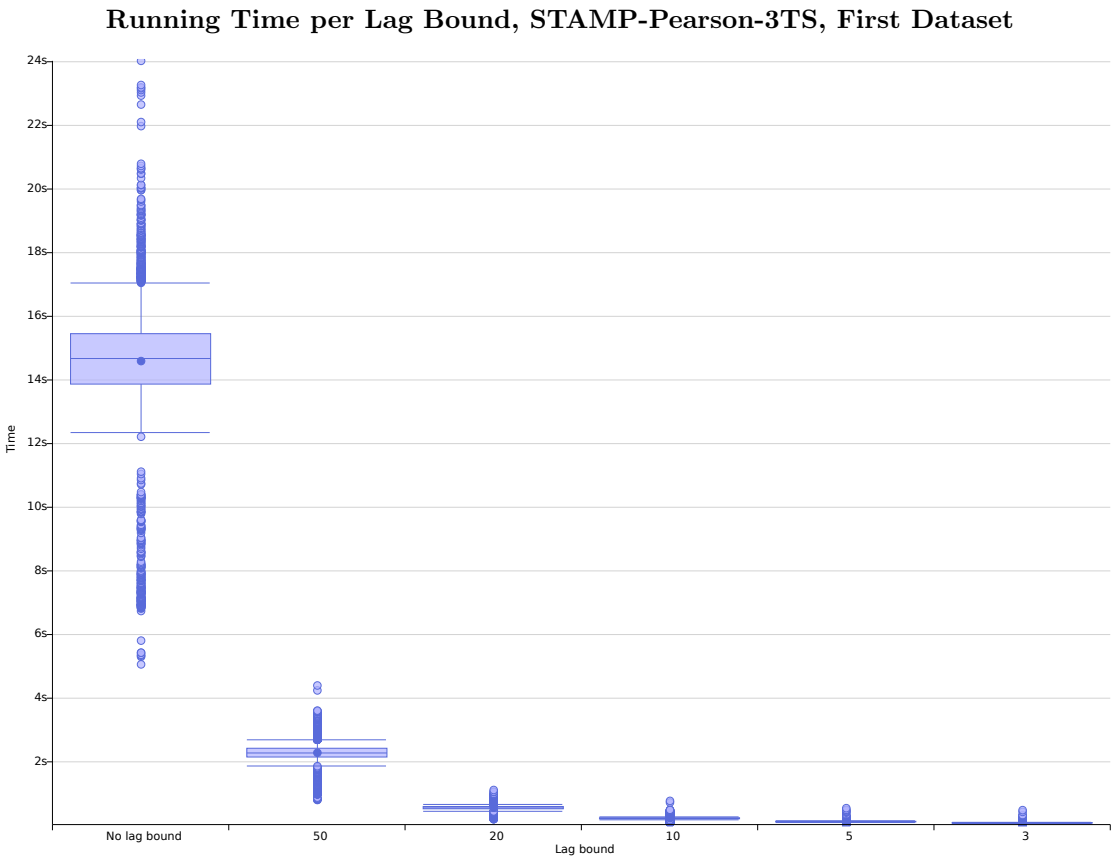

Figure 6.4: Each data point is the time of the correlation calculation of a single group of 3 time series.

for the size of the lag bound where the data analyst would still find an interesting enough lagged correlation.

6.2.3 Sliding Means and -Standard Deviation with Aggregation

The sliding means and sliding standard deviation calculations needed to be adapted to allow for some aggregation for STAMP-Pearson-3TS to work correctly. This is explained in Section 4.2.2.2. While we attempted to make it as efficient as possible, we expect it to still be slower than the original variant, simply due to the larger amount of calculations.

To test the speed, we calculated the aggregated- and non aggregated sliding means and -standard deviations for 7 random time series from the second dataset. This we did for each combination of 6 different window sizes and 8 different time series sizes from the second data set, resulting in Figures 6.5 and 6.6.

Looking at the results, we can spot several things. First of all, the bulk of tests show that the aggregated version is almost always slower than the normal one. It is not by much, but it will add up when it is run many times. It is however a necessary sacrifice since aggregation needs to occur for correlation in groups of three. What is interesting to see is that the window size does not seem to significantly impact the running time. This indicates that the sliding behavior works to save calculations both for the aggregated and non-aggregated versions of the algorithm. Finally, the time series size impacts the running time as expected.

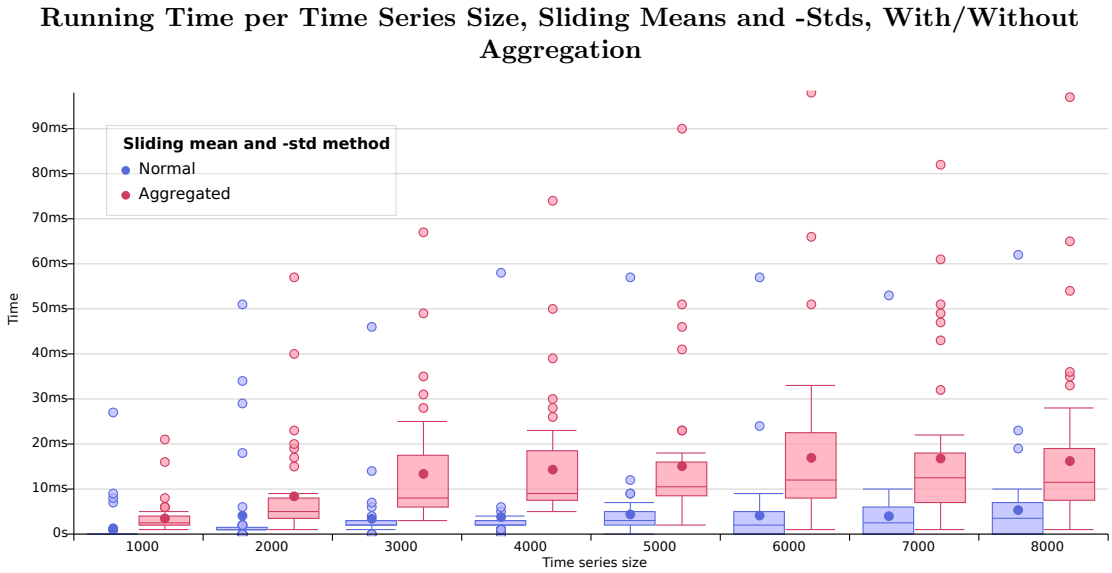

Figure 6.5: Each data point represents the time to calculate the sliding means and -stds for one time series.

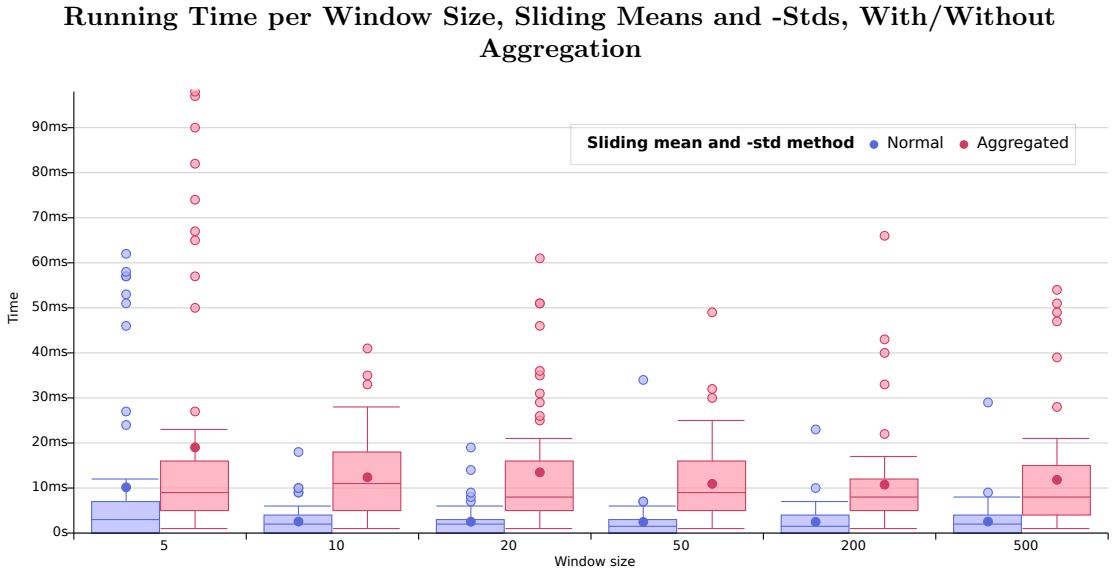

Figure 6.6: Each data point represents the time to calculate the sliding means and -stds for one time series.

6.3 Second problem

The second research sub-question aims to find correlations even faster by applying approximation- and window skipping techniques. We first go over testing the straight line skipping technique for both data sets in Section 6.3.1. Next, we do the same for the motif skipping technique in Section 6.3.2. Afterward, we show the effectiveness of combining simplification with both techniques in Sections 6.3.3 and 6.3.4. Finally, we show the effect the amount of skipped windows has on the running time of STAMP-Pearson-3TS in Section 6.3.5.

6.3.1 Straight line skipping per window size

We want to test the effectiveness of using the straight line skipping technique, described in Section 5.1. We do this by calculating how many window positions can be skipped, after running the algorithm. This value is purely based on the number of NaN values and the set window size. Lag bound is not considered.

We first test using all the time series in the first dataset and using various window sizes. See Figure 6.7.

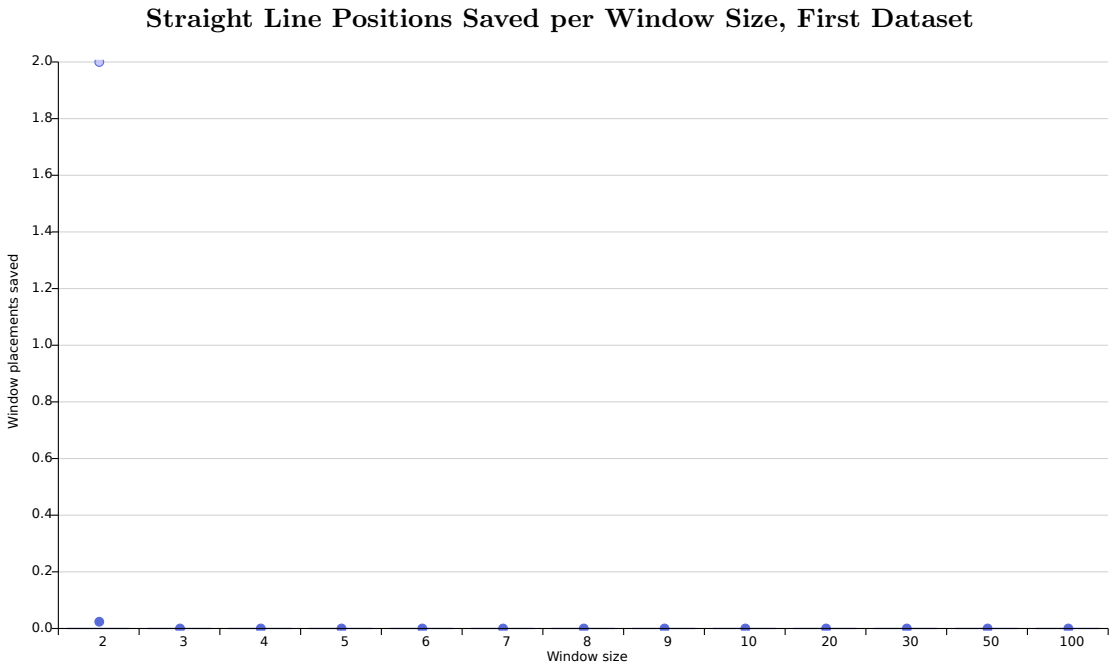

Figure 6.7: Each data point represents the number of window placements saved for a single time series.

Next, we test using 100 time series from the second dataset using slightly more window sizes, since the time series in the second dataset are longer. See Figure 6.8.

As can clearly be seen, the algorithm is almost completely ineffective on the first dataset. For the second dataset, which has longer time series and of various lengths and contains linear interpolation, it works better. The aggregations of the time series could be an explanation for this difference. Stock data is usually quite fluctuating, especially when seen across multiple days. The second data set is aggregated across hours, where the missing values are linearly interpolated. This means more long straight pieces can be expected to exist. What is noteworthy, however, is the decrease of window placements saved when the window size increases. This is of course explicable, again, by the volatility of the time series. The larger the window is, the less chance it has of enveloping a piece of a time series that is entirely straight. This might differ when using another dataset, but we suspect that it is still very common. Although for some data sets a lot of window positions can be skipped when the window size is small, in practice, larger window sizes tend to give more useful information, lowering the number of window positions that can be skipped.

6.3.2 Motif skipping per window size

We also want to test the effectiveness of using the motif technique to skip windows. This technique was described in Section 5.2.

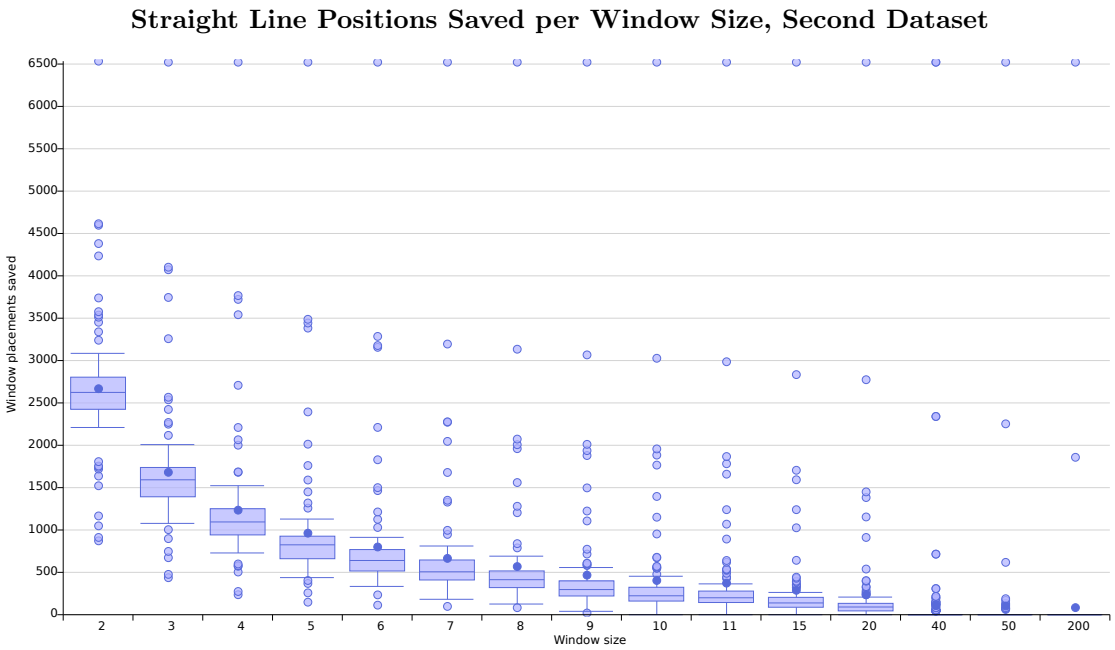

Figure 6.8: Each data point represents the number of window placements saved for a single time series.

First, we test using all the time series in the first dataset, again using various window sizes. However, since the motif skipping method can have different precision values, we decided to test with 4 different Matrix Profile precisions and only show the average amount of window positions saved.

See Figure 6.9.

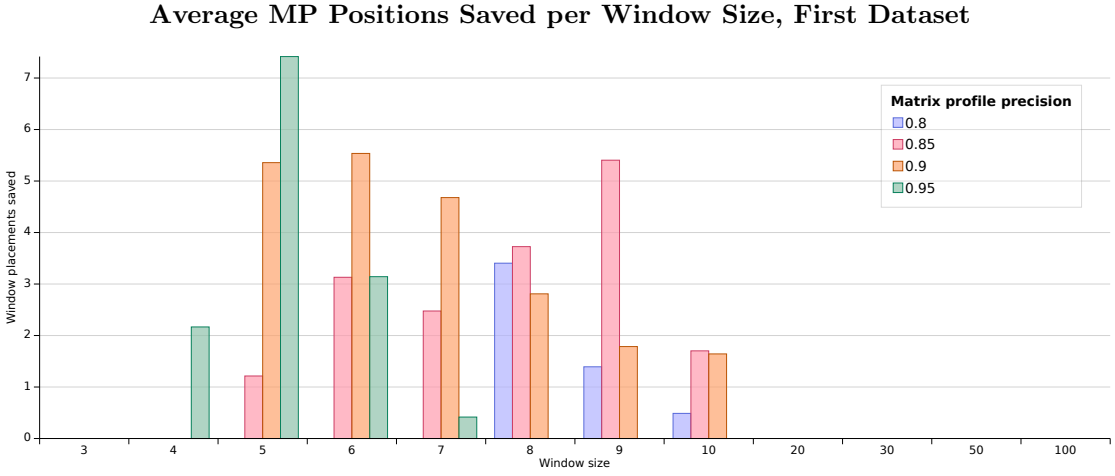

Figure 6.9: Each data point represents the average number of window placements saved for all time series.

Secondly, we test using 30 time series from the second dataset. The number is smaller since the calculations took a lot longer using the motif detection technique, compared to the straight line technique. See Figure 6.10.

Now, looking at the results: A big difference with the previous experiments is that the Matrix

Figure 6.10: Each data point represents the average number of window placements saved for all time series.

Profile Motif precision is another variable now. When this value is lower, a piece of the time series is qualified more quickly as appearing in a repeating motif.

What is interesting to see is that this method actually has a 'sweet spot' for a window size, dependent on the data set used. For the first dataset, this lies at a window size of 5, with a precision of 0.95, for the second dataset it lies at the same precision and a window size of 7.

The best window size is also dependent on the Matrix Profile Motif precision, tending to lie higher when the motif detection is less precise.

What is surprising is that on average fewer window placements are saved when the motif detection is less precise. It is hard to say why exactly this is happening, but it could be due to multiple motifs being detected as a single one, which decreases the number of motif instances that can be marked to be skipped. It is a convenient result though since a higher precision means that, when a motif instance is marked to be skipped, the chance is lower that mistakes are made and that sections of time series are incorrectly marked as skippable.

One last observation is that, again, not many windows are saved for large window sizes.

6.3.3 Straight line skipping with simplification

We want to show the effect simplification has on the ability to skip window positions. The simplification method, RamerDouglasPeucker, as described in Section 5.6, takes an ε value. The larger this value, the further the time series are simplified.

The first test is run using all the values in the first dataset. The same window sizes are used as before and several ε values are tested. The window placements saved are averaged across all time series. See Figure 6.11 for the result.

The second test is again run using 100 values from the second dataset, with the same window sizes as before, as well as the same ε values as in the last test. See Figure 6.12.

The simplification comes with an ε value, which represents the maximum Hausdorff distance of the simplified time series compared to the original. This means that when ε is larger, the time series is further simplified. With an ε of 0.0, the time series is unchanged.

As also could be seen in Figure 6.7, for an ε of 0.0, on average no window placements are saved using the first dataset. Although, for larger ε values, more promising results show up, displaying an overall increase in window placements saved when ε gets larger. There even appear to be a number of window positions that can be saved using larger window sizes, which is what is more common in practice. The best ε for the first dataset appears to be 1.0, providing a nice balance between windows saved, even at larger window sizes, while not over-simplifying the time series.

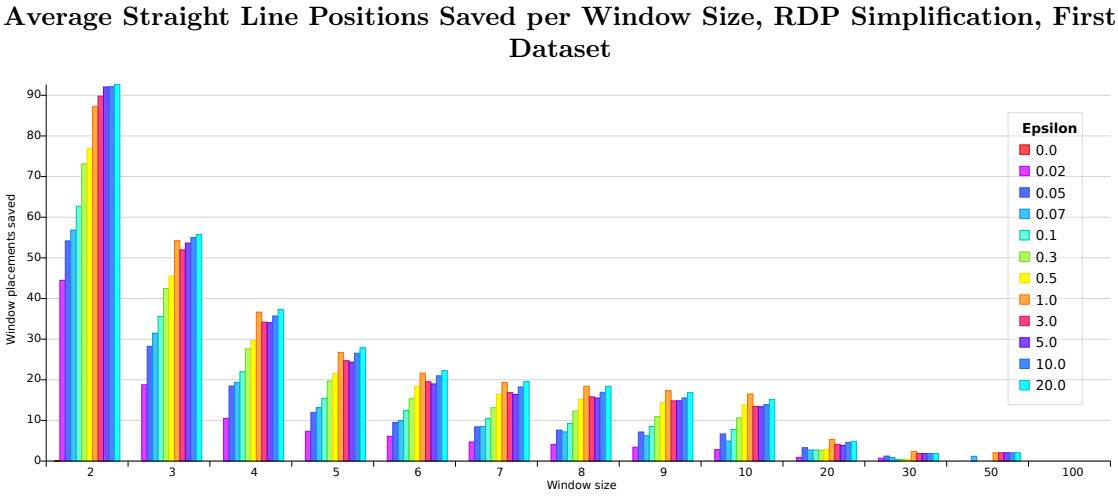

Figure 6.11: Each data point represents the average number of window placements saved for all time series.

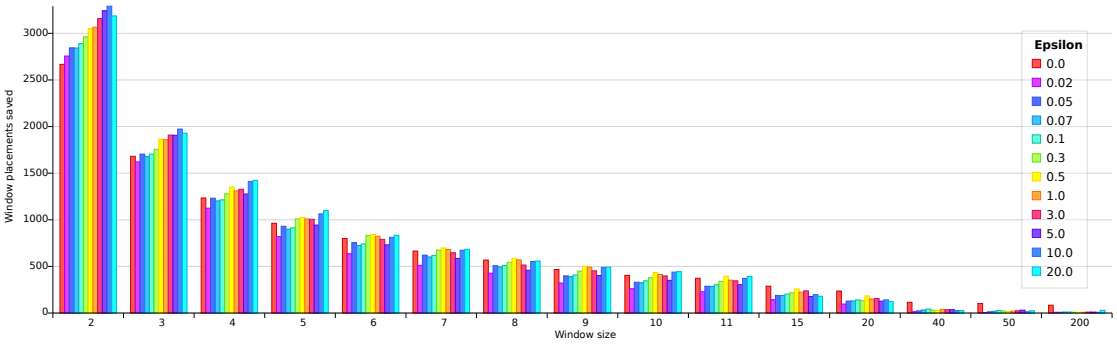

Figure 6.12: Each data point represents the average number of window placements saved for all time series.

Why 1.0 performs as well or better than 20.0 is an unclear but welcome result, since the result is closer to the original time series.

For the second dataset, a similar structure appears, showing a slight increase in window positions saved as ε gets larger, with a well-performing ε around 0.5..1.0. There is a surprising result, however, as for larger window sizes no simplification provides more saved window positions compared to with simplification. This proves that the straight line skipping method does not always benefit from this kind of simplification.

6.3.4 Motif skipping with simplification

We also want to test the effect simplification has on the ability to save window positions using the motif skipping method. The motif skipping method, however, also has a precision value that influences this ability, as shown in Section 6.3.2.

For the first test, we look again at the first dataset, using all its time series. While multiple Matrix Profile precisions are tested, we decided to only show the one which best shows the effect the simplification has on the result, so we picked the precision to be 0.9. See Figure 6.13.

The second test is similar, but it uses 30 time series from the second dataset again. The Matrix

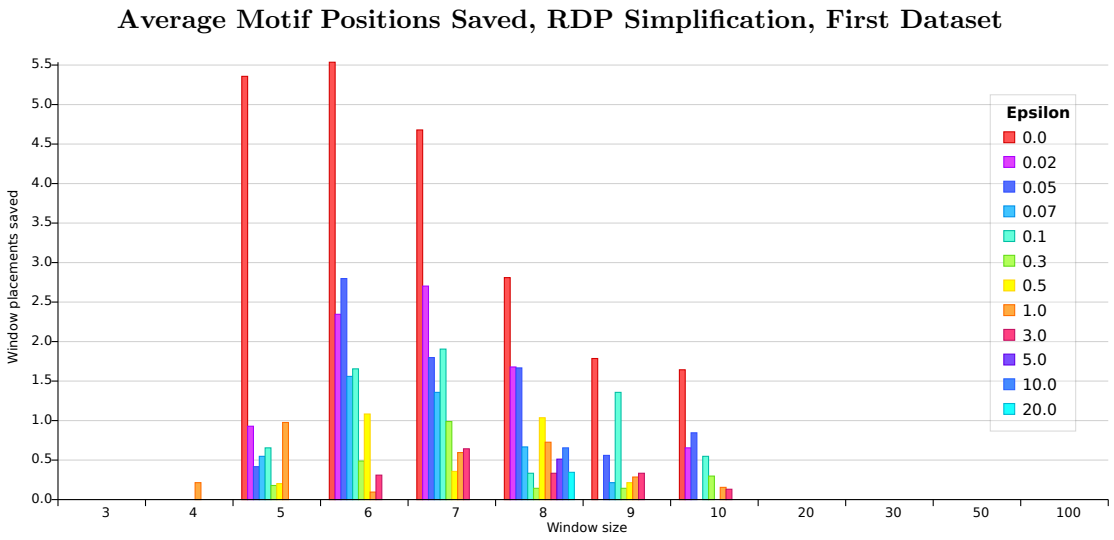

Figure 6.13: Each data point represents the average number of window placements saved for all time series. Matrix Profile precision: 0.9

Profile precision is set to 0.9 again. See Figure 6.14.

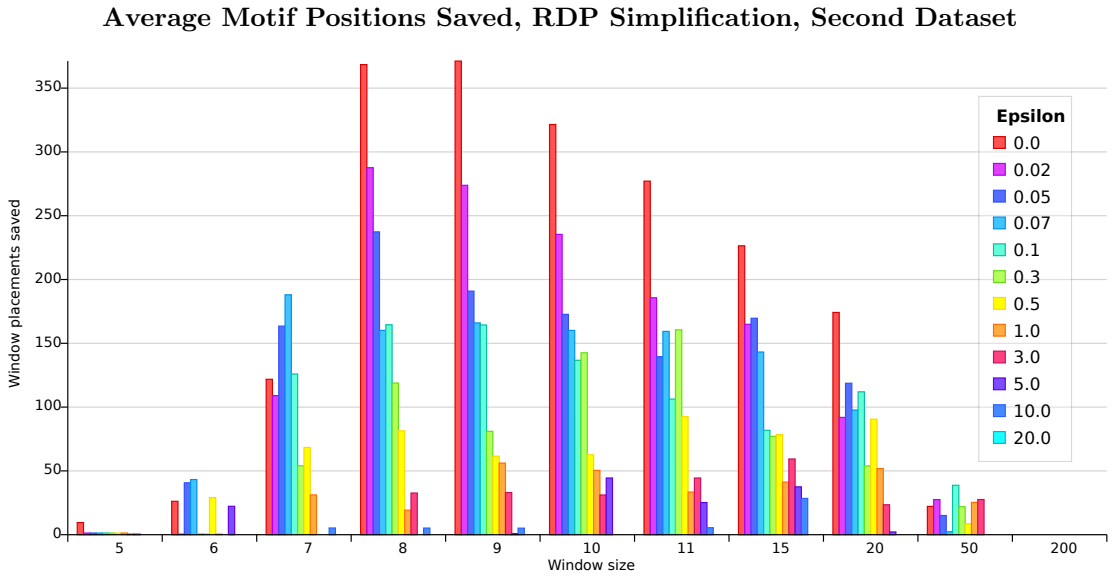

Figure 6.14: Each data point represents the average number of window placements saved for all time series. Matrix Profile precision: 0.9

It is very clear, in both plots, that having an ε of 0.0 (meaning, no simplification) has the best result almost all the time. The only exceptions are the window size of 4 for the first dataset, and for the second dataset, the window sizes 6, 7, and 50. For most of the other cases, the motif skipping method seems to perform worse as ε gets larger. This might be explained by the fact that motifs are recognized better when there is a higher level of granularity. The simplification process could also merge motif instances together with instances of other motifs, making them individually less recognizable.

6.3.5 Effectiveness of window skipping in STAMP-Pearson-3TS

Finally, we want to show the effectiveness of skipping windows in terms of STAMP-Pearson-3TS running time. To do this, for a couple of window sizes we gather groups of 3 time series where each has the same amount of skipped window positions. We use all the time series from the first data set with various ε simplification values before using the straight line skipping algorithm to save window positions. It was, however, not possible to generate a lot of data for time series where many positions were skipped, since it does not occur often. All time series have the same length, the lag bound is set to 10 and various window sizes are used. Figure 6.15 shows the STAMP-Pearson-3TS running time per window positions saved for different window sizes. The number of data points gathered can be found in Table 6.1. Keep in mind that the significance of the results decreases when the window size increases, due to the lower amount of data points, however, the overall slope of the trend line can still be deduced.

To compare the running time of STAMP-3TS with the skipping of windows to the running time without, we also provide the box plot in Figure 6.16. These results are calculated using the original STAMP-3TS algorithm, untouched by any window skipping.

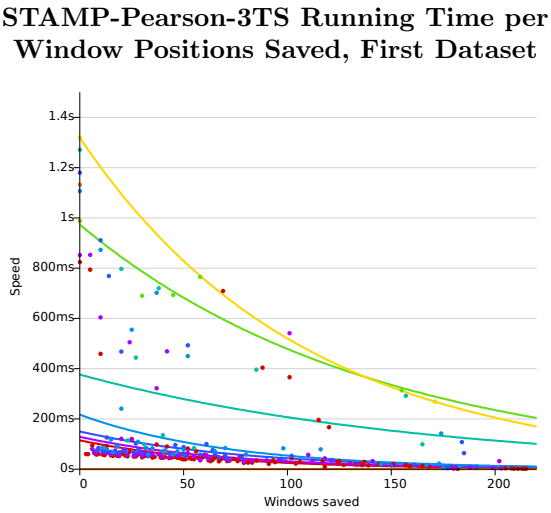

Figure 6.15: Each data point represents the running time of STAMP-Pearson-3TS with a group of 3 time series with the same number of saved window positions. Approximate exponential trend lines are visible. See Table 6.1

.

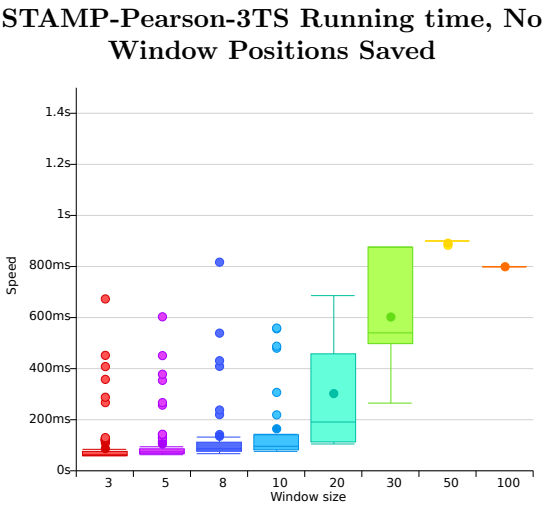

Figure 6.16: Each data point represents the running time of STAMP-Pearson-3TS with a group of 3 time series.

Window size	Amount of data points
3	114
5	68
8	45
10	31
20	11
30	5
50	2
100	1

Table 6.1: Amount of data points per window size, first dataset

Looking at Figure 6.15, we can see that, for all window sizes, it applies that STAMP-Pearson-

3TS performs quicker on average when more window positions are skipped. When only a couple of window positions are saved, the original STAMP-Pearson-3TS (as shown in Figure 6.16), in some cases, still performs better. This is likely due to the search for skippable straight pieces. Only when enough windows are skipped, the time saved outweighs this. When about 50 or more window positions can be saved, the algorithm outperforms the original, which shows that window skipping does make STAMP-Pearson-3TS faster.

What needs to be noted, however, is in the accompanying table, Table 6.1. It shows that the number of data points we are able to generate for each window size. A data point represents a group of three time series where any window position can be skipped. The table clearly shows that it becomes more and more difficult to skip any window position when the window size gets larger, which is what we also found in previous experiments. This also means that the trend lines for larger window sizes need to be taken with a grain of salt.

We can also show the window position saving in practice. To do this, we can revisit the second experiment in Section 6.2.1, where the running time per window size is measured for a naive approach (described at the Section 6.2.1), STAMP-Pearson-3TS, and now also STAMP-Pearson-3TS where window positions are saved. Similar to the other experiment, 30 time series of the first data set are picked and with a lag bound of 10, the experiments are run again. The straight line window saving method is used without and with RamerDouglasPeucker simplification with an ε of 1.0 (since that performed well in Figure 6.11). See Figure 6.17 for the results.

Running Time per Window Size, STAMP-Pearson-3TS versus Naive versus Window Skipping, First Dataset

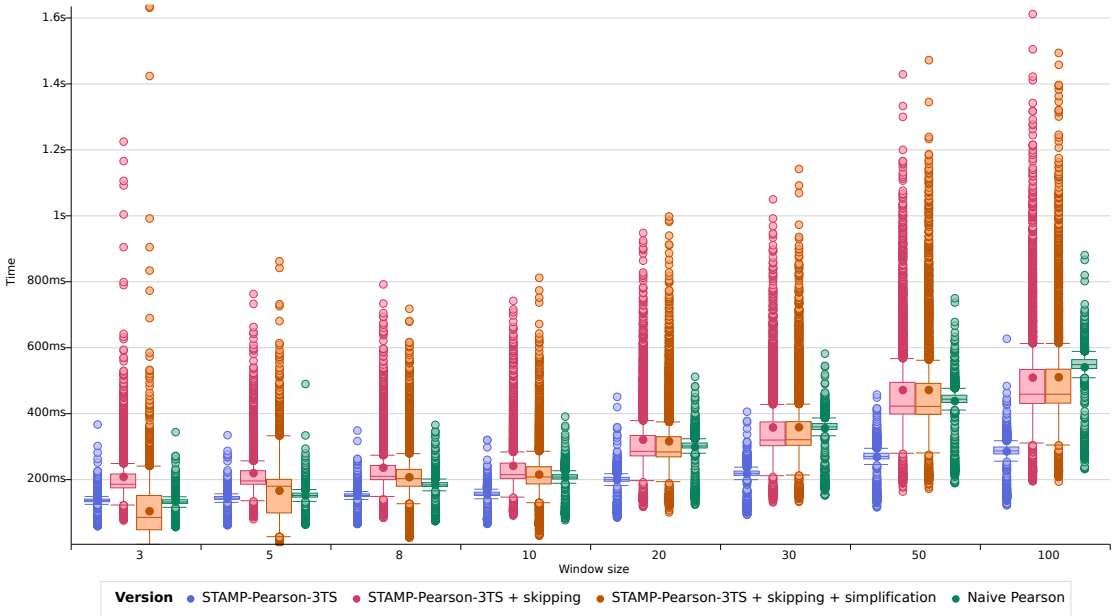

Figure 6.17: Each data point is the time of the correlation calculation of a single group of 3 time series.

We repeat this for the second data set. 7 time series of this data set are used with a lag bound of 10. The straight line window saving method is used without and with RamerDouglasPeucker simplification with an ε of 0.5 (since that performed well in Figure 6.12). The results can be seen in Figure 6.18.

Applying the window skipping method in practice shows that the naive approach still matches or beats the normal STAMP-Pearson-3TS for smaller window sizes for both data sets, however, for larger window sizes it is considerably slower.

Running Time per Window Size, STAMP-Pearson-3TS versus Naive versus Window Skipping, Second Dataset

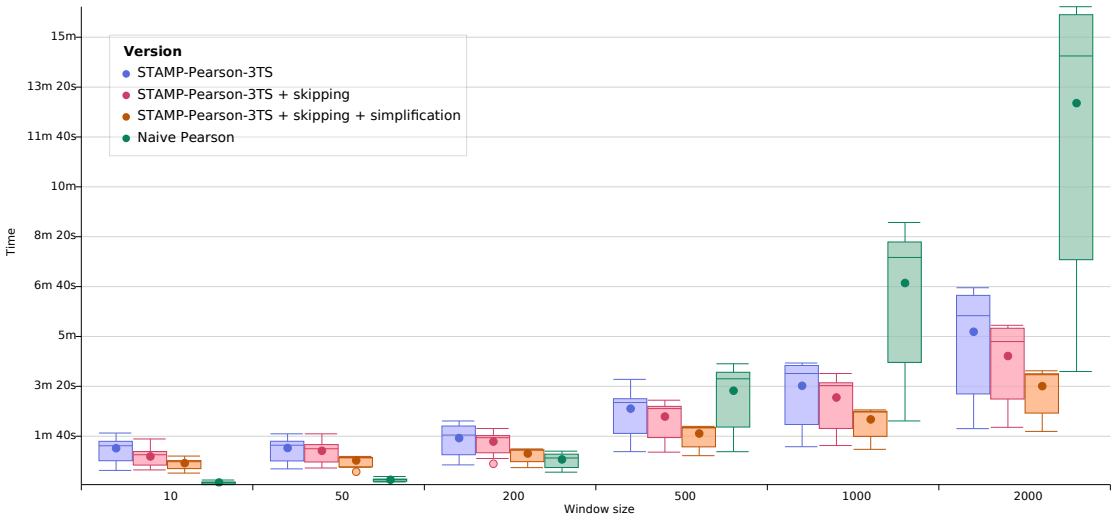

Figure 6.18: Each data point is the time of the correlation calculation of a single group of 3 time series.

The window skipping method yields different results depending on the data set. For the first data set, where almost no windows can be skipped (see Figure 6.7), the search of finding windows to skip makes the algorithm measurable slower. Adding simplification can make some windows skippable for very small window sizes (see Figure 6.11) and this is reflected in the plot as well, seeing the method even beat the normal STAMP-Pearson-3TS at a window size of 3. However, it is clear that for most window sizes in a data set where window skipping cannot occur often, window skipping only makes the algorithm slower.

For the second data set, which has larger time series including lots of linearly interpolated (and thus skippable) values, the window skipping method does give a considerable edge over the normal STAMP-Pearson-3TS algorithm, especially if the data analyst is willing to get only approximate results.

Chapter 7

Conclusions

7.1 Conclusion

There already exists previous research in detecting (cross-)correlations and similarities between pairs of time series and in groups [13, 17, 16, 14] and its applications [4, 11]. There also exists research regarding pairwise lagged correlation and its applications [2, 5]. However, the combination of simple lagged correlations and correlations in groups is something that has not been extensively researched before until now.

In this thesis, we propose a solution for finding lagged correlations in groups of three time series in the form of the optimized STAMP-Pearson-3TS algorithm and some window skipping techniques.

The first solution shows that when trying to detect ternary lagged correlations for relatively large window sizes, the optimized STAMP-Pearson-3TS proves to be a faster solution than a naive approach. For long time series and a relatively small window size, the naive approach is still faster. Lag bound is an essential tool to make longer time series more manageable, getting a faster result the tighter it is. This does come at the cost of limiting the search space for lagged correlations, but this can also be viewed as helpful when a data analyst is only interested in correlations with little lag. As described as future work, in Section 7.2.1.1, by introducing a minimum lag as well, lag bound could be used as an analytical tool even more.

The second solution shows that window skipping can be used in some cases to speed up calculations, however, this is most effective for relatively small window sizes. When looking at larger window sizes, which usually provide more helpful information, the number of windows that can be skipped drops off significantly. Why this amount decreases can be firstly explained by the fact that there are fewer window positions when the window size is larger. Secondly, since time series of stocks are fluctuating a lot, the chance of finding a window position with just a straight line inside decreases when windows get larger. However, in the cases where it does work, the straight line skipping technique can be seen as an exact and free speed gain, since it can work even without simplification. Although, when the data has no or just a few window positions that can be marked as skippable, simply running the window skipping part will make the algorithm perform slower than the normal STAMP-Pearson-3TS. If a data analyst is willing to only find approximate solutions and there are windows that can be skipped, simplification can be used in combination with straight line window skipping to speed up the calculation more. This approach, however, this still suffers from the decline in the amount of windows that can be skipped when the window size increases. Using the motif skipping method is another approximate solution. In some cases and for certain window sizes, it is able to save more positions than the (simplified) straight line method, but the calculations of motif discovery are a lot heavier than discovering straight lines. When combined with the fact that simplification does not improve this method we have to

conclude that the motif skipping method is not a viable solution.

Overall, combining both chapters, we can conclude that the most efficient solution for ternary lagged correlations, found in this thesis, is:

- When window sizes are relatively small: A well-implemented naive algorithm.
- When lots of windows can be skipped (so longer time series with a lot of linear pieces, potentially as a result of interpolation): The optimized STAMP-Pearson-3TS algorithm with straight line skipping (and optional simplification).
- When not many windows can be skipped: The optimized STAMP-Pearson-3TS algorithm without straight line skipping.

Whether the tool can immediately be adopted by data analysts remains to be discussed. While there are many applications of this tool, for instance in the financial sector, we think that more features need to be implemented to make it more usable. A couple of these features are mentioned in Section 7.2. The aim of the thesis, however, is to make the detection of ternary lagged correlations as fast and efficient as possible. Making it more accessible to use in practice is up for future work.

7.2 Future work

As mentioned before, this thesis does open the door to plenty of future work. In this section we go over future work for either sub-problem. The future work of the first sub-problem is discussed in Section 7.2.1 and the future work of the second in Section 7.2.2.

7.2.1 Regarding first sub-problem

While a lot of thought and time went into the development of the STAMP-Pearson-3TS algorithms, we decided to group a set of features outside the scope of the thesis. This future work, however, might be still an interesting addition one day. It includes adding a minimum lag (Section 7.2.1.1), two optional methods to return more than simply the best correlation in a group of three time series (Section 7.2.1.2), the idea of introducing a correlation threshold (Section 7.2.1.3), the ability to calculate the lagged correlation for more than three time series (Section 7.2.1.4), and finally, considering the option of using other aggregation methods and ways to lock the lag between certain time series (Section 7.2.1.5).

7.2.1.1 Minimum lag

While the current lag bound implementation only considers a maximum lag bound size, depending on the knowledge of the data analyst, it might be beneficial to have a minimum lag as well. This could limit the running time of the algorithm if the data analyst, for instance, were only interested in correlations lagged by more than two months. While currently not implemented, it might be a straightforward addition for the future.

7.2.1.2 Getting top-x results

Currently, the STAMP-Pearson-3TS algorithms find the single best window positions and aggregation method based on the correlation within three time series. This is very helpful for the research question since many groups of time series need to be considered, and the best group of three time series needs to be found. However, it might also be beneficial to find the top-x of best windows placements and aggregation methods within a group of three. The current algorithms do not consider this yet, but we can still hint towards two potential tactics: The ability for STAMP-Pearson-3TS to return multiple values and the ability to skip certain indices.

Return multiple values The first option would be to modify the returning values of the algorithms. Instead of returning just the single best correlation and indices, the algorithms would return multiple ones. This would entail some expanding of the algorithms STAMP-Pearson-3TS (Algorithm 3), STAMP-Pearson-3TS-Sub (Algorithm 4), MASS-Pearson-3TS (Algorithm 3), MASS-Pearson-Agg (Algorithm 6), and MASS-Pearson (Algorithm 1). In practice, it might not be very interesting to look further than a top-5, so the memory usage, while it will increase, will not increase by much.

Skip indices The second option would be to simply run the algorithms x times and provide a list of indices to skip after the first run. Compared to the previous method, this method trades memory for time. Instead of collecting and comparing multiple values at each algorithm, this version collects the top-x results one at a time. To implement this idea, only MASS-Pearson-Agg (Algorithm 6) and MASS-Pearson (Algorithm 1) need to be modified.

7.2.1.3 Correlation threshold

Based on what the data analyst is looking for, the algorithm could be tweaked to stop when a correlation above a certain threshold is measured. While this does not give the absolute best window placements for correlation, it can give a good indication of whether a group of three time series has any correlation, all while saving a lot of time.

7.2.1.4 More than three time series

While the current approach already increased the amount of time series considered from two (from the Matrix Profile papers [13]) to three, this concept could be increased to n time series. However, the usability of this might be debatable, especially since the number of calculations will increase exponentially. Although, if a data analyst would require a solution like that, then STAMP-Pearson-3TS would be a good base to start from.

7.2.1.5 Other aggregation methods and other lag methods

The current implementation is built around the concept of aggregating all unique combinations of two time series in the group of three by average and checking how it correlates with the third. It might however also be interesting to look at different aggregation methods, like aggregating the maximum or minimum value. Aside from that, it might also be interesting to lock the windows of two time series that are aggregated together. From a data analytical perspective, this could provide good information about how a single time series can correlate with multiple other time series at a different point in time.

7.2.2 Regarding second sub-problem

The experiments regarding this sub-problem are not very promising for all types of data. However, this does not mean that it is impossible to skip certain window positions to speed up the search for the best ternary lagged correlation in any dataset.

One technique considered is simplifying the time series to just 45-degree angles and trying to find how the presence of monotonicity can inform the sliding algorithm when to skip a section. Unfortunately, we are not able to detect any repeating patterns using this.

Another approach looks at where inside the time series the best correlations are usually found, for instance in a 'curve point' (where the time series slope goes from positive to negative or vice versa) or in a straight piece. Disappointingly, no conclusion can be drawn from this either.

While our research did not find anything usable in this area, there might still be something here. This, however, has got to be classified as future work.

Bibliography

[1] Zoltán Bankó and János Abonyi. Correlation based dynamic time warping of multivariate time series. *Expert Systems with Applications*, 39(17):12814–12823, 2012.

[2] Kinga Bierwiaczonek, Jonas R. Kunst, and Olivia Pich. Belief in COVID-19 conspiracy theories reduces social distancing over time. *Applied Psychology: Health and Well-Being*, 12(4):1270–1285, 2020.

[3] David H. Douglas and Thomas K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10:112–122, 1973.

[4] Maria Lourdes González-Miret, Anass Terrab, Dolores Hernanz, Maria Ángeles Fernández-Recamales, and Francisco J. Heredia. Multivariate correlation between color and mineral composition of honeys and by their botanical origin. *Journal of Agricultural and Food Chemistry*, 53(7):2574–2580, 2005. PMID: 15796597.

[5] Renata R. V. Gonçalves, Jurandir Zullo Jr, Luciana A. S. Romani, Cristina R. Nascimento, and Agma J. M. Traina. Analysis of NDVI time series using cross-correlation and forecasting methods for monitoring sugarcane fields in brazil. *International Journal of Remote Sensing*, 33(15):4653–4672, 2012.

[6] E.F. Guedes and G.F. Zebende. DCCA cross-correlation coefficient with sliding windows approach. *Physica A: Statistical Mechanics and its Applications*, 527:121286, 2019.

[7] Koen Minartz, Jens d'Hondt, and Odysseas Papapetrou. Multivariate correlations discovery in static and streaming data. Technical report, Eindhoven University of Technology, Oct 2021.

[8] Ladislav Kristoufek. Measuring correlations between non-stationary series with DCCA coefficient. *Physica A: Statistical Mechanics and its Applications*, 402:291–298, 2014.

[9] Abdullah Mueen, Suman Nath, and Jie Liu. Fast approximate correlation for massive time-series data. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 171–182, 2010.

[10] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Comput. Graph. Image Process.*, 1:244–256, 1972.

[11] Stefan Liess, Saurabh Agrawal, Snigdhansu Chatterjee and Vipin Kumar. A teleconnection between the west siberian plain and the ENSO region. *Journal of Climate*, 30.1:301–315, 2017.

[12] C. M. Yeh, N. Kavantzas, and E. Keogh. Matrix profile VI: Meaningful multidimensional motif discovery. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 565–574, Nov 2017.

[13] C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh. Matrix profile I: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 1317–1322, Dec 2016.

[14] Naiming Yuan, Elena Xoplaki, Congwen Zhu, and Juerg Luterbacher. A novel way to detect correlations on multi-time scales, with temporal evolution and for multi-variables. *Scientific Reports*, 6(1):27707, Jun 2016.

[15] Xiaojun Zhao, Pengjian Shang, and Jingjing Huang. Several fundamental properties of DCCA cross-correlation coefficient. *Fractals*, 25(02):1750017, 2017.

[16] Y. Zhu, C. M. Yeh, Z. Zimmerman, K. Kamgar, and E. Keogh. Matrix profile XI: SCRIMP++: Time series motif discovery at interactive speeds. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 837–846, Nov 2018.

[17] Y. Zhu, Z. Zimmerman, N. S. Senobari, C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh. Matrix profile II: Exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 739–748, Dec 2016.