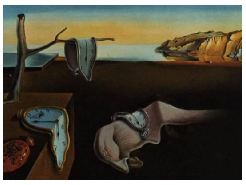


Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. doi: 10.18637/jss.v000.i00

IncDTW: An R Package for Incremental Calculation of Dynamic Time Warping

Maximilian Leodolter
Austrian Institute of Technology

Claudia Plant
University of Vienna

Norbert Brändle
Austrian Institute of Technology

Abstract

Dynamic Time Warping (DTW) is a popular distance measure for time series analysis and has been applied in many research domains. This paper proposes the R package **IncDTW** for time series clustering, classification, subsequence matching and k-Nearest Neighbors search based on DTW. DTW can measure dissimilarity between two temporal sequences which may vary in speed, with a major downside of quadratic runtime complexity in the number of observations. Especially for analyzing live data streams, subsequence matching or calculating pairwise distance matrices, the quadratic runtime complexity is unfavorable or can even make the analysis intractable. **IncDTW** tackles this problem of runtime intensive computations by the following contributions: (1) vector-based implementation of an incremental DTW algorithm to reduce the runtime and space complexity from a quadratic to a linear level in number of observations, (2) the novel runDTW algorithm enabling efficient subsequence matching and k-Nearest Neighbor search in a long time series, (3) parallelization and (4) transfer of the most intensive computations to C++. We apply **IncDTW** to cluster and classify multivariate live stream accelerometer time series for activity recognition, and demonstrate how to build a pattern recognition algorithm based on the incremental calculation of DTW to detect representations of a query pattern of different lengths in longer time series. Runtime experiments for various data analysis tasks emphasize the broad applicability of **IncDTW**.

Keywords: dynamic time warping, time series, k-NN, subsequence matching, distance measure, clustering, classification.

1. Introduction

Time series are sets of observations that follow a consecutive temporal relation. Many time series data analysis tasks such as clustering, classification, outlier detection or pattern matching require the definition of a distance measure. Many distance measures such as the Euclidean distance are rather ill-suited whenever two time series are shifted in time, locally recorded with different sampling rates, warped, or have different lengths. Dynamic Time Warping (DTW) was originally proposed by Sakoe and Chiba (1978), and has since been the distance measure of choice in many works for time series analysis (Berndt and Clifford 1994; Keogh 2002; Ding, Trajcevski, Scheuermann, Wang, and Keogh 2008; Kwankhoom and Muneesawang 2017; Oregi, Pérez, Del Ser, and Lozano 2017; Giorgino *et al.* 2009). DTW is capable of dealing with deformed time series by identifying the best alignment of two time series in a dynamic way.

The major downside of DTW is its quadratic runtime complexity in the number of observations. Expensive computation of DTW is particularly unfavorable for online algorithms processing continuous data streams, where time series analysis must be faster than the elapsed time between consecutive observations. One solution to reduce complexity for online processing is to incrementally calculate DTW by recycling interim results of previous calculations for every new observation. Without any loss of accuracy, such an incremental processing allows reducing computation time complexity towards linear level in number of observations.

Kwankhoom and Muneesawang (2017) applied the incremental DTW principle for online algorithms which re-identify movement trajectories of persons captured with a 3D depth sensing camera, where time series matching is updated as soon as new video frames are recorded. The Online-DTW (ODTW) algorithm presented by Oregi *et al.* (2017) also makes use of the incremental calculation principle.

Apart from stream processing, computation time is also key whenever relatively short query patterns must be detected in longer time series, which usually requires a large number of comparisons between many segments of the longer time series and the query pattern. For example, the Caterpillar algorithm presented by Leodolter, Brändle, and Plant (2018) scans long time series to detect patterns which are possibly warped or of different lengths than a query pattern, based on a combination of incremental DTW calculation and the Minimum Description Length. The incremental calculation of DTW enables the Caterpillar algorithm to search the space of possible fits in reasonable time and yields better results than comparable time series pattern recognition techniques. This paper presents the incremental update of the DTW distance and the R package **IncDTW**, the functions of which can serve as components for pattern recognition algorithms.

Dynamic Time Warping has already been applied in many research domains and also published in different software packages and programming languages. Table 1 gives an overview of R packages for DTW computation available on the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/`. The package **dtw** (Giorgino *et al.* 2009) offers functions for DTW calculation with different step patterns (see (2) and (3)), warping path restrictions and plotting functions, also for a profound visual analysis of warping alignments of two time series. **dtwclust** (Sarda-Espinosa 2018) puts emphasis on clustering time series based on DTW distances. The functions for DTW calculations are wrappers for those of the **dtw** package. The package **dtwSat** (Maus, Câmara, Appel, and Pebesma 2019) provides with the Time-Weighted Dynamic Time Warping a distance method customized to analyzing satellite image time series. **ucrsuite** (Boersch-Supan 2016) is the R version of UCR Suite (Rakthanmanon, Campana, Mueen, Batista, Westover, Zhu, Zakaria, and Keogh 2012) which is a Nearest Neighbor search algorithm accelerated by lower bounding and pruning methods. It detects the closest fit to a query time series in either one long time series or many of the same length. To the best of our knowledge **ucrsuite** is – besides **IncDTW** – the only package with a vector-based implementation of the DTW algorithm, thus avoiding memory allocation of matrices. However, the package does neither support multivariate time series nor full alignments for time series of different lengths (i.e., from begin to end for both time series). The package **parallelDist** (Eckert 2017) is the parallel implementation of the function `stats::dist()` by incorporating **RcppParallel** (Allaire, Francois, Ushey, Vandenbrouck, Geelnard, and Intel 2018) to speed up computations.

The main contributions of this paper and the R package **IncDTW** are:

- the principle of the incremental DTW calculation ready to use in R functions
- vector-based implementation of the DTW algorithm – also for multivariate time series – to decrease the computation time
- runDTW: a runtime efficient algorithm to detect the non-overlapping k-NN within a long time series, i.e., the k disjoint subsequences with minimum DTW distance to a query pattern
- support for R users to build their own algorithms based on incremental and recycling principles of the DTW calculation for typical pattern recognition problems for time series data

This paper is organized as follows: Section 2 gives an introduction to DTW in general and explains the incremental calculation. Section 3 describes the R package **IncDTW**, discusses the vector-based functions and the algorithm runDTW for k-NN search. Section 4 shows the application of **IncDTW** for typical time series data mining tasks and demonstrates how to write customized algorithms. Section 5 concludes this paper and gives an outlook of future developments.

2. Dynamic Time Warping

In the following we recapitulate the classic Dynamic Time Warping algorithm from Sakoe and Chiba (1978) which calculates the distance measure between a query time series $\mathbf{q}$ and a candidate time series $\mathbf{c}$, and their alignment – the so-called warping path – providing information which observations of $\mathbf{q}$ are best matched to the respective observations of $\mathbf{c}$.

The distance measure DTW is defined as the minimal cumulative costs of the shortest non-linear alignment of two time series $\mathbf{q}$ and $\mathbf{c}$. This alignment has the following properties:

1. Boundary conditions: The first element of $\mathbf{q}$ is aligned to the first element of $\mathbf{c}$, and the last element of $\mathbf{q}$ is aligned to the last element of $\mathbf{c}$. Relaxing these conditions allows to find an open alignment, i.e., a partial alignment of two time series with lowest DTW distance (normalized for the lengths). Appendix **??** discusses open alignments (open-end, open-begin and open increment) in more detail, and we apply open alignments in Sec. 4 to accelerate pattern recognition algorithms for searching patterns of different lengths.
2. Monotonicity: Consecutive elements of $\mathbf{q}$ and $\mathbf{c}$ must not be aligned out of time order. The DTW algorithm also returns vectors of indices of $\mathbf{q}$ and $\mathbf{c}$ defining the ordering of the best aligned observations. These vectors must be monotonically increasing, such

Package name	on CRAN since	incre-mental	vector-based	diff. lengths	multi-variate	k-NN	Description/ Focus
IncDTW	2017	Yes	Yes	Yes	Yes	Yes	incremental and fast vector-based DTW calculations, described in this paper.
dtw	2007	No	No	Yes	Yes	No	Highly functional implementation of DTW (Giorgino *et al.* 2009)
dtwclust	2015	No	No	Yes	Yes	No	time series clustering with DTW (Sarda-Espinosa 2018)
dtwSat	2015	No	No	Yes	Yes	No	Time-Weighted DTW, for satellite image time series (Maus *et al.* 2019)
rucrdtw	2016	No	Yes	No	No	No	1NN-search via DTW (Boersch-Supan 2016)
parallelDist	2017	No	No	Yes	Yes	No	The pendant to `dist()` of **stats** (R Core Team 2018), by incorporating **RcppParallel**, (Eckert 2017)

Table 1: Overview of various R packages with different emphasis on calculating and applying the DTW distance.

that $i_k \leq i_{k+1}$, where $1 \leq i_k \leq n = |\mathbf{q}|$, and i_k defines which elements of $\mathbf{q}$ are aligned to $\mathbf{c}$ at the k-th point of time. The same applies to the indices $j_k \leq j_{k+1}$ defining which elements of $\mathbf{c}$ are aligned to $\mathbf{q}$ at the k-th point of time.

3. Non-linear alignment: In contrast to the Euclidean distance, one observation of $\mathbf{q}$ can be aligned to more than one observation of $\mathbf{c}$, and vice versa. Hence it is possible that $i_k = i_{k+1}$ or $j_k = j_{k+1}$.

4. Restrictions: Global or local warping path restrictions can be applied to reduce the space of possible alignments. The most known is the Sakoe Chiba warping window Sakoe and Chiba (1978), where the time difference of two aligned observations must not exceed the window size parameter, ω: $|i_k - j_k| \leq \omega \; \forall k$.

5. Local distance measure: The distance of two (possibly multivariate) observations of the time series $\mathbf{q}$ and $\mathbf{c}$ can be defined by any distance metric. The standard metrics are the 1-norm and 2-norm. Appendix **??** elaborates how to apply customized distance functions.

6. Step Pattern: The step pattern defines how the local distances are accumulated to

calculate the global cost matrix and the walking path. The two popular step patterns (2) and (3) are implemented in **IncDTW**. Sakoe and Chiba (1978) and Giorgino *et al.* (2009) give a more detailed discussion on step patterns.

It is worth noting that the DTW distance measure is not a metric, since it does not fulfill the triangle inequality. Consequently, lower bounding with the help of the reverse triangle inequality is not possible, which is a method applied for fast nearest neighbor search Wang (2011).

For the two time series $\mathbf{q}$ of length n and $\mathbf{c}$ of length m we define $\mathbf{C} \in \mathbf{R}^{n \times m}$ as the local cost matrix, where

$$\mathbf{C}_{i,j} := d(\mathbf{q}_i, \mathbf{c}_j), \tag{1}$$

with $d(.,.)$ as a local distance function for univariate or multivariate time series as described above. The global cost matrix $\mathbf{G} \in \mathbf{R}^{n \times m}$ is determined in an iterative fashion, where each element depends on its predecessors. The step pattern defines these dependencies by weighting and selecting the predecessors. Giorgino *et al.* (2009) present a more detailed discussion on step patterns, here we concentrate on the two most popular and start with the original and naive step pattern that regards the direct neighboring elements in $\mathbf{G}$ equally weighted:

$$\mathbf{G}_{i,j} = \begin{cases} \sum_{k \le i} \mathbf{C}_{k,1} & j = 1 \\ \sum_{l \le j} \mathbf{C}_{1,l} & i = 1 \\ \mathbf{C}_{i,j} + \min(\mathbf{G}_{i-1,j},\ \ \mathbf{G}_{i,j-1},\ \ \mathbf{G}_{i-1,j-1}) & i, j > 1. \end{cases} \tag{2}$$

Algorithm (2) is applied in several works about time series clustering, classification, indexing and pattern mining

(Fu 2011; Berndt and Clifford 1994; Sakurai, Faloutsos, and Yamamuro 2007; Keogh 2002; Rath and Manmatha 2003; Keogh and Pazzani 2000; Rakthanmanon *et al.* 2012). Another typical step pattern, that is also the default step pattern in the R package **dtw**, is called 'symmetric2'. Here the diagonal step is weighted with a weight of 2:

$$\mathbf{G}_{i,j} = \begin{cases} \sum_{k \le i} \mathbf{C}_{k,1} & j = 1 \\ \sum_{l \le j} \mathbf{C}_{1,l} & i = 1 \\ \min(\mathbf{C}_{i,j} + \mathbf{G}_{i-1,j},\ \ \mathbf{C}_{i,j} + \mathbf{G}_{i,j-1},\ \ 2 \cdot \mathbf{C}_{i,j} + \mathbf{G}_{i-1,j-1}) & i, j > 1. \end{cases} \tag{3}$$

Both step patterns (2) and (3) are discussed as special cases of the general formulation in Sakoe and Chiba (1978). The direction matrix $\mathbf{D} \in \mathbf{N}^{n \times m}$ gives information about the alignment of the two time series and is calculated simultaneously with the calculation of $\mathbf{G}$. The following equation defines $\mathbf{D}$ for the step pattern of (2):

$$\mathbf{D}_{i,j} = \begin{cases} 1 & \text{if} \quad \mathbf{G}_{i,j} = \mathbf{C}_{i,j} + \mathbf{G}_{i-1,j-1} \\ 2 & \text{if} \quad \mathbf{G}_{i,j} = \mathbf{C}_{i,j} + \mathbf{G}_{i,j-1} \\ 3 & \text{if} \quad \mathbf{G}_{i,j} = \mathbf{C}_{i,j} + \mathbf{G}_{i-1,j}. \end{cases} \tag{4}$$

The DTW distance measure is stored in the last column of the last row of $\mathbf{G}$, $\mathbf{G}_{nm}$, and indicates the cheapest cumulative costs to align $\mathbf{q}$ and $\mathbf{c}$. The warping path vector $\mathbf{w}$ is an excerpt of the direction matrix $\mathbf{D}$ and achieved by backtracking $\mathbf{D}$. Starting at the last row and last column of **D**, backtracking (Algorithm 1) checks the cheapest next step (1 is diagonal, 2 is horizontal, 3 is vertical) and stores this integer in a vector. The backtracking algorithm also returns the vectors **ii** and **jj**, the vectors of indices of **q** and **c** for the best alignment in the respective order.

Algorithm 1 Backtracking the direction matrix **D** delivers the warping path **w**

```
1: procedure BACKTRACKING($\mathbf{D}$)
2:     i ← n                                  ▷ $n$ = length of the time series $\mathbf{q}$
3:     j ← m                                  ▷ $m$ = length of the time series $\mathbf{c}$
4:     $\mathbf{w}, \mathbf{ii}, \mathbf{jj}$ ← empty vectors
5:     repeat
6:         step ← $\mathbf{D}(i, j)$;
7:         if step == 1 then
8:             i ← i - 1
9:             j ← j - 1
10:        else if step == 2 then
11:            j ← j - 1
12:        else
13:            i ← i - 1
14:        end if
15:        $\mathbf{ii}$ ← append(i, $\mathbf{ii}$)
16:        $\mathbf{jj}$ ← append(j, $\mathbf{jj}$)
17:        $\mathbf{w}$ ← $append(step, \mathbf{w})$
18:    until $i < 0 \mid j < 0$ return $\mathbf{w}$, $\mathbf{ii}$ and $\mathbf{jj}$
19: end procedure
```

2.1. Incremental calculation

Calculating the DTW distance measure is computationally expensive, especially for long time series, due to the quadratic complexity $O(n \cdot m)$, where n and m are the lengths of the time series **q** and **c**, respectively. To update an alignment of two time series after new observations, it is possible to reuse interim results instead of calculating DTW from scratch. This incremental calculation has a complexity of $O(n \cdot k)$ to update the DTW distance for k new observations of **c**, instead of $O(n \cdot (m + k))$ which is the complexity of calculating the DTW distance from scratch.

The following two applications benefit from the the principle of incremental DTW calculation:

- Live streaming data: For live systems computation time is key. `idtw()` (and `idtw2vec()`) facilitates a fast update of time series distance measures when new observations arise. This can be of interest for any system analyzing live data streams.

- Scanning a longer time series: To detect a match of a query pattern within a longer time series is a common problem in time series data mining. Yeh, Zhu, Ulanova, Begum, Ding, Dau, Silva, Mueen, and Keogh (2016) use the Euclidean distance to detect query matches. Leodolter *et al.* (2018); Rakthanmanon *et al.* (2012); Sakurai *et al.* (2007) use DTW for the same problem to make use of the advantages of DTW over the Euclidean distance. Since DTW can measure similarities of time series of different lengths, it is capable to detect matches of a query pattern of varying lengths. However computing and comparing all combinations of alignments to find the best would be computationally expensive or even intractable. The incremental algorithm can reduce the computational effort. Section 4.1 and 4.3 demonstrate this by applying `IncDTW::idtw()` (and `IncDTW::idtw2vec()`) scanning a longer time series with an incremental growing scanning window.

The input to incrementally calculate DTW of $\mathbf{q}[1:n]$ and $\mathbf{c}[1:m+k]$ is the output of the former calculation DTW$(\mathbf{q}[1:n],\ \mathbf{c}[1:m])$. This output is composed of three matrices: the global cost matrix $\mathbf{G}^0$, the local cost matrix $\mathbf{C}^0$ and the direction matrix $\mathbf{D}^0$. Additional required input is the time series of new observations of $\mathbf{c}$. To calculate the global cost matrix $\mathbf{G}^1$ of DTW$(\mathbf{q}[1:n],\ \mathbf{c}[1:m+k])$, we append new costs and direction entries to the previously calculated matrices and proceed analogously to (2):

1. First we build the local cost Matrix $\mathbf{C}^1$:
$$\mathbf{C}^1_{ij} := \begin{cases} \mathbf{C}^0 ij & i \le m \\ dist(q_i, c_j) & m < i \le m+k. \end{cases} \tag{5}$$

2. Next the global cost matrix is appended to the former results and new entries are defined analogously to (2):
$$\mathbf{G}^1_{ij} := \begin{cases} \mathbf{G}^0 ij & i \le m \\ \sum_{k \le i} \mathbf{C}_{k,1} & j = 1 \\ \mathbf{C}_{i,j} + \min(\mathbf{G}_{i-1,j},\ \ \mathbf{G}_{i,j-1},\ \ \mathbf{G}_{i-1,j-1}) & else \end{cases} \tag{6}$$

3. The direction matrix $\mathbf{D}^1$ is calculated simultaneously to $\mathbf{G}^1$.

4. Finally, the warping path needs to be calculated completely new from scratch, since in general it can not be excluded that new observations may open up completely different options to warp the two time series.

Equation 6 is the incremental version of (2). For (3) the definition of the new entries of $\mathbf{G}$ is analogous, as for any other step pattern presented in Sakoe and Chiba (1978).

2.2. The plane of possible fits

A well known pattern recognition problem is to detect a query pattern $\mathbf{q}$ in a longer time series $\mathbf{c}$. Fu (2011) give an overview about publications dealing with this research question, also called subsequence matching (or the familiar problem: pattern discovery). A subsequence of a time series $\mathbf{c}$ are all observations between a start index a and end index b, $\mathbf{c}[a,b]$, where $1 \le a \le b \le |\mathbf{c}|$. Next we give an overview of different forms of subsequence matching before (7) to (12) formalize these problems, and finally Sec. 3 introduces the functions of **IncDTW** to solve these problems.

- 1-SS: finding the one subsequence in a long time series $\mathbf{c}$ that has minimum distance to a query time series $\mathbf{q}$ and the same length as $\mathbf{q}$. This problem is similar to the 1-NN search. Equation 7 formalizes this problem.

- δ-SS: as 1-SS, but find all subsequences with a distance smaller than a threshold δ, similar to k-NN search. See (8) and (9).
- 1-SS^{vl}: as 1-SS, but with varying length. So the index b is free, and not restricted by $b = a + |\mathbf{q}| - 1 = a + n - 1$, as 1-SS and δ-SS require. See (10).
- δ-SS^{vl}: as δ-SS, but with varying lengths of all k fits. See (11) and (12).

We formalize the problem 1-SS as finding the start index a^* such that

$$\begin{aligned} a^* = \underset{a}{\operatorname{argmin}} \;\; & d(\mathbf{q},\ \mathbf{c}[a:b]) \\ \text{s. t.} & \\ 1 \le a \le & \; m - n + 1 \\ b = & \; a + n - 1, \end{aligned} \tag{7}$$

where $d(.,.)$ is the distance function, $m = |\mathbf{c}|$ and $n = |\mathbf{q}|$. The end index b^* is defined by $b^* = a^* + n - 1$. We call such a couple of indices (a, b) a fit of $\mathbf{q}$ in $\mathbf{c}$. Figure 1a illustrates that possible fits can only be located on the red line which fulfills $b = a + n - 1$. A similar problem is δ-SS, that is to find the set I of indices such that:

$$I = \{(a_i, b_i)|\ d(\mathbf{q},\ \mathbf{c}[a_i : b_i]) \le \delta \wedge 1 \le a_i \le m - n + 1 \wedge b_i = a_i + n - 1\}. \tag{8}$$

Adding the condition $b_i < a_{i+1}$ in the following formulation prevents detected fits to overlap:

$$I = \{(a_i, b_i)|\ d(\mathbf{q},\ \mathbf{c}[a_i : b_i]) \le \delta \wedge 1 \le a_i \le m - n + 1 \wedge b_i = a_i + n - 1 \wedge b_i < a_{i+1}\}. \tag{9}$$

For dealing with 1-SS^{vl} we are looking for fits of $\mathbf{q}$ in $\mathbf{c}$ that are possibly of different lengths than $\mathbf{q}$, and so need to relax the condition $b = a + n - 1$. This is only possible if the applied distance function is capable of comparing time series of different lengths, as DTW is. The general formalization:

$$\begin{aligned} (a^*, b^*) = \underset{a,b}{\operatorname{argmin}} \;\; & d(\mathbf{q},\ \mathbf{c}[a:b]) \\ \text{s. t.} & \\ 1 \le a \le & \; b \le m. \end{aligned} \tag{10}$$

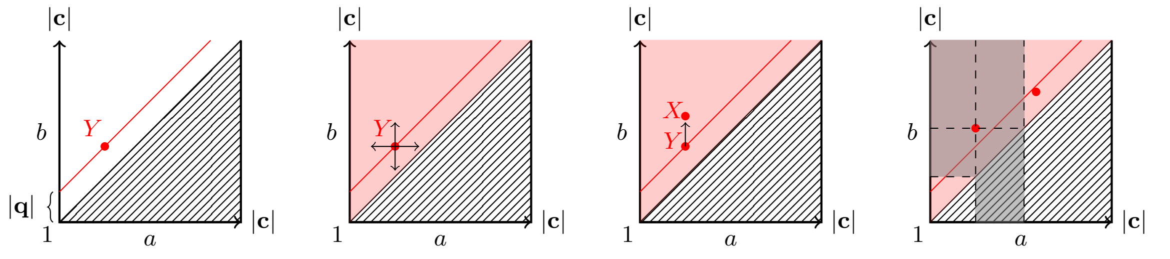
Figure 1b shows the red area of possible fits. Since the start index a needs to be smaller or equal than the end index b, no valid index combinations can be found in the shaded area below the line with slope $= 1$, starting at the lowest possible start index, 1.

δ-SS^{vl} describes the problem of finding multiple fits of varying lengths. The following set I defines the target set of indices:

$$I = \{(a_i, b_i)|\ d(\mathbf{q},\ \mathbf{c}[a_i : b_i]) \le \delta \wedge 1 \le a_i \le b_i \le m\}. \tag{11}$$

Again the following additional restrictions prevent detected fits to overlap (visualized by the dashed rectangles in Fig. 1d):

$$I = \{(a_i, b_i)|\ d(\mathbf{q},\ \mathbf{c}[a_i : b_i]) \le \delta \wedge 1 \le a_i \le b_i \le m \wedge b_i < a_{i+1}\}. \tag{12}$$



(a) Detected fit Y on the red line of all possible (a, b) combinations where $b - a = |\mathbf{q}|$.

(b) Combinations in the red triangle fulfill $a \leq b$. Arrows indicate in- and decremental (partial) DTW calculation.

(c) Increment b to navigate from Y to a better fit X.

(d) Detecting multiple fits conditioning on non-overlapping with other fits.

Figure 1: Plane of possible fits: A detected fit is represented in the plane of possible index combinations as couple of initial and final index of a fit, (a, b). The lowest possible initial index is 1 and the maximum final index is the length of $\mathbf{c}$, $|\mathbf{c}|$. Since $a \leq b$ needs to hold, there are no possible combinations (a, b) in the lower triangle.

Typical approaches dealing with the problems (see (7) to (9)) scan $\mathbf{c}$ via a sliding window (overlapping or non-overlapping: Keogh and Lin (2005) elaborate why analyzing or clustering of overlapping subsequences of time series can cause misleading results), calculate a distance measure for each point of time $a \in \{1, ..., m - n + 1\}$ between $\mathbf{q}$ and the respective segment of $\mathbf{c}[a : (a + n - 1)]$ with the same length as $\mathbf{q}$, and finally decide which index – or set of indices– is best. Rakthanmanon *et al.* (2012) present UCR-Suite to attack the problem (7) with the three distance measures: Euclidean Distance, Dynamic Time Warping and Uniform Scaling. UCR-Suite applying the former two distance measures is limited in detecting fits of the same length as the query pattern, located on the red line of Fig. 1a, such as the point Y. Uniform scaling is a distance measure which – depending on a scaling parameter – stretches and compresses a query pattern uniformly and calculates the Euclidean distance for each of the stretched and compressed queries to the subsequences of equal lengths of the longer time series. So UCR Suite with Uniform Scaling is capable of detecting fits in the red area of Fig. 1b. The main difference to DTW-based approaches is that DTW can compensate time warps locally as well as globally, while uniform scaling is designed to detect similar patterns that are of different speed for the complete length of the query pattern, and not locally.

In contrast, the Caterpillar Leodolter *et al.* (2018) algorithm can search the entire red area in Fig. 1b for index combinations, compare (10). The Caterpillar algorithm extends and contracts the scanning window at the end (index b) or at the start (index a), while fixing the other, and calculates the DTW distance. This way the Caterpillar algorithm applies DTW as distance measure $d(.,.)$ and deals with the more general problem described in (12) to detect time warped fits of possibly different lengths of the query pattern. These movements of extending and contracting one end while fixing the opposite end resemble the movements of a crawling caterpillar, and are implemented via incremental and decremental calculations of DTW.

Any heuristic attempting to solve one of the problems (10) to (12) could start with a possible index combination Y in Fig. 1b. A possible improvement (reduction of the DTW distance DTW($\mathbf{q}$, $\mathbf{c}[a : b]$)) can be achieved via incremental calculations, visualized by the arrows in Fig. 1b. The vertical arrows adjust the end index b while fixing the start index a. The horizontal arrows adjust the start index a while fixing the end index b. Say the red point X in Fig. 1c describes the best fit of $\mathbf{q}$ and $\mathbf{c}[a^X : b^X]$. X could be detected by starting from Y and increasing b while fixing a.

In this paper we address the problems 1-SS and δ-SS – see (7) to (9) – by proposing the algorithm runDTW (see Sec. 3.4 to 3.5) and the R function `rundtw()`, that extends the idea of UCR Suite applying DTW to a k-NN search and also supports multivariate time series. In addition, for the problems 1-SS^{vl} and δ-SS^{vl} – see (10) to (12) – the class `planedtw` initialized by `initialize_plane()` and the methods for this class (`increment()`, `decrement()` and `reverse()`, see Sec. 3.3, 4.2 and 4.3) help to navigate in the plane of possible fits – similar to the Caterpillar algorithm – to evaluate possible index combinations at low calculation costs, instead of recalculating each combination from scratch.

3. The R package IncDTW

This section describes the functions of the R package **IncDTW** and how to apply them to calculate the DTW distance: (1) Matrix- or vector-based, (2) from scratch or incrementally, and (3) for subsequence matching. All results presented in this paper are achieved with version 1.1.2 of **IncDTW**. The computationally expensive parts of **IncDTW** are outsourced to C++ via the packages **Rcpp** (Eddelbuettel and François 2011) and **RcppArmadill** (Eddelbuettel and Sanderson 2014), and parallelized via the packages **parallel** (R Core Team 2018) and **RcppParallel** (Allaire *et al.* 2018).

3.1. Matrix-based implementation

The classical DTW implementation relies on the local cost matrix $\mathbf{C}$, the direction matrix $\mathbf{D}$ and the global cost matrix $\mathbf{G}$ (see Section 2). $\mathbf{C}$ can be stored as matrix or calculated entry-wise when $\mathbf{G}$ is calculated. Returning the matrices $\mathbf{G}$ and $\mathbf{D}$ facilitates a detailed analysis of the alignment of two time series. The plot and the visual analysis in Figures 3 and **??** are possible due to the information provided by the warping path, which in turn is an excerpt of the direction matrix $\mathbf{D}$ and is achieved by backtracking. The entry $\mathbf{C}_{ij}$ is the distance between q_i and c_j and can be described by any distance metric for univariate or multivariate time series dependent on the dimension of $\mathbf{q}$ and $\mathbf{c}$. In case of multivariate time series, they need to have the same dimension, but still can vary in number of observations. In the univariate case the 1-norm is equivalent to the 2-norm, which is the absolute value of the difference $|q_i - c_j|$.

The basic DTW algorithm for computing the global cost matrix $\mathbf{G}$, according to (2), steps through the local cost matrix $\mathbf{C}$. The following parameters characterize in detail how the algorithm defines $\mathbf{G}$ and finds the warping path:

- `dist_method`: The local distances are stored in $\mathbf{C}$, where $\mathbf{C}_{ij} =$ `dist_method`(q_i, c_j). So the parameter `dist_method` defines how the local distance of observations are measured. For O-dimensional time series the distances 'norm1', 'norm2' and 'norm2_square' are defined as:

$$\begin{aligned} ||\mathbf{q}_i, \mathbf{c}_j||_1 &:= \sum_{o=1}^{O} |\mathbf{q}_{io} - \mathbf{c}_{jo}| \\ ||\mathbf{q}_i, \mathbf{c}_j||_2 &:= \sqrt{\sum_{o=1}^{O} (\mathbf{q}_{io} - \mathbf{c}_{jo})^2} \\ ||\mathbf{q}_i, \mathbf{c}_j||_2^2 &:= \sum_{o=1}^{O} (\mathbf{q}_{io} - \mathbf{c}_{jo})^2. \end{aligned} \tag{13}$$

Appendix **??** demonstrates how to apply a customized distance function for the DTW algorithm.

- `ws`: The space of all possible alignments of two time series can be constrained by warping windows. As Sec. 2 mentions, the most popular constraint is the Sakoe-Chiba window Sakoe and Chiba (1978), which adjusts the DTW algorithm by setting $\mathbf{G}_{ij} = \infty$ if $|i - j| >$ `ws`. So `ws` defines the window size of allowed warping paths. If we set `ws` $= 0$ then only the diagonal of $\mathbf{G}$ is used for aligning $\mathbf{q}$ and $\mathbf{c}$, which is identical to the Euclidean distance. In this case the time series must have the same length. If the lengths of the time series differ more than `ws`, then obviously no valid alignment can be found.

- `step_pattern`: The step pattern defines how the DTW algorithm finds the cheapest path through the local cost matrix. In (2) the most basic and broadly applied step pattern "symmetric1" is used, where the direct neighbors are considered and all are weighted equally. In (3) the step pattern "symmetric2" is uses a weight of 2 for the diagonal step and 1 for the vertical and horizontal to compensate the favor of diagonal steps. The current version of IncDTW concentrates on these two patterns and we consider other step patterns for future developments. A more detailed discussion of step patterns gives Giorgino *et al.* (2009).

The following commands install and load the package **IncDTW**:

```
R> install.packages("IncDTW")
R> library("IncDTW")
```

First we define the help function `rw()` (which we also use in the next sections) to simulate a Gaussian random walk. Then a basic calculation of the DTW distance is done as follows:

```
R> rw <- function(n) cumsum(rnorm(n))
R> Q <- rw(100)
R> C <- rw(80)
R> result <- dtw(Q, C, ws = 30, step_pattern = "symmetric2")
R> result$distance
```

```
[1] 197.1266
```

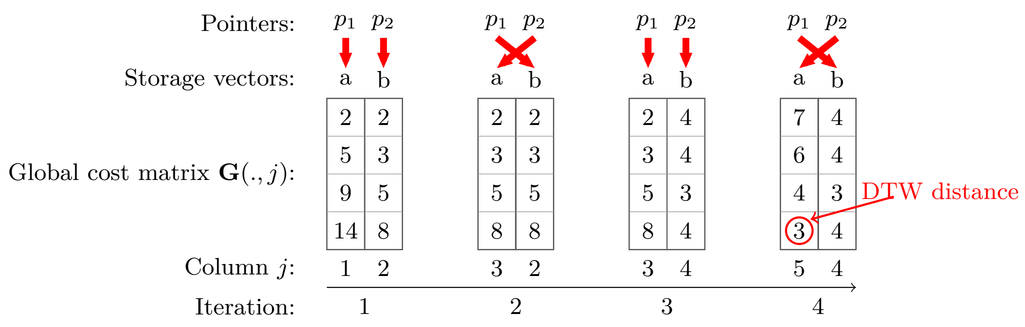


Figure 2: Iteratively overwriting vectors makes it obsolete to allocate matrices for DTW distance calculation.

3.2. Vector-based implementation

The matrix-based implementation is necessary for a detailed analysis of the alignment of two time series since it allows to calculate and return the warping path. Tasks such as nearest neighbor search, or the calculation of a matrix of pairwise distances to cluster or classify a database of time series require many DTW computations, and so the computation time of DTW is a major bottleneck.

The vector-based implementation offers a solution which is faster than the matrix based implementation, since memory allocation for matrices is not required. The space complexity for the matrix-based implementation is $O(m \cdot n)$ for calculating the local and global cost matrix, and the direction matrix. The vector-based computation principle is the same as for the matrix-based method, but instead of allocating matrices only two vectors are needed, and so the space complexity is reduced to $O(n)$. To obtain the DTW distance between the time series $\mathbf{q}$ and $\mathbf{c}$ the calculation of the j-th column of the global cost matrix $\mathbf{G}_{.,j}$ solely depends on the values of the previous column $\mathbf{G}_{.,j-1}$ and the respective distances of the time series $\mathbf{c}$ and the j-th entry of $\mathbf{q}$. Since there is no dependence on the column $\mathbf{G}_{.,j-2}$, the algorithm overwrites $\mathbf{G}_{.,j-2}$ with the newly calculated vector $\mathbf{G}_{.,j}$. Figure 2 demonstrates this principle with a simple example, and the following lines of code perform the DTW calculation for the same time series first via matrix based implementation (`dtw`) and second via vector based implementation (`dtw2vec`). The global cost matrix $\mathbf{G}$ is also printed to compare it to the vectors illustrated in Fig. 2.

```
R> Q <- c(3,4,5,6)
R> C <- c(1,3,3,5,6)
R> result <- IncDTW::dtw(Q,C)
R> result$gcm

     [,1] [,2] [,3] [,4] [,5]
[1,]    2    2    2    4    7
[2,]    5    3    3    4    6
[3,]    9    5    5    3    4
[4,]   14    8    8    4    3


R> dtw2vec(Q,C)$distance
```

```
[1] 3
```

In the first iteration step in Fig. 2 the initial two vectors a and b are defined according to the DTW step pattern and are identical to the first two columns of $\mathbf{G}$. In the second iteration the pointers p_1 and p_2 switch the address, so that the new entries of $\mathbf{G}_{.,3}$ overwrite a (where p_2 points to) and b (where p_1 points to) stores the entries of $\mathbf{G}_{.,2}$ of the previous iteration. Finally after four iterations the DTW distance measure (red encircled) is given in the last row of the last vector, which is identical to the fifth column of $\mathbf{G}$. Algorithm 2 formalizes this principle for the general case.

Algorithm 2 Vector based implementation of DTW without allocating any matrices

```
1: procedure VECTOR BASED DTW($\mathbf{q} \in \mathbb{R}^{n \times O}$, $\mathbf{c} \in \mathbb{R}^{m \times O}$)
2:     p1 ← cumsum(dist($q_1$, $\mathbf{c}$))          ▷ initial column of $\mathbf{G}$, $\mathbf{G}_{.,1}$
3:     for j in 2:m do
4:         $p2[1] \leftarrow$ dist($q_1$, $c_j$) + $p1[1]$
5:         for i in 2:n do
6:             $p2[i] \leftarrow$ step( dist($q_i$, $c_j$), min($p2[i-1]$, $p1[i]$, $p1[i-1]$))
7:         end for
8:         ptmp ← p1
9:         p1 ← p2
10:        p2 ← ptmp
11:    end for
12:    return p1[n]
13: end procedure
```

Even though the information about the warping alignment is lost by applying the vector-based method, the warping path still can be constrained by the parameter `ws`, defining the Sakoe Chiba warping window size. To continue with the same time series we constrain the warping path to allow a maximum deviation of the time index of $\mathbf{q}$ and $\mathbf{c}$ of 1, so $|i - j| \leq 1$. Since the warping path needs to adapt slightly the calculated distance changes from 3 to 4.

```
R> IncDTW::dtw2vec(Q, C, ws = 1)$distance
```

```
[1] 4
```

"Early abandoning" is a pruning method to break calculations if the cheapest possible alignment of two time series hits an upper bound (set by the user). This method helps to lower the calculation run time when comparing many time series. If the DTW algorithm hits this threshold the for-loop breaks and returns NaN. We continue the example and set the threshold to 2. Since no value in the fourth column of the global cost matrix is smaller or equal to 2, so $\mathbf{G}_{i,4} > 2\ \forall i$, the calculation stops here and NaN is returned.

```
R> IncDTW::dtw2vec(Q, C, threshold = 2)$distance
```

```
[1] NaN
```

3.3. IncDTW for incremental DTW calculations

For the incremental calculation of DTW we can choose between (1) the matrix based implementation to get more information about the alignment of the two time series and to facilitate analyses of the warping paths and (2) the vector based implementation for a faster distance calculation. For the latter the initial column in Alg.2 is defined as the last column of the former calculated global cost matrix, the last pointer vector respectively. That is, instead of passing matrices as input to the incremental DTW function, only the last column vector of **G** is passed for the vector based implementation. Further the class `planedtw` and its methods deal as convenient wrapper functions around the vetor based implementation. For a better understanding the following examples first walk through the more basic matrix based and vector based incremental update, and finally present the incremental update by hand of the `planedtw` class.

We demonstrate the principle of incrementally updating the DTW global distance matrix and the distance measure by hand of the following example. We define the time series **q** and **c**, and calculate the initial alignment with `dtw()`.

```
R> Q <- c(1:3, 4:1, 2:4)
R> C_initial <- c(1:3, 4, 4,  3:1) + 2
R> align_initial <- IncDTW::dtw(Q = Q, C = C_initial, return_wp = TRUE,
+    return_QC = TRUE, step_pattern = "symmetric1")
```

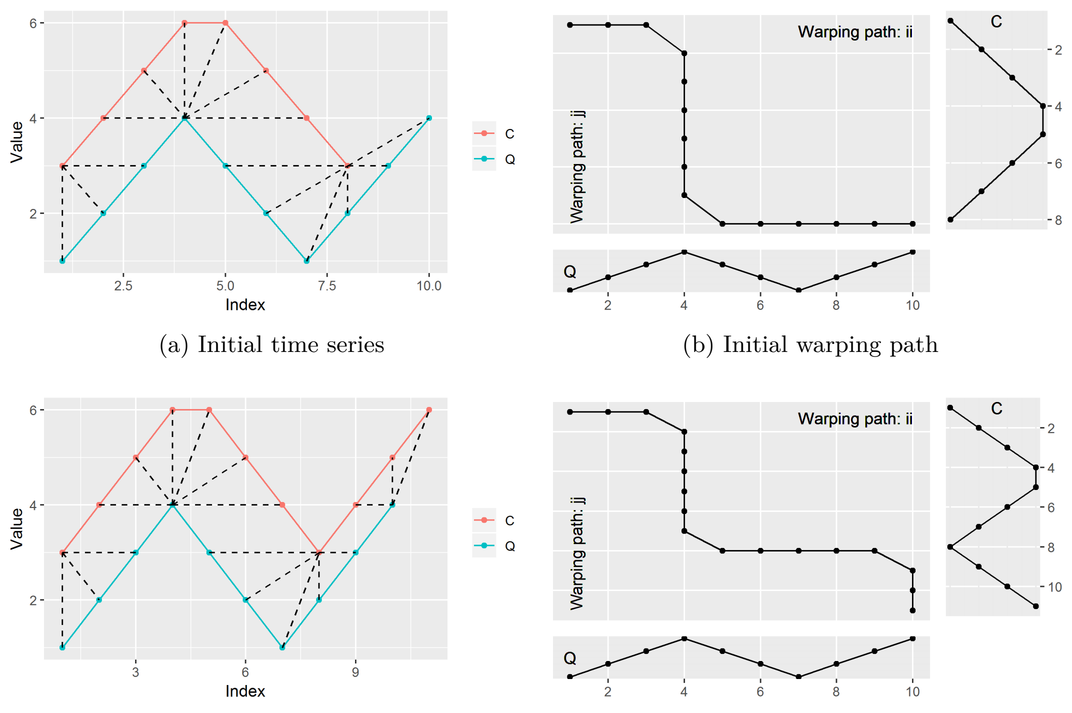
Figure 3a shows the time series and the aligned observations connected with dashed lines, and Fig. 3b contains the same information but focuses on the warping path (the main plot). One can see that the last observation of **c** is matched to the final six observations of **q**. We plotted the results with `plot(align_initial, type = "warp")` and `type = "QC"` respectively.

With new observations of **c** we can easily update the global cost matrix and the warping path by applying `idtw()` and compare the initial and updated versions of **G**.

```
R> C_newObs <- Q[8:10] + 2
R> C_update <- c(C_initial, C_newObs)
R> align_inc <- IncDTW::idtw(Q = Q, C = C_initial, newObs = C_newObs,
+    gcm = align_initial$gcm, dm = align_initial$dm, return_wp = TRUE,
+    return_QC = TRUE, step_pattern = "symmetric1")
R> identical(align_inc$gcm[, 1:8], align_initial$gcm)

[1] TRUE
```

As expected the first eight columns of the updated **G** and the initial **G** are identical. Figure 3c and 3d show the updated alignment and warping path. Finally, we compare the DTW distance of the updated calculation with the one from scratch (again using the basic function `dtw()`) and see that they are equal:

```
R> align_scr <- IncDTW::dtw(Q = Q, C = C_update, return_wp = TRUE,
+    return_QC = TRUE, step_pattern = "symmetric1")
R> align_scr$distance - align_inc$distance
```



(a) Initial time series (b) Initial warping path

(c) Updated time series (d) Updated warping path

Figure 3: Initially **c** and **q** are aligned. The warping path has 4 vertical steps before the update of **c**. The alignment is updated with an update of **c**.

```
[1] 0
```

We continue with the former example and perform the incremental calculation with the vector based implementation with `idtw2vec()`. This function distinguishes between an initial calculation and the incremental by checking whether results of previous calculations are passed or not, particularly the argument `gcm_lc`.

```
R> alignV_init <- IncDTW::idtw2vec(Q = Q, newObs = C_initial, gcm_lc = NULL)
R> alignV_inc  <- IncDTW::idtw2vec(Q = Q, newObs = C_newObs,
+    gcm_lc = alignV_init$gcm_lc_new)
```

Finally we compare the DTW distances of the incremental calculation (`idtw2vec()`) with the one from scratch (`dtw2vec()`) and their matrix based counterparts. As expected they are identical:

```
R> C_update <- c(C_initial, C_newObs)
R> alignV_scr <- IncDTW::dtw2vec(Q = Q, C = C_update)
R> c(align_scr$distance,  align_inc$distance,
+    alignV_scr$distance, alignV_inc$distance)
```

```
[1] 16 16 16 16
```

Section 4.4 gives runtime comparisons for these update functions.

New observations for both time series

With the knowledge of the basics and main modules for incremental calculation of DTW, `idtw()` and `idtw2vec()`, we apply the functions `initialize_plane()` and `increment()` which are convenient wrappers around `idtw2vec()`. The former function performs the initial calculation of `idtw2vec()` and returns an object of class `planedtw`, whereas the latter function applies the incremental calculation of `idtw2vec()`. Section 4.2 discusses further methods for the S3 class `planedtw` that support the navigation in the plane of possible fits.

If new observations for both time series are available, the update of the DTW calculation works in a consecutive fashion, similar to the case where only one time series is updated. The initial step is to apply `initialize_plane()` on the initial observations of **c** and **q**. Next we update the calculations for the first time series with `increment()`:

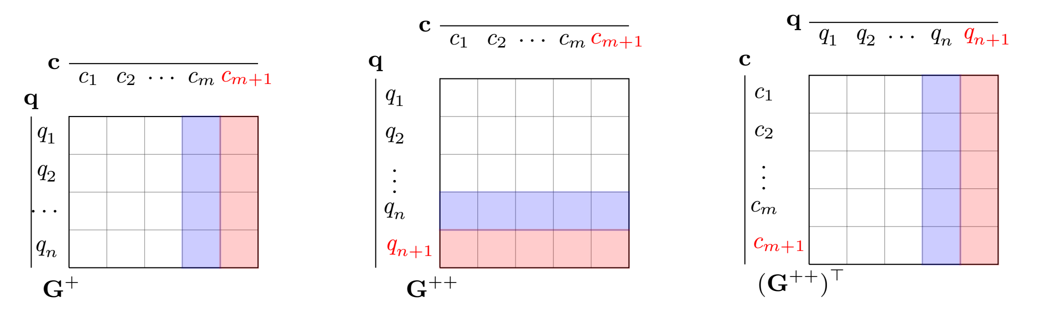
```
R> x <- initialize_plane(Q = Q, C = C_initial)
R> print(x)

control:
 dist_method step_pattern nQ nC   ws reverse
       norm1   symmetric2 10  8 NULL   FALSE

DTW distance:
14

Normalized DTW distance:
0.7777778

R> x <- increment(x, newObs = C_newObs)
```

Figure 4a visualizes relevant sections of the updated global cost matrix **G**. For a new observation of **c** the new area of **G** is colored red and the required column for the update in blue. Next we update **G** for the new observations of **q**. Again the red and blue rows in Fig. 4b indicate the updated and required areas. So we switch places of **q** and **c** as input for `idtw2vec()` and proceed analogously. Also we need to switch the last column with the last row of the global cost matrix. Figure 4c illustrates that switching **c** and **q** and the `gcm_lr` with `gcm_lc` is the same as transposing **G**. We could either switch the positions of these elements by hand and apply `idtw2vec()` directly, or apply the more convenient function `increment()` and set `direction = "Q"` to tell the function in which direction to update the last row and column of the global cost matrix:

```
R> Q_newObs <- rw(10)
R> x <- increment(x, newObs = Q_newObs, direction = "Q")
```

Finally we compare the results with the results from scratch and see that the calculated distance measures are equal:

```
R> x$distance - dtw2vec(c(Q, Q_newObs), c(C_initial, C_newObs))$distance
```



$\mathbf{c}$
c_1 c_2 $\cdots$ c_m c_{m+1}
$\mathbf{q}$
q_1 q_2 $\cdots$ q_n
$\mathbf{G}^{+}$
$\mathbf{q}$
q_1 q_2 $\vdots$ q_n q_{n+1}
$\mathbf{G}^{++}$
q_1 q_2 $\cdots$ q_n q_{n+1}
c_1 c_2 $\vdots$ c_m c_{m+1}
$(\mathbf{G}^{++})^{\top}$

(a) New observations for $\mathbf{c}$. (b) New observations for $\mathbf{q}$. (c) Transposed, new observations for $\mathbf{q}$.

Figure 4: Incremental update of $\mathbf{G}$ for new observations of $\mathbf{c}$ and $\mathbf{q}$. The updated areas of $\mathbf{G}$ are coloured in red and the areas storing the required input for the vector based update calculation are coloured blue.

```
[1] 0
```

3.4. runDTW: Detect multiple fits of the same length

δ-SS and the Equation 8 and 9 describe the set of subsequences of a long time series $\mathbf{c}$ that are closer than a threshold δ to a query pattern $\mathbf{q}$. This is a similar problem definition to the well known k-NN search, where the set of subsequences to be found are the k-nearest to the query time series. We propose the algorithm runDTW – also presented at (Leodolter, Plant, and Brändle 2019) – solving both problems δ-SS and the k-NN search by sliding a window along $\mathbf{c}$ and calculating the DTW distances to detect recurring patterns similar to $\mathbf{q}$. In the following we first discuss the algorithm runDTW to detect all fits beyond a threshold δ and subsequently elaborate how to adjust the algorithm slightly to detect the k nearest neighbors. To reduce computation time, the algorithm

- incrementally updates the local cost matrix $\mathbf{C}$,
- applies lower bounding methods to skip unnecessary DTW calculations,
- applies the vector-based implementation of the DTW algorithm,
- early abandons the DTW computation, and
- incrementally updates the normalization of the sliding window.

The algorithm runDTW was inspired by the work of Sakurai *et al.* (2007) and Rakthanmanon *et al.* (2012). However, neither supports a k-NN search, further the former does not consider normalization and the latter is for univariate time series only and especially designed for finding the nearest neighbor (7) which is similar but different to (9). The function `rundtw()` of **IncDTW** is the implementation of the algorithm runDTW.

The local normalization enables the algorithm to detect similar patterns in the presence of steady global trends. Equation 14 gives the standard formula for the min-max normalization to normalize the subsequence $\{\mathbf{c}_j\}_{j=i,\ldots i+n-1}$ of a time series $\mathbf{c}$, where $\mathbf{c}_i^{max}$ and $\mathbf{c}_i^{min}$ are the maximum and minimum of this subsequence. Algorithm 3 elaborates the running incremental normalization.

$$\mathbf{y}_j := \frac{\mathbf{c}_i^{max} - \mathbf{c}_j}{\mathbf{c}_i^{max} - \mathbf{c}_i^{min}} \qquad \forall j = i, ...i + n - 1. \tag{14}$$

Algorithm 3 Incremental update of the normalization of the sliding window of **c**

```
1: procedure INCNORM($\mathbf{c}$, $i$, $u$, $\mathbf{c}_i^{min}$, $\mathbf{c}_i^{max}$)
2:     Status ← 1
3:     if $\mathbf{c}_u < \mathbf{c}_i^{min}$ then        ▷ typically the upper index $u$ is equal $i + n - 1$
4:         $\mathbf{c}_i^{min} \leftarrow \mathbf{c}_u$
5:         Status ← 0
6:     end if
7:     if $\mathbf{c}_u > \mathbf{c}_i^{max}$ then
8:         $\mathbf{c}_i^{max} \leftarrow \mathbf{c}_u$
9:         Status ← 0
10:    end if
11:    if $\mathbf{c}_i = \mathbf{c}_i^{max} | \ \mathbf{c}_i = \mathbf{c}_i^{min}$ then        ▷ $\mathbf{c}_i$ drops out of the window
12:        $\mathbf{c}_i^{max} \leftarrow \max(\mathbf{c}_{i+1}, \ ... \ \mathbf{c}_u)$
13:        $\mathbf{c}_i^{min} \leftarrow \min(\mathbf{c}_{i+1}, \ ... \ \mathbf{c}_u)$
14:        Status ← 0
15:    end if
16:    if Status = 1 then
17:        normalize($\mathbf{c}_u$)        ▷ see (14)
18:    else
19:        normalize($\mathbf{c}_{i+1}, \ ... \ \mathbf{c}_k$)        ▷ see (14)
20:    end if
21: end procedure
```

The running normalization checks if the new element in the sliding window is smaller or bigger than the current minimum or maximum, or the element that drops out of the window is equal to one of them. If one of these conditions are true, then the new minimum and maximum need to be defined and the whole subsequence needs to be normalized from scratch. Otherwise only the new observation in the window is normalized. Based on the resulting status from the running normalization the local cost matrix **C** is updated by either:

- reusing former results and appending the new column of costs (see (5)), if status = 1, or
- recalculating **C** from scratch if status = 0.

For multivariate time series the update of **C** depends on the normalization status of all dimensions. `rundtw()` also supports the z-normalization $\mathbf{y}_j := \frac{\mathbf{c}_j - \mu_i}{\sigma_i}$, $\forall j = i, ...i+n-1$, where μ_i and σ_i are mean and standard deviation of the subsequence $\{\mathbf{c}_j\}_{j=i,...i+n-1}$. The initial μ_1

and σ_1 are calculated with the standard formulas and for $i > 1$ updated incrementally by:

$$\begin{aligned} \mu_i &= \mu_{i-1} + \frac{(\mathbf{c}_{i+n-1} - \mathbf{c}_{i-1})}{n} \\ \sigma_i^2 &= \sigma_{i-1}^2 + \frac{(\mathbf{c}_{i+n-1}^2 - \mathbf{c}_{i-1}^2)}{n-1} + (\mu_{i-1}^2 - \mu_i^2)\frac{n}{n-1} \\ \sigma_i &= \sqrt{\sigma_i^2}. \end{aligned} \tag{15}$$

Contrary to the min-max normalization, when applying the z-normalization the normalization of the subsequence and the local cost matrix need to be calculated from scratch for each single step. Further, `rundtw()` makes use of early abandoning and lower bounding as proposed originally by Keogh (2002) and refined for multivariate time series by Rath and Manmatha (2002). They define the lower bound as:

$$lb_{mv}(\mathbf{q}, \mathbf{c}) = \sum_{i=1}^{n}\sum_{o=1}^{O} \begin{cases} \text{local\_dist}(\mathbf{c}_{i,o}, u_{i,o}) & \text{if} \quad \mathbf{c}_{i,o} > u_{i,o} \\ \text{local\_dist}(\mathbf{c}_{i,o}, l_{i,o}) & \text{if} \quad \mathbf{c}_{i,o} < l_{i,o} \\ 0 & \text{else} \end{cases} \tag{16}$$

where the local_dist function can be any of the predefined local distance functions ('norm1', 'norm2', 'norm2_square', see (13)) and the lower and upper bounds l and u are defined by:

$$\begin{aligned} u_{io} &:= max(\mathbf{q}[i - r : i + r, o]) \\ l_{io} &:= min(\mathbf{q}[i - r : i + r, o]). \end{aligned} \tag{17}$$

The parameter r depends on the applied warping path restrictions, applying the Sakoe Chiba warping window r is equal the window size parameter. Rath and Manmatha (2002) prove that for multivariate time series of equal lengths the lower bound is always smaller or equal to the DTW distance: $lb_{mv}(\mathbf{q},\ \mathbf{c}) \leq \text{DTW}(\mathbf{q},\ \mathbf{c})$ for the step pattern "symmetric1". This fact can be used to decide whether or not to calculate the DTW distance between two time series and so speed up pattern recognition algorithms as runDTW. Algorithm 4 describes the algorithm runDTW in detail. The following section discusses how to adjust the algorithm to perform k-NN search.

3.5. k-NN search with DTW

In general the k nearest neighbors of one object ϕ^* are those k objects ϕ_i of a given set of objects $\Phi = \{\phi_j\}_{j=1}^{J}$, where $d(\phi^*, \phi_i) < d(\phi^*, \phi_j)$, $\forall i \in I \subseteq J$ and $\forall j \in J \setminus I$, and $d()$ is a distance function and $|I| = k$. For the special case of finding k-NN in a long time series $\mathbf{c}$, we do not want to return trivial fits, since analyzing trivial fits is meaningless as discussed by Keogh and Lin (2005). They define trivial fits $(i^* \pm j)$ as follows: Given the function $d(i) = \text{DTW}(\mathbf{q},\ \mathbf{c}[i : i + n])$, and the best found fit i^* such that $d(i^*) < d(i)\ \forall i$, then it is very likely that the values $d(i^* + j)$ and $d(i^* - j)$ for any small additive term j have similarly low distances as well. To ensure that the found k-NN are no trivial fits we need to guarantee that two consecutive fits i_1^* and i_2^* do not overlap: [1] $|i_1^* - i_2^*| \geq n$.
The algorithm runDTW detects multiple non-trivial fits to a query pattern and saves time by avoiding computing unnecessary distances. Two modifications of Alg. 4 are necessary to fulfill the search of the k nearest non-overlapping neighbors:

[1]Per default the function `rundtw()` does not accept any overlap, but the user may relax this condition with care by setting the parameter `overlap_tol`.

Algorithm 4 Sliding Window approach to detect multiple fits of a query pattern **q** in a longer time series **c**

```
1:  procedure RUNDTW($\mathbf{q} \in \mathbb{R}^{n \times o}$, $\mathbf{c} \in \mathbb{R}^{m \times o}$, $w, \delta$)
2:      $x \leftarrow$ return vector of length $m - n + 1$
3:      $x^* \leftarrow \delta$                                   ▷ best-sofar-value within the range of the $w$
4:      $j^* \leftarrow 1$
5:      $j \leftarrow 1$
6:      $j_{upper} \leftarrow j + n - 1$
7:      y $\leftarrow$ incnorm($\mathbf{c}$, $j - 1$, $j_{upper}$)                          ▷ normalization
8:      $\mathbf{C}[1:n,\ j:k] \leftarrow$ dist(y, $\mathbf{q}$)        ▷ update $\mathbf{C}$ based on local distance function
9:      $lb \leftarrow$ lowerbound($\mathbf{q}$, $y$, $w$)                                  ▷ see (16)
10:     if $x^* \leq lb$ then              ▷ break since best-sofar-in-range is smaller than $lb$
11:         $x[j] \leftarrow$ NAN
12:         $j \leftarrow j + 1$ and go back to 6
13:     else
14:         $x[j] \leftarrow$ dtw2vec($\mathbf{q}$, $\mathbf{c}[j : j_{upper}]$, $w$, $x^*$)      ▷ use $x^*$ as early abandon threshold
15:         if $x[j] < x^*$ then
16:             $x^* \leftarrow x[j]$
17:             $j^* \leftarrow j$                                    ▷ index of best value in range
18:         end if
19:     end if
20:     if $j - j^* > n$ then                                  ▷ guarantees no overlap of fits
21:         $x^* \leftarrow \delta$                             ▷ resets the best-sofar-value/ threshold
22:     end if
23:     if $j \leq m - n + 1$ then
24:         $j \leftarrow j + 1$ and go back to 6
25:     end if
26:     return $x$
27: end procedure
```

1 For the threshold adjustment in Line 21 of Alg. 4 we need to keep track of the best k found fits so far and their positions to prevent overlapping fits. The best k found fits so far serve as thresholds for lower bounding and early abandoning. However, some nearest neighbors could be overlooked during the computation process. The second point solves this.

2 runDTW picks the better of two consecutive candidate fits (i and j, $j > i$) which have an overlap. If the fit at position j is better, then i is dropped, no matter how small the DTW distance at i is. If this happens multiple consecutive time, it is possible that one of the actual k nearest neighbors is overlooked. To take care of this run-away problem we append an algorithm that steps through the vector of calculated distances in reverse order (Alg. 5).

Algorithm 5 steps through the distance vector from end to beginning and remembers the best so far index i^* with lowest distance d^*. If the previous $(n-1)$-many entries are bigger than d^* then i^* is appended to the result vector. The algorithm doesn't care about the run-away problem since the $(n-1)$-many entries after a found fit are NaN, due to the skipping or early abandoning in Line 11 and 14 of Alg.4.

Algorithm 5 Step through the vector of DTW distances in reverse order to find the non-overlapping k-NN

```
1:  procedure REV-KNN(distance vector $\mathbf{d} \in \mathbb{R}^{m-n}$, window size $n = |\mathbf{q}|$)
2:      $kNN \leftarrow$ empty vector
3:      $d^* \leftarrow \mathbf{d}_{m-n}$                      ▷ initiate the best-so-far distance value
4:      $i^* \leftarrow m - n$
5:      for $i = m - n - 1 : 1$ do             ▷ step through distance vector in reverse order
6:          if $n \leq i^* - i$ then                          ▷ guarantees no overlap
7:              $kNN \leftarrow append(kNN, i^*)$                 ▷ remember the index $i^*$
8:              $d^* \leftarrow \mathbf{d}_i$
9:              $i^* \leftarrow i$
10:         else if $\mathbf{d}_i < d^*$ then
11:             $d^* \leftarrow \mathbf{d}_i$
12:             $i^* \leftarrow i$
13:         end if
14:     end for
15: return $kNN$
16: end procedure
```

In the following we showcase the functionality of `rundtw()` for a sine wave query pattern **q**, and a long time series **c**, which is a concatenation of random walks and deformed representations of **q**. We simulate the deformation by first simulating a warp, and then shifting and scaling by random:

```
R> set.seed(1234)
R> rw <- function(nn) cumsum(rnorm(nn))
R> deform <- function(x, p){
+    (simulate_timewarp(x, p, preserve_length = TRUE) + rnorm(1, 0, 3)) *
+      abs(rnorm(1, 0, 3))
+  }
R> Q <- sin(seq(1, 20, length.out = 100))
R> C <- c(rw(100), deform(Q, 0.3), rw(10),  deform(Q, 0.3), rw(200),
+    deform(Q, 0.3), rw(300))
R> rundtw( Q = Q, C = C, normalize = "01", dist_method = "norm1",
+    ws = 10, threshold = NULL, lower_bound = TRUE, k = 3 )

counter:
      norm_reset norm_new_extreme       norm_1step         cm_reset
              70               82              659              137
        cm_1step    early_abandon      lower_bound        completed
             534              628              140               43

Indices of k nearest neighbors knn_indices:
[1] 101 511 211
```

```
Distances of k nearest neighbors knn_values:
[1] 0.7101164 0.9022618 2.1050072
```

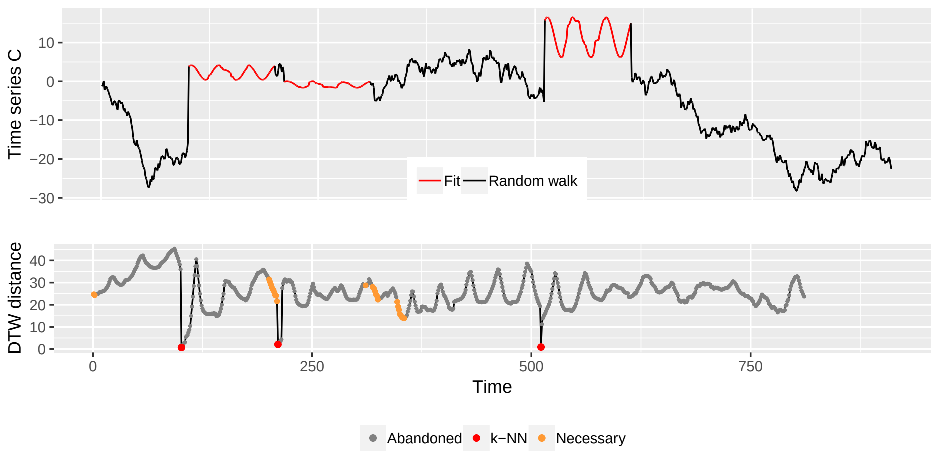


Figure 5: Find the k-NN (here k = 3) of a query pattern **q** (sine wave) in a long time series **c**. The upper graph shows the time series and the fits, the simulated warped representations of **q** (red). The bottom graph shows the vector of DTW distances for all subsequence: skipped or abandoned due to early abandoning or lower bounding (grey), the necessary ones (orange), and the k-NN (red).

As supposed the algorithm runDTW detects the k-NN to be located at the indices where the deformed versions of **q** very put into **c**: 101, 511 and 211. Also the respective DTW distances of the normalized subsequences to the query pattern **q**are printed.

Figure 5 shows time series **c** (top) and the DTW distances computed with `rundtw()` (bottom). There are $811 = 910 - 100 + 1 = m - n + 1$ possible combinations that the starting index i can take to calculate DTW($\mathbf{c}[i : i + 100 - 1]$, $\mathbf{q}$). Of these 811 steps, the runDTW algorithm needs to reset the minimum and maximum of the sliding window (see Lines 12 and 13 in Alg. 3) 70 times, and 82 times the new observation in the sliding window is smaller than the current minimum or bigger then the current maximum (Lines 3 to 10 in Alg. 3). In these cases the complete window needs to be normalized. In the remaining 659 times only the new observation in the sliding window is normalized. (Lines 17 and 19 in Alg. 3). The incremental update of the local cost matrix **C** depends on the exit status of the normalization and on the lower bounding. If the calculation is skipped due to the lower bound (here 140 times), **C** is not touched at all. For the remaining 671 cases, either one column of **C** is updated (534 times), or the complete reset of the matrix was necessary (137). Apart from lower bounding, early abandoning also reduces the computation time. Here 628 times the computation of DTW is abandoned due to hitting the threshold. So, only for 43 out of 811 starting indices the complete DTW computation was performed. These are the orange and red points in Fig. 5. The red points are the $k = 3$ nearest neighbors (to plot the grey points we did the calculation again with the settings `lower_bound = FALSE` and `k = NULL`). As soon as runDTW finds the third NN, the maximum of the distances of the three NN is set as threshold for lower bounding and early abandoning. For this reason, no DTW calculation is finished beyond the index 511.

Section. 4.4 demonstrates the computation time benefit of `rundtw()` contrary to traditional solutions.

4. Applying IncDTW

This section demonstrates the applicability of **IncDTW** in different fields of time series data mining:

- **Clustering** and **prototypical patterns**: The DTW distance measure is often applied for clustering or classifying a database of observed time series which have similar lengths. Section **??** demonstrates the calculation of a matrix of pairwise DTW distances, the clustering based on this distance matrix and how to calculate a representative – a prototypical pattern – of a cluster of time series of different lengths and non-linearly aligned.
- **Classification** of **live data streams**: Section 4.1 discusses a time series classification task for live data streams solved by either the traditional DTW implementation `dtw2vec()` or the incremental updating of DTW distances to speed up calculations with `idtw2vec()`.
- **Pattern recognition**: Scanning longer time series to detect similar representations of query patterns is especially challenging when the representations can vary in speed and time. DTW is a suitable distance measure to detect such representations, but expensive to calculate. The incremental calculation algorithm can save computation time to solve the problem of matching a query pattern. Section 4.2 demonstrates the key principle on a simple example and based on these principles Sec. 4.3 discusses an exemplary algorithm to navigate in the plane of possible fits.

Finally Sec. 4.4 compares the run times for different DTW implementations, packages and use cases.

In the following experiments we work with data sets (Bruno, Mastrogiovanni, Sgorbissa, Vernazza, and Zaccaria 2013) downloaded from UCI machine learning repository (Dheeru and Karra Taniskidou 2017). The data was collected by participants wearing a smart watch recording a 3-dimensional accelerometer signal with a sampling rate of 32 Hz. Among other actions the participants were asked to collect data during walking (`Walk`), drinking a glass (`drink_glass`) and brushing teeth (`brush_teeth`). The time series data of these experiments are included in the package **IncDTW**.

4.1. Incremental DTW update for live data

When applying data mining methods on live streams of data, it is mandatory that the computation time of the analysis is smaller than the time in between two consecutive observations. In this experiment we simulate the situation of dealing with data streams by iteratively including more observations of the time series into analysis. As soon as new observations are 'recorded' we classify the time series streams by comparing their DTW distances to prototype patterns, so we need to update the DTW calculation for each set of new observations.

For this experiment we use the accelerometer time series `Walk`, `drink_glass` and `brush_teeth`. For each activity we determine representative centroid patterns with `IncDTW::dba()`. Next we calculate the initial DTW distances for the first 100 observations (about 3 seconds) of each time series of the three data sets to the three centroids. Then we simulate the continuous recording of new observations and apply `idtw2vec()` to update the DTW distance measures, which requires to store the last columns of **G** (see (2)) of the previous calculations. For comparing the computation times we fulfill the same classification task with `dtw2vec()`, and of course the classification results are identical. Figure 6 depicts this simulation of a data

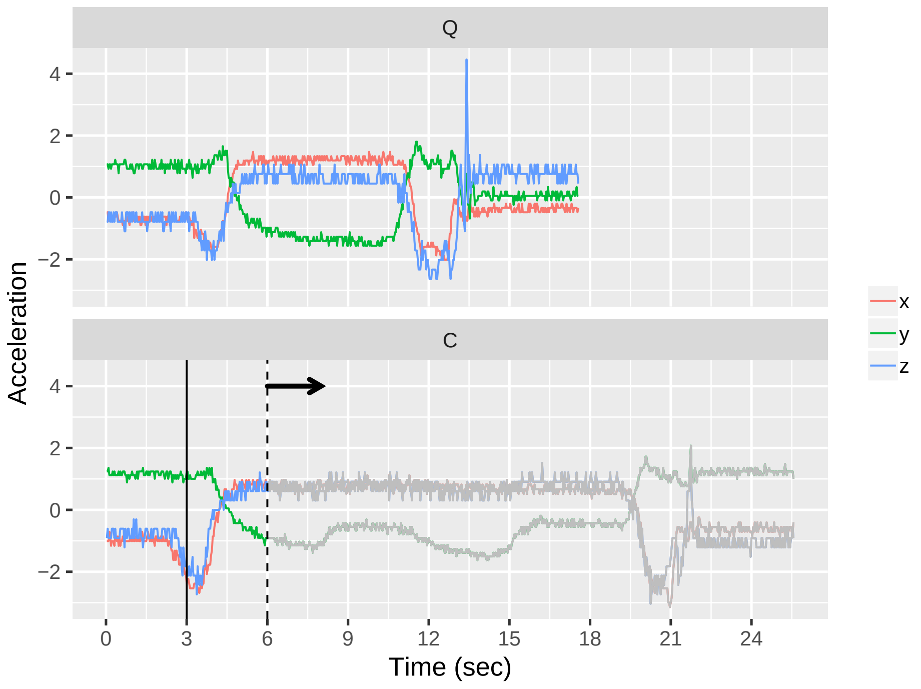


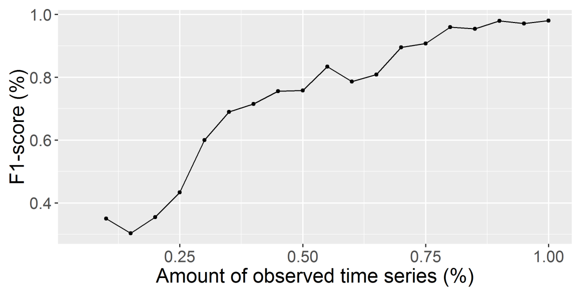
Figure 6: Iteratively increasing the observation window. As the dashed line moves to the right, more data is included in analysis and the DTW alignment is updated for the new observations.

stream **c** and the query time series **q**, both selected from the `drink_glass` data set. This plot shows the situation after the initial step – the first three seconds are already observed (vertical solid line) – when **c** has already been recorded for six seconds in total (the vertical dashed line). As the data stream continuously updates the dashed line moves to the right and more observations are included to the DTW alignment with **q**.

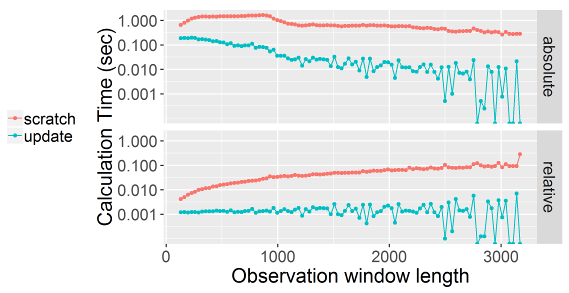
Figure 7a plots the classification accuracy against the 'observed' (used) percentage of the time series, and shows that the accuracy increases the more observations are recorded. Already about 75% are enough to reach an F1-score of 90%. We used 4-fold cross validation, where we trained the representatives on one fold and classified the remaining 3 folds. Figure 7a shows aggregated results.

Figure 7b compares the computation times of `idtw2vec()` (incremental) and `dtw2vec()` (from scratch) to process one set of new observations, which we represent as the set of observations recorded within one second, so 3-dim time series with 32 rows (since originally recorded with 32Hz). The collection of these three data sets consists of 212 time series of different lengths. The calculation times depend on the length of the observation window and the number of time series that are at least as long as the observation window. Since the time series are of different lengths, with increasing observation window, more and more time series can not be processed further until the observation window is equal to the length of the longest time series. For this reason the graph for 'scratch' in Fig. 7b (top) first rises and then drops continuously. All time series are at least 187 observations long and beyond this observation window length the shorter time series drop out of further analysis and so are not relevant for the total computation time. For clarification we also plot the relative times per time series in Fig. 7b (bottom). It is worth mentioning that the y-axis are log-scaled.

We conclude that the incremental update can process about 7 to 108 times more time series than the calculation from scratch, dependent on the length of the time series, the observation window respectively. This exemplary data analysis task would not be solvable in time by applying `dtw2vec()` (which is vector-based implemented in C++ via **Rcpp**) since the calculation of DTW distances and classification takes longer than one second, which is the time in-between two sets of new observations. However, the incremental method with `idtw2vec()` is capable. As expected this experiment demonstrates the calculation time for the incremental step to be independent of the total length of the time series, see Fig. 7bc. We performed this experiment applying a single core of a 2.8 GHz and 16GB RAM laptop. If we split the work for this example among a few couple of cores `dtw2vec()` would manage the classification in time as well, however the relation of 7 to 108 remains the same, so the incremental solution is capable to deal with much more time series updates in less time.



(a) Prediction Accuracy.



(b) Runtime: Absolute (top) for all time series, and relative (bottom) per time series.

Figure 7: Prediction accuracy and computation time comparison for classifying multivariate time series of the data sets `Walk`, `drink_glass` and `brush_teeth` by simulating to observe these time series live and update the prediction once per second.

4.2. Moving in the plane of possible fits

This section demonstrates by means of a simple example the key principles how the functions of **IncDTW** help to navigate in the plane of fits, to detect a warped instance of a query pattern in a longer time series. Section 4.3 presents an exemplary heuristic that applies these functions. We presented the more elaborate Caterpillar algorithm in Leodolter *et al.* (2018), which relies on the same principles.

Before we start with the example we give an outlook how we will use the functions of **IncDTW**. Say $1 \leq a \leq b \leq m = |\mathbf{c}|$ and we just have initialized the alignment of $\mathbf{q}$ and the section of $\mathbf{c}$ at the point $Y = (a, b)$ in Fig. 1b, so `initialize_plane(q, c[a:b])`. This command calculates the DTW distance of the respective time series and returns an object of class `planedtw` (see

Sec. 2.2 and 3.3). To adjust (a, b) and detect an index combination of lower normalized DTW distance, we make use of two different views on the problem:

- The regular view: The incremental calculation (via `idtw2vec()` or `increment()`) in combination with the open end alignment (via `dtw_partial()` or `decrement()`, discussed in more detail in Appendix **??**) facilitates the evaluation of the extension and contraction of the scanning window at the end (index b) while the start (index a) is fixed. (see vertical arrows in Fig.1b)
- The reverse view: The same logic as for the regular view is applied on $\mathbf{q}$ and $\mathbf{c}$ in reverse time order to evaluate the extension and contraction of the scanning window at the start index a while fixing the end index b (see horizontal arrows in Fig.1b). The function `reverse()` helps to reverse the order of the `planedtw` object.

The reverse view makes use of the fact that DTW is reversible as proved by Assent, Wichterich, Krieger, Kremer, and Seidl (2009) for the step pattern "symmetric1". Appendix **??** discusses deviations of the regular and reverse view for "symmetric2" and shows empirically that those are minor. For the following examples in Sec. 4.2 and 4.3 we demonstrate the results achieved with the default step pattern "symmetric2", and we also accomplished all experiments with "symmetric1" and the results are almost identical.

To start with the example, first we simulate the time series $\mathbf{q}$ as random walk. Next we set $\mathbf{c}$ as copy of $\mathbf{q}$ and add random noise. To simulate a time warp we compress $\mathbf{c}$ for the following example (in Sec. 4.3 we apply a simple heuristic relying on the same principles on the `drink_glass` data, covering longer and shorter instances of the same pattern). Finally we append additional random walks at the beginning and end of $\mathbf{c}$, representing noise.

```
R> set.seed(213)
R> Q <- rw(500)
R> C <- Q + rnorm(length(Q))
R> C <- IncDTW::simulate_timewarp(C, stretch = 0, compress = 0.1)
R> C <- c(rw(100), C, rw(100))
R> length(C)

[1] 650
```

Figure 8 depicts the exemplary time series, $\mathbf{q}$ at the top and the simulated $\mathbf{c}$ in the middle. $\mathbf{c}$ has a length of 650, where the first and last 100 observations were simulated as noise without relation to $\mathbf{q}$ (green parts of the graph). The purple section at the middle of $\mathbf{c}$, at a length of 450, represents the compressed noisy instance of $\mathbf{q}$.

A simple but effective heuristic – that applies the functions of **IncDTW** as kind of vehicles – is incrementing and testing partial alignments until no further reduction of the normalized distance can be achieved and the partial alignment does not change for further incremental steps. Then the best partial alignment of the reverse time series is used to omit initial noise.

We start with the initial DTW alignment of $\mathbf{q}$ and the first `nQ = n` observations of $\mathbf{c}$.

```
R> nQ <- length(Q)
R> x <- initialize_plane(Q, C[1:nQ])
R> dtw_partial(x, partial_Q = FALSE, partial_C = TRUE)
```

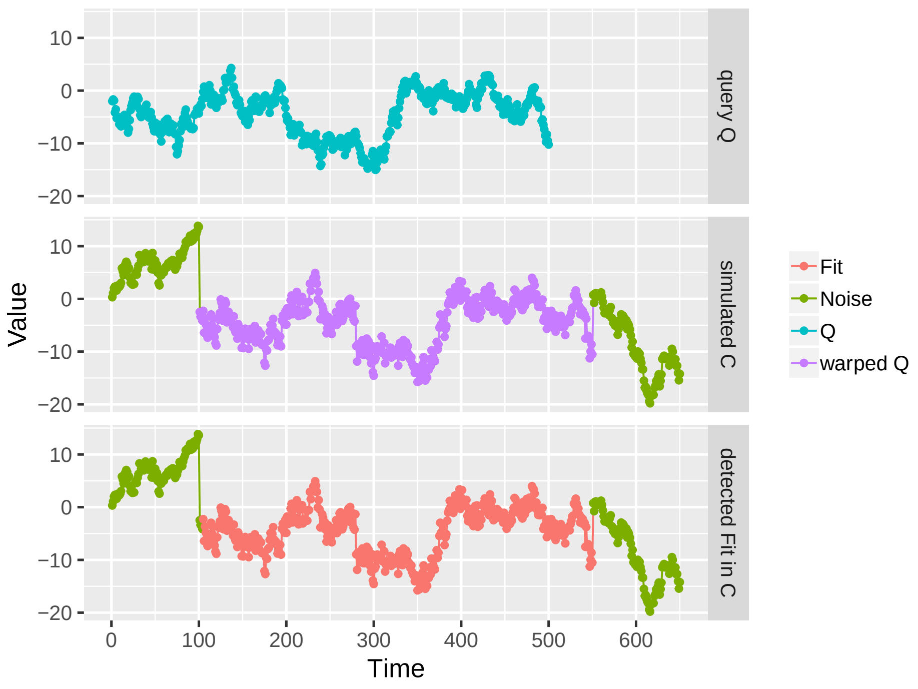


Figure 8: Example for a query pattern **q** (top), a noisy time warped candidate pattern **c** with additionally appended noise at the beginning and end (middle), and the detected fit (bottom).

```
$rangeQ
[1]   1 500

$rangeC
[1]   1 500

$normalized_distance
[1] 1.335524
```

The function `dtw_partial()` returns 500 for the end index of **c**, which is a full alignment for the considered range. Next we start the incremental steps with a step width of 25. In each step the results of the previous step (`x`) is recycled by `increment()`, especially the required parts of the global cost matrix (compare Fig. 4).

```
R> step <- 25
R> #---- first step ----
R> x <- increment(x, newObs = C[(nQ + 1):(nQ + step)])
R> dtw_partial(x, partial_Q = FALSE)

$rangeQ
[1]   1 500
```

```
$rangeC
[1]   1 519

$normalized_distance
[1] 1.302147

R> #---- second step ----
R> x <- increment(x, newObs = C[(nQ + step + 1):(nQ + step * 2)])
R> dtw_partial(x, partial_Q = FALSE)

$rangeQ
[1]   1 500

$rangeC
[1]   1 550

$normalized_distance
[1] 1.253576

R> #---- third step ----
R> x <- increment(x, newObs = C[(nQ + step * 2 + 1):(nQ + step * 3)])
R> par <- dtw_partial(x, partial_Q = FALSE); par

$rangeQ
[1]   1 500

$rangeC
[1]   1 550

$normalized_distance
[1] 1.253576

R> x <- decrement(x)
```

At the third step we test the alignment of $\mathbf{q}$ and $\mathbf{c}[1 : 575]$, and the best partial alignment has not changed to the previous step with a maximum index of 550. So we `decrement() x` and continue with the partial alignment of the reverse time series:

```
R> x <- reverse(x)
R> dtw_partial(x, partial_Q = FALSE, reverse = TRUE)

$rangeQ
[1]   1 500

$rangeC
[1] 105 550

$normalized_distance
[1] 0.4919826
```

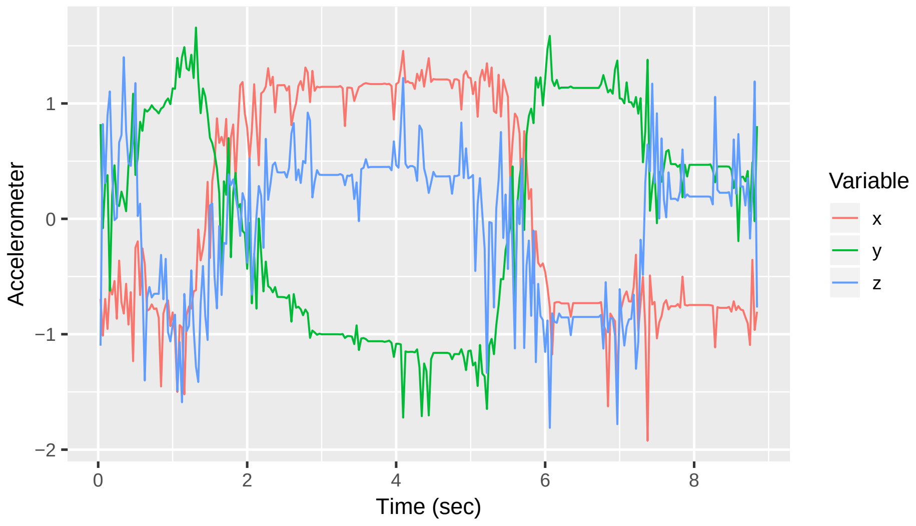


Figure 9: Barycenter of drinking glass records as query pattern to demonstrate simple and powerful pattern recognition algorithms.

```
R>
```

Finally we detect the best fit of the query pattern **q** in **c** from 105 to 550, which is very close to the simulated ground truth of 101 and 550. The bottom plot of Fig. 8 shows the detected fit in red. One can also observe that the normalized distance always decreased from one step to the next.

4.3. Incremental DTW heuristic

In this section we design a simple heuristic that uses the functions of **IncDTW** to answer the following question: What's the average duration people need to empty the glass, so the time in-between lifting the glass and putting it back on the table? Or analogously, when is the start and end of the actual drinking, and not the recording? These questions are examples for the generic question: When does the event XY start and end within a record? With this example we demonstrate how to build an algorithm for the 1-SS^{vl} problem (see (10)), that applies the tools provided by **IncDTW**, and is designed to detect specific patterns in your data.

First we plot the barycenter (calculated with `dba()`) of all time series of `drink_glass` in Fig.9.

```
R> dba0 <- dba(drink_glass, dist_method = "norm2")
```

It's not trivial to set limits by hand when the actual drinking starts and ends, but we define lower and upper bounds of second 1.5 and 6. In-between these borders the drinking most certainly takes place, and the amount of additional recorded hand moving action is reduced.

```
R> Q <- dba0$m1[(1.5*32):(6*32),]
```

Once we have the query pattern **q**, we need a tool to detect it in each of the time series **c** of `drink_glass`. So we define two different algorithms, (1) a brute force method that tries out each alignment of partial **c** and **q**, and (2) an algorithm that updates the DTW calculation incrementally similarly as demonstrated in Sec. 4.2. The brute force algorithm finds the minimum of (18) by simply trying out each combination.

$$(i^*, j^*) = \underset{i,j}{\operatorname{argmin}} \ \text{DTW}(\mathbf{q},\ \mathbf{c}[i:j]) \tag{18}$$

For the two time series **q** of length n and **c** of length m, and a fixed $i \in \{1, ...m\}$, the brute force method covers all combinations (i, j) $\forall j \in \{i, ...m\}$ by calculating DTW($\mathbf{q}$, $\mathbf{c}[i : m]$) and comparing all entries in the last row of the global cost matrix, after the appropriate normalization. In the plane of possible fits (see Fig. 1) this corresponds to all points on a vertical line starting at the point (i, i) .

```
R> getBruteFit <- function(C, Q){
+    nC <- nrow(C)
+    tmp <- lapply(1:nC, function(i){
+      x <- idtw2vec(Q = Q, newObs = C[i:nC,, drop = FALSE],
+        dist_method = "norm2")
+      x_par <- dtw_partial(x, partial_Q = FALSE)
+      c(i, x_par$rangeC, x_par$normalized_distance)
+    })
+    tmp <- do.call(rbind, tmp)
+    min_ix <- which.min(tmp[,4]) # 4th column is the normalized distance
+    rangeC_0 <- tmp[min_ix, 1]
+    rangeC_1 <- tmp[min_ix, 1] + tmp[min_ix, 3] - 1
+    return(c(rangeC_0, rangeC_1))
+  }
```

The advantage of the brute force algorithm is that it certainly detects the fit of minimal normalized DTW distance, so the section of **c** that matches best to the query pattern **q**. The disadvantage is of course the run time. For bigger problems, with many time series and long time series, this is not feasible. For this reason the functions of **IncDTW** can deal as components for a heuristic that approximates the best solution in reasonable time. The following heuristic `getFit()` simply increases the scanning window by incrementing j by the step size `step` until no reduction of the DTW distance within this new range can be found. Finally the DTW distance of the reversed time series is calculated and `dtw_partial()` finds the best partial alignment of **c**, so the best i.

```
R> getFit <- function(C, Q, i = 1, j = nrow(Q), step = 25){
+    # initial value
+    best_j_sofar <- j
+    x <- initialize_plane(Q = Q,
+      C = C[i:j, ,drop = FALSE], dist_method = "norm2")
+
+    # step
+    increase_window <- TRUE
```

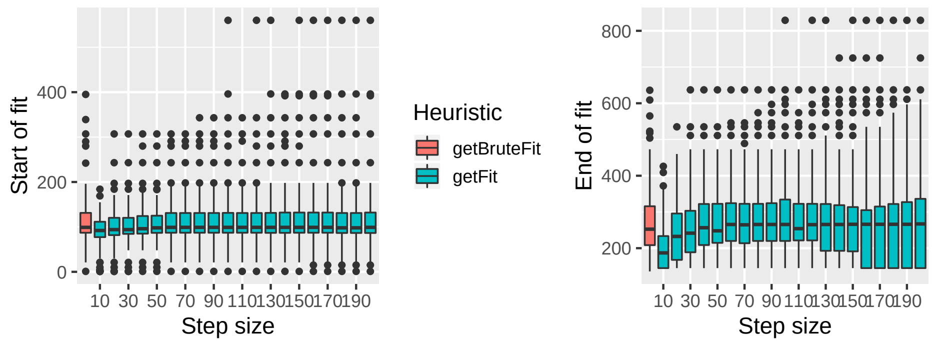


Figure 10: Results of incremental algorithm to detect drinking pattern.

```
+     while(increase_window){
+       j_step <- min(j + step, nrow(C))
+       x <- increment(x, newObs = C[(j + 1):(j_step), , drop=FALSE])
+       x_par <- dtw_partial(x, partial_Q = FALSE)
+
+       # check
+       if(x_par$rangeC[2] <= best_j_sofar | j_step == nrow(C)){
+         increase_window <- FALSE
+       }else{
+         best_j_sofar <- x_par$rangeC[2]
+         j <- j_step
+       }
+     }
+
+     # reverse step
+     x <- decrement(x, nC = best_j_sofar) %>% reverse()
+     x_par <- dtw_partial(x, partial_Q = FALSE, reverse = TRUE)
+     return(c(x_par$rangeC[1], x_par$rangeC[2]))
+   }
```

With the following two statements we apply the two algorithms `getBruteFit` and `getFit` onto the set of 3-dim time series `drink_glass`.

```
R> bf_fits <- lapply(drink_glass, getBruteFit, Q)
R> fits <- lapply(drink_glass, getFit, Q)
```

We varied the parameter `step` for `getFit()` and plot the results in Fig. 10, which shows the boxplots of the distributions of the first (left plot) and last (right plot) indices of the detected fits. It seems that if the step parameter is set too small, the heuristic stops in some cases too early. Beginning with a step size of 30 to 40 the distributions stabilize and are almost identical to the fits detected by the brute force algorithm. As a refinement step we add the following lines at the end of `getFit()`, after the reverse step. These lines are equivalent to the reverse step, but in the opposite direction. We name the new function `geFit2()`.

```
R> iC <- x_par$rangeC[1]
```

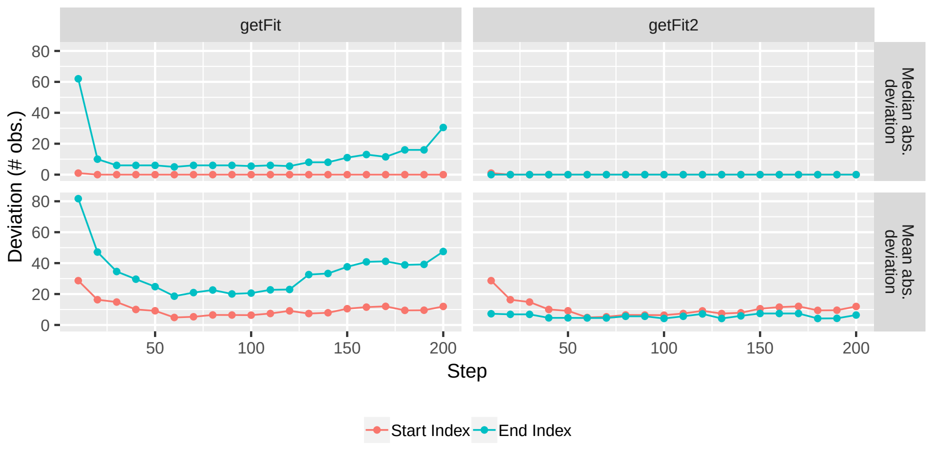


Figure 11: Deviations of the detected start and end indices of the two algorithms `getFit()` and `getFit2()` to the results of `getBruteFit()`. The red line for median errors of `getFit2()` is overlaid by the blue one.

```
R> x <- decrement(x, nC = x$control$nC - iC + 1) %>% reverse() %>%
+    increment(., newObs = C[(best_j_sofar + 1):nrow(C), ])
R> x_par <- dtw_partial(x, partial_Q = FALSE, reverse = FALSE)
```

Figure 11 summarizes the deviations of the two functions `getFit()` and `getFit2()` from the brute force solution. The two rather simple heuristics show a good approximation of the best solutions, especially for detecting the start index. We measured the deviations with the mean and median absolute deviation since there are some outliers, as Fig. 10 also shows. Next we compare the results and run time for two time series of `drink_glass`, we pick the longest and shortest:

```
R> sC <- drink_glass[[which.min(lengths(drink_glass))]]
R> lC <- drink_glass[[which.max(lengths(drink_glass))]]
R> rbind(getBruteFit(sC, Q), getFit(sC, Q), getFit2(sC, Q))

     [,1] [,2]
[1,]   65  179
[2,]   65  179
[3,]   65  179

R> rbind(getBruteFit(lC, Q), getFit(lC, Q), getFit2(lC, Q))

     [,1] [,2]
[1,]  131  523
[2,]  131  535
[3,]  131  523
```

We see that the results differ only marginally. Table 2 summarizes the computation times measured with `microbenchmark()` for 100 runs, and shows that the brute force method takes about 77 to 424 times longer than `getFit()`.

	shortest		longest	
heuristic	median	relative	median	relative
getBruteFit	96.60	76.72	1687.85	424.10
getFit	1.26	1.00	3.98	1.00
getFit2	1.69	1.35	6.16	1.55

Table 2: Median (in millisec) and relative run times for the shortest and longest glass drinking sample.

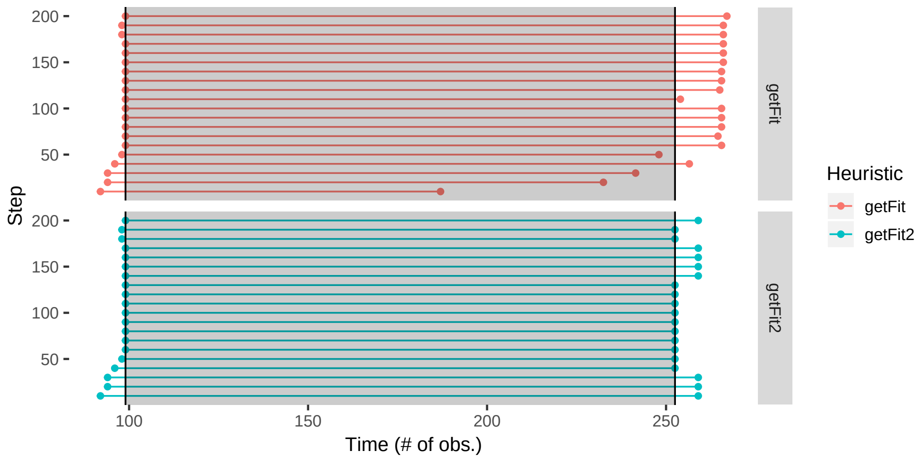


Figure 12: Median start, end and duration of drinking a glass for different step parameters for the two algorithms `getFit()`. The gray shaded box, bounded by the vertical black lines, shows the start, end and duration of the brute force method.

Finally we answer the initial question of this task: How long lasts the average act of drinking? Figure 12 shows horizontal lines for each step parameter and both heuristics, `getFit()` and `getFit2()`. The start and end of the line indicate the median start and end of the detected fits. The vertical black lines are the respective start and end of the brute force method, which is of course independent of the step parameter. We can see rather stable solutions, especially for `getFit2()`. Table 3 finally compares the results for the fixed step parameter of 50 observations. The median duration of the found fits is 4.8 seconds (153.5 observations), starts at second 3.1 and ends at second 7.9. The standard deviation of the starts and ends detected by `getBruteFit()` are 1.9 seconds (59.9 observations) and 3.2 seconds (101.8 observations), respectively.

Since the found barycenter is shorter than the average record of this data set (8.8 and 12 seconds), it seems reasonable that also the specific pattern of the drinking action is shorter: We extracted a query pattern of 3.5 seconds and the average found fit has a length of 4.8 seconds.

4.4. Run time comparisons

In the following we compare computation times for the 3 data analysis tasks: (1) the incremental update for new observations, (2) single DTW computation for two time series, and (3) computing the matrix of pairwise DTW distances for a set of time series. Table 4 gives

heuristic	start	end	duration
getBruteFit	99 (3.1)	252.5 (7.9)	153.5 (4.8)
getFit	98 (3.1)	248 (7.8)	150 (4.7)
getFit2	98 (3.1)	252.5 (7.9)	154.5 (4.8)

Table 3: Start, end and duration in number of observations (seconds in brackets) per heuristic for step equals 50.

Package	**IncDTW**	**dtw**	**dtwclust**	**rucrdtw**	**parallelDist**
Version	1.0.4	1.20-1	5.4.1	0.1.3	0.2.1
Single DTW	`dtw2vec()`	`dtw()`	`dtw_basic()`	`ucrdtw_vv()`	`parDist()`
Dist matrix	`dtw_dismat()`				`parDist()`
Incr. update	`idtw()`, `idtw2vec()`				
k-NN search	`dtw()`, `rundtw()`				

Table 4: Overview of selected functions per package and computation task for runtime comparisons. Please see Table 1 for more details about the tested packages.

an overview of the tested functions per package and computation task. To compare the calculation times we use the package **microbenchmark**. To the best of our knowledge we set the parameters of the functions to guarantee fair computation time comparisons. So we omitted to return additional output objects (like the warping path) which obviously would cause higher computation times. For runtime comparisons we used a standard laptop computer with 2.8 GHz and 16GB RAM.

Incremental update of DTW

In the previous sections we emphasize and demonstrate the two main ways of applying the incremental DTW calculation, on the one hand for updating existing results for new observations (Sec. 2.1 and 4.1) and on the other hand for incrementally extending a warping window (Sec. 2.2, 4.2 and 4.3). In the following we present run time experiments for the former fields of application. For the latter Tab. 2 compares run times.

The fastest way to compute the DTW distance measure is by recycling former calculated results. For this experiment we simulate the situation of continuously recording new observations and compare the runtime for the incremental calculation with a traditional calculation from scratch. Figure 13a shows the results. Each red point is the median of 100 computations of the DTW distance with `dtw2vec()` of two univariate time series, both of the respective length given at the x-axis. The blue points visualize the median computation time for one incremental step (via `idtw2vec()`), so one new observation of **c**, and **q** of length as given by the x-axis. Both axes are in log scale.

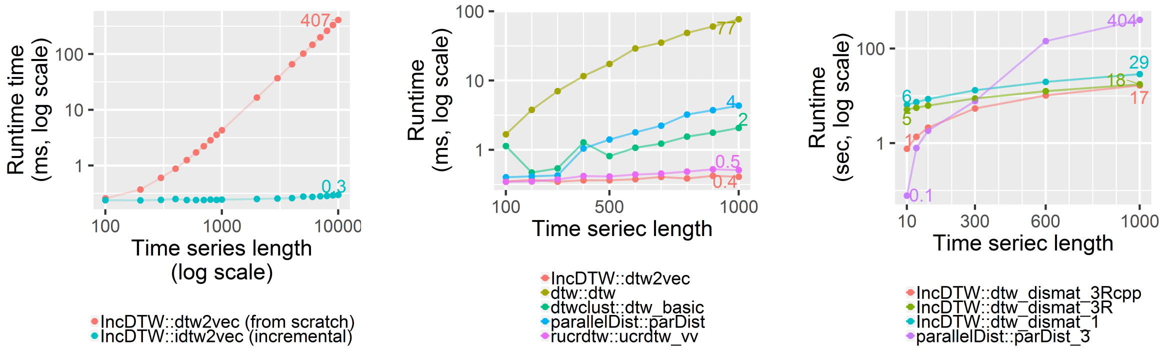
Single computations

Figure 13b depicts the run time comparison of the functions listed in Tab. 4 in a log scale. The only two methods using a vector-based implementations (as discussed in Sec. 3.2) are `rucrdtw::ucrdtw_vv()` and `IncDTW::dtw2vec()`, and these are significantly faster than the remaining functions. To guarantee a fair comparison we set the step pattern to 'symmetric1' (since `rucrdtw::ucrdtw_vv()` only supports 'symmetric1') and the warping window size equal 10 for all functions.

Compute a distance matrix

To cluster or classify a set of time series by their pairwise DTW distance measures we need a distance matrix. The function `IncDTW::dtw_dismat()` helps to get this matrix for a list of univariate or multivariate time series of possibly different lengths. The calculations can be performed single threaded (`ncores = 1`) or multihreaded.

We compare the run times for calculating distance matrices for a set of 500 time series of varying lengths and also set the window size parameter to 10. Figure 13c depicts the run times, where `dtw_dismat_1()` is the standard function `dis_mat()` without parallelization. `dtw_dismat_3Rcpp()` splits the work via **RcppParallel** and `dtw_dismat_3R()` uses the package **parallel**, both with three cores (`ncores = 3`). For short time series `parDist_3()` is up to 10 times faster than `dtw_dismat_3Rcpp()` and for long time series it's the other way round (about 25 times faster).



(a) Incremental vs. from scratch. (b) Single DTW computations for two time series. (c) Matrices of pairwise DTW distances for a list of time series.

Figure 13: Run time comparisons for different data analysis tasks.

k-NN search with runDTW

In this experiment we simulate a long time series $\mathbf{c} \in R^{1000\times 3}$, set the distance metric to 'norm1' and the step pattern to 'symmetric1'. We tested the run time against the following parameters by varying one while holding the others constant. If not varied we set $k = 3$, the number of fits $=$ 3, the global threshold parameter $=$ `Inf`, the length of $\mathbf{q}$ $=$ 50, and the Sakoe Chiba window size $=$ 20.

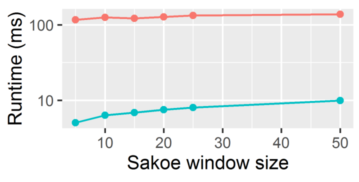
We tested the computation time of `rundtw()` and compared it to the naive basic approach, which computes the DTW for all subsequences applying `dtw2vec()` and looks for k-NN without overlap afterwards. Figure 14 summarizes the runtime results:

- The smaller the window size parameter, the tighter the lower bound and the more calculations are skipped (Fig. 14a).
- The local threshold for skipping or abandoning calculations depends on the global input

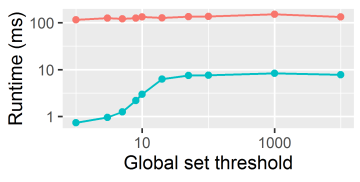
threshold parameter δ, the remaining number of NN to be found and the time steps in-between the current index and the previous best-so-far index. The smaller the global threshold, the more often the calculations are skipped or abandoned. (see Lines 3, 10 and 21 of Alg. 4) (Fig. 14b)

- The lower the parameter k, the earlier the local threshold is set more tight. Prior to finding the k-th NN, the local threshold is set to the minimum of the global input threshold parameter δ and the best value so far in range $x^* = \text{DTW}(\mathbf{q}, \mathbf{c}[j^* : j^*{+}n{-}1])$. If $j - j^* > n = |\mathbf{q}|$, then the local threshold is set equal δ.
  As soon as k non-overlapping NN have been found, the local threshold is set to the minimum of δ and the maximum of the k best-so-far DTW distances, which is in general tighter. (see Lines 3, 10 and 21 of Alg. 4) (Fig. 14c)
- The longer the query pattern $\mathbf{q}$, the longer it takes to hit the threshold and early abandon. (Fig. 14d)
- The more fits we simulate to be hidden in $\mathbf{c}$, the more frequent the algorithm finds a low threshold and skips or abandons unnecessary calculations. However, even if we simulate 0 fits hidden in $\mathbf{c}$, `rundtw()` is 10 times faster than the baseline. (Fig. 14e)

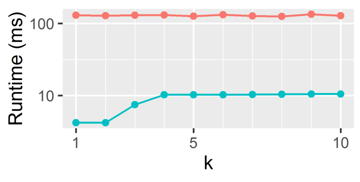
Over all experimental settings `rundtw()` is on average 30 times (ranging from 6 to 133) faster than the naive approach to find the k-NN.



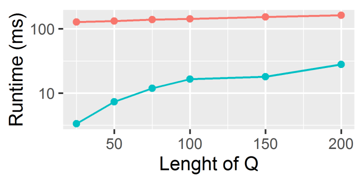
(a) Run time vs. window size



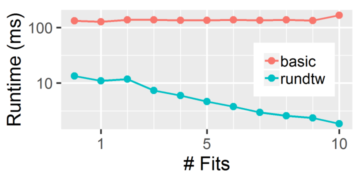
(b) Run time vs. threshold



(c) Run time vs. k



(d) Run time vs. query length



(e) Run time vs. number of fits

Figure 14: Comparison of the run time of `rundtw()` and the baseline, dependent on the parameters ws, δ and k, and on the data characteristics.

5. Conclusion

In this paper we discuss the incremental calculation of the widely applied DTW distance measure and different forms of subsequence matching (Fu 2011) with the plane of possible fits. We present the R package **IncDTW** – current version 1.1.2 available on the Comprehensive R Archive Network `https://CRAN.R-project.org/` – that offers the novel algorithm runDTW

(Sec. 3.4) as solution to the problems of finding the closest subsequences or all subsequences below a threshold and k-NN search (finding the k closest non-overlapping subsequences). In addition, we demonstrate how **IncDTW** and its functions (Sec. 3.3) can be used as a tool set for building customized pattern recognition algorithms (Sec. 4) for the more general problems such as finding fits of varying lengths. Section **??** showcases how to apply **IncDTW** to calculate pairwise DTW distance matrices (parallelized via **RcppParallel** or **parallel**) for clustering a set of time series and how to get cluster representatives.

Due to the quadratic runtime complexity in number of observations of DTW, we put a special emphasis on accelerating our algorithms. Consequently, **IncDTW** transfers the most intensive computations to C++ via **Rcpp** and stresses the principle of the incremental calculation of DTW, by recycling previous calculation results. Further accelerating methods as lower bounding (Keogh, Wei, Xi, Lee, and Vlachos 2006; Rath and Manmatha 2002) and early abandoning methods are also applied.

Future developments will incorporate a parallelized implementation of `dba()` to decrease the runtime. Also, currently the k-NN search within one long time series is covered by `rundtw()`, but for searching the k-NN in a set of time series of equal lengths we will improve `dtw_disvec()` by the same accelerating methods as applied for `rundtw()`.

References

Allaire J, Francois R, Ushey K, Vandenbrouck G, Geelnard M, Intel (2018). ***RcppParallel**: Parallel Programming Tools for **Rcpp**.* R package version 4.4.1, URL `https://CRAN.R-project.org/package=RcppParallel`.

Assent I, Wichterich M, Krieger R, Kremer H, Seidl T (2009). "Anticipatory DTW for Efficient Similarity Search in Time Series Databases." *Proc. VLDB Endow.*, **2**(1), 826–837. ISSN 2150-8097. `doi:10.14778/1687627.1687721`. URL `http://dx.doi.org/10.14778/1687627.1687721`.

Berndt DJ, Clifford J (1994). "Using Dynamic Time Warping to Find Patterns in Time Series." In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, AAAIWS'94, pp. 359–370. AAAI Press. URL `http://dl.acm.org/citation.cfm?id=3000850.3000887`.

Boersch-Supan P (2016). "**rucrdtw**: Fast Time Series Subsequence Search in R." *The Journal of Open Source Software*, **1**, 1–2. `doi:10.21105/joss.00100`. URL `http://doi.org/10.21105/joss.00100`.

Bruno B, Mastrogiovanni F, Sgorbissa A, Vernazza T, Zaccaria R (2013). "Analysis of Human Behavior Recognition Algorithms Based on Acceleration Data." In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 1602–1607. IEEE.

Dheeru D, Karra Taniskidou E (2017). "UCI Machine Learning Repository." URL `http://archive.ics.uci.edu/ml`.

Ding H, Trajcevski G, Scheuermann P, Wang X, Keogh E (2008). "Querying and Mining of Time Series Data: Experimental Comparison of Representations and Distance Measures." *Proceedings of the VLDB Endowment*, **1**(2), 1542–1552.

Eckert A (2017). ***parallelDist**: Parallel Distance Matrix Computation Using Multiple Threads.* R package version 0.2.1, URL `https://CRAN.R-project.org/package=parallelDist`.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. `doi:10.18637/jss.v040.i08`. URL `http://www.jstatsoft.org/v40/i08/`.

Eddelbuettel D, Sanderson C (2014). "**RcppArmadillo**: Accelerating R with High-Performance C++ Linear Algebra." *Computational Statistics and Data Analysis*, **71**, 1054–1063. URL `http://dx.doi.org/10.1016/j.csda.2013.02.005`.

Fu Tc (2011). "A Review on Time Series Data Mining." *Engineering Applications of Artificial Intelligence*, **24**(1), 164–181.

Giorgino T, *et al.* (2009). "Computing and Visualizing Dynamic Time Warping Alignments in R: The **dtw** Package." *Journal of Statistical Software*, **31**(7), 1–24.

Keogh E (2002). "Exact Indexing of Dynamic Time Warping." In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pp. 406–417. Elsevier.

Keogh E, Lin J (2005). "Clustering of Time-Series Subsequences is Meaningless: Implications for Previous and Future Research." *Knowledge and information systems*, **8**(2), 154–177.

Keogh E, Wei L, Xi X, Lee SH, Vlachos M (2006). "LB_Keogh Supports Exact Indexing of Shapes under Rotation Invariance with Arbitrary Representations and Distance Measures." In *Proceedings of the 32nd international conference on Very large data bases*, pp. 882–893. VLDB Endowment.

Keogh EJ, Pazzani MJ (2000). "Scaling Up Dynamic Time Warping for Datamining Applications." In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pp. 285–289. ACM, New York, NY, USA. ISBN 1-58113-233-6. `doi:10.1145/347090.347153`. URL `http://doi.acm.org/10.1145/347090.347153`.

Kwankhoom W, Muneesawang P (2017). "An Incremental Dynamic Time Warping for Person Re-Identification." In *Computer Science and Software Engineering (JCSSE), 2017 14th International Joint Conference on*, pp. 1–5. IEEE.

Leodolter M, Brändle N, Plant C (2018). "Automatic Detection of Warped Patterns in Time Series: The Caterpillar Algorithm." In *2018 IEEE International Conference on Big Knowledge (ICBK)*, pp. 423–431. `doi:10.1109/ICBK.2018.00063`.

Leodolter M, Plant C, Brändle N (2019). "runDTW: An Algorithm to Detect Prototypical Patterns in Long Time Series." Accepted at useR! 2019 Toulouse - France.

Maus V, Câmara G, Appel M, Pebesma E (2019). "**dtwSat**: Time-Weighted Dynamic Time Warping for Satellite Image Time Series Analysis in R." *Journal of Statistical Software*, **88**(5), 1–31. `doi:10.18637/jss.v088.i05`.

Oregi I, Pérez A, Del Ser J, Lozano JA (2017). "On-Line Dynamic Time Warping for Streaming Time Series." In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 591–605. Springer-Verlag.

Rakthanmanon T, Campana B, Mueen A, Batista G, Westover B, Zhu Q, Zakaria J, Keogh E (2012). "Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping." In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 262–270. ACM. doi:10.1145/2339530.2339576. URL http://doi.org/10.1145/2339530.2339576.

Rath TM, Manmatha R (2002). "Lower-Bounding of Dynamic Time Warping Distances for Multivariate Time Series." *University of Massachusetts Amherst Technical Report MM*, **40**.

Rath TM, Manmatha R (2003). "Word Image Matching using Dynamic Time Warping." In *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, volume 2, pp. II–II. IEEE.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Sakoe H, Chiba S (1978). "Dynamic Programming Algorithm Optimization for Spoken Word Recognition." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **26**(1), 43–49. ISSN 0096-3518. doi:10.1109/TASSP.1978.1163055.

Sakurai Y, Faloutsos C, Yamamuro M (2007). "Stream Monitoring under the Time Warping Distance." In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 1046–1055. IEEE.

Sarda-Espinosa A (2018). ***dtwclust**: Time Series Clustering Along with Optimizations for the Dynamic Time Warping Distance*. R package version 5.4.1, URL https://CRAN.R-project.org/package=dtwclust.

Wang X (2011). "A Fast Exact k-Nearest Neighbors Algorithm for High Dimensional Search Using k-Means Clustering and Triangle Inequality." In *The 2011 International Joint Conference on Neural Networks*, pp. 1293–1299. IEEE.

Yeh CCM, Zhu Y, Ulanova L, Begum N, Ding Y, Dau A, Silva D, Mueen A, Keogh E (2016). "Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets." pp. 1317–1322. doi:10.1109/ICDM.2016.0179.

Affiliation:

Maximilian Leodolter
Austrian Institute of Technology
Center for Mobility Systems
1210 Wien, Austria
E-mail: maximilian.leodolter@ait.ac.at

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd