Large Scale and Streaming Time Series Segmentation and Piece-Wise Approximation Extended Version

Ran Gilad-Bachrach and Johan Verwey

May 28, 2016

Abstract

Segmenting a time series or approximating it with piecewise linear function is often needed when handling data in the time domain to detect outliers, clean data, detect events and more. The data varies from ECG signals, traffic monitors to stock prices and sensor networks. Modern data-sets of this type are large and in many cases are infinite in the sense that the data is a stream rather than a finite sample. Therefore, in order to segment it, an algorithm has to scale gracefully with the size of the data. Dynamic Programming (DP) can find the optimal segmentation, however, the DP approach has a complexity of $O\left(T^2\right)$ thus cannot handle data-sets with millions of elements, nor can it handle streaming data. Therefore, various heuristics are used in practice to handle the data.

This study shows that if the approximation measure has an inverse triangle inequality property (ITIP), the solution of the dynamic program can be computed in linear time and streaming data can be handled too. The ITIP is shown to hold in many cases of interest. The speedup due to the new algorithms is evaluated on a variety of data-sets to be in the range of $8 - 8200x$ over the DP solution without sacrificing accuracy. Confidence intervals for segmentations are derived as well.

1 Introduction

The problem studied in this work is the following: given time series data, find an approximation to the observed data which is made of few simple constructs that are concatenated together. That is, find simple functions and a breakdown of time to continuous periods, such that the first simple function describes well the observations during the first time period, the second function describes the observations in the second time period, and so on. This solution is referred to as a segmentation or piece-wise approximation of the data. Segmentation of data is useful in many ways. For example, in a sensor network, it may be used to save power by sending the piece-wise approximation as a compressed version of the observed information [1]. In finance, it is used to locate transition points (break-points) and study what auxiliary conditions contribute to these transitions [2]. When analyzing ECG data, it is used to measure the durations between pulses [3]. More broadly, segmentation can be used for data compression,

data cleaning, identifying outliers, classification, clustering, timing events and more [4, 5, 6, 7].

The problem of segmentation is not new and has been studied for almost 50 years [8]. When properly formulated, it can be solved in polynomial time using dynamic programming [8, 9]. However, in the era of big data and internet of things, data-sets have become so large that algorithms need to be extremely efficient to cope with the demand. Therefore, even though the dynamic programming solution has a complexity of $O\left(T^2\right)$ where T is the number of observed samples, it cannot handle many tasks as demonstrated in Section 6. For example, if it is applied to a 2 hour long recording of an ECG signal, it becomes about $20\times$ slower than real-time. Therefore, different heuristics exist to approximate the optimal segmentation [10, 11, 3]. The main contribution of this work is a linear time algorithm that finds the exact solution but much faster. Moreover, it does so with a by only adding two *if* clauses to the original solution.

The linear time algorithms allows more than just accelerating the process, it allows handling streaming data too. In the streaming scenario, samples are arriving continuously and the goal is to detect transition points, as close as possible to when they occur [11]. For example, consider the stream of data defining the quote for a stock in a market. For a trader, it may be insufficient to know, at the end of the day, when there were trend transitions in the price of the stock. Instead, the trader would like to know about these events soon after they had happened. In section 4 this scenario is discussed and an algorithm is presented for this case. This algorithm cannot always detect the exact point of the transition. Instead, it is able to mark an interval in which a transition happen. These intervals can be viewed as a special kind of a confidence interval.

Confidence intervals are used in statistics to reflect uncertainty. Different confidence intervals reflect different kinds of uncertainty to guard against. Bai and Perron [9] proved confidence intervals for transition points based on the observation that there is uncertainty about the value of the process in between sample points. However, there is even greater uncertainty about future values of the process. Therefore, the confidence intervals presented are such that regardless of what happen in the future, there must be a transition point in the interval and there is no transition which is not in one of the confidence intervals.

There are many ways to define the segmentation problem. For example, some define it such that the number of segments is given and only the transition points are to be found. This may be justified in a scenario in which a signal has to be compressed to a fixed size which dictates the number of segments. However, in this paper we consider the scenario in which the number of segments has to be optimized. The problem is formally defined in the next section.

2 Basic definitions

In order to define the problem we first define the cost function which is the function that measures the deviation between the observations and the approximation on a single segment. Let $\text{cost}\left(t_1 \to t_2\right]$ be the "penalty" of a single segment starting after time t_1 and ending at time t_2. For example, if approximating the observations using a piecewise constant approximation, the cost function can be defined as $\text{cost}\left(t_1 \to t_2\right] =$

$\inf_{c\in\mathbb{R}} \sum_{t\in(t_1,t_2]} (c-y_t)^2$. Therefore, in this case, the cost is the sum of squared difference between the observations y_t and the best constant $c \in \mathbb{R}$. This can be extended to a more generic setting in which there is a set of functions $\mathbb{F}$ from the time domain to the domain of the observations Y, and a loss function $l : Y \times Y \mapsto \mathbb{R}^+$ such that the cost function is

$$\text{cost}\,(t_1 \to t_2] = \inf_{f\in\mathbb{F}} \sum_{t\in(t_1,t_2]} l\,(f\,(t)\,, y_t) \quad . \tag{1}$$

These cost functions have a common property that play a key role in the analysis to follow:

Definition 1. *A cost function has the inverse triangle inequality property (ITIP) if it is non-negative and for every $t_1 \le t_2 \le t_3$:*

$$\textit{cost}\,(t_1 \to t_2] + \textit{cost}\,(t_2 \to t_3] \le \textit{cost}\,(t_1 \to t_3] \quad .$$

The following lemma shows that this property holds in many cases.

Lemma 1. *Let $\mathbb{F}$ be a set of functions from the time domain to the domain of observations Y and let $l : Y \times Y \mapsto \mathbb{R}^+$be a non negative loss function. The cost function defined in in (1) has the ITIP.*

Proof. Let $t_1 \le t_2 \le t_3$ then

$$\begin{aligned}
\text{cost}\,(t_1 \to t_3] &= \inf_{f\in\mathbb{F}} \sum_{t\in(t_1,t_3]} l\,(f\,(t)\,, y_t) \\
&\ge \inf_{f\in\mathbb{F}} \sum_{t\in(t_1,t_2]} l\,(f\,(t)\,, y_t) + \inf_{f\in\mathbb{F}} \sum_{t\in(t_2,t_3]} l\,(f\,(t)\,, y_t) \\
&= \text{cost}\,(t_1 \to t_2] + \text{cost}\,(t_2 \to t_3] \quad . \qquad \square
\end{aligned}$$

The ITIP is sufficient for the algorithms proposed in the following to be correct. However, in order to gain from these algorithms, a strict version of the ITIP is needed.

Definition 2. *A cost function is (Δ, κ, n) Strictly ITIP (SITIP) if for every t there are $t = t_1 \le t_2 \le \cdots \le t_n = t + \Delta$ such that*

$$\textit{cost}\,(t \to t+\Delta] \ge \kappa + \sum_{i\le n} \textit{cost}\,(t_i \to t_{i+1}] \quad .$$

A segmentation should tradeoff between the number of segments and how well they fit the data [8]. According to this definition, the score of a partition $0 = t_1 < t_2 < \cdots < t_{n+1} = T$ is $nC + \sum_{i=1}^{n} \text{cost}\,(t_i \to t_{i+1}]$ and the optimal partition is the partition that minimizes this value over all possible choices of n and $\{t_i\}_i$. The value of the optimal tradeoff is $\text{opt}\,(T)$. It has already been noted [8] that this can be solved using Dynamic Programming (DP) due to the following observation: if $0 = t_1 \le t_2 \le \cdots \le t_{n+1} = T$ is the optimal partition up-to time T then

$$\text{opt}\,(T) = \text{opt}\,(t_n) + C + \text{cost}\,(t_n \to T] \quad .$$

Therefore, if opt (t) is known for every $t < T$ then it can be computed by opt $(T) = \min_{t<T}$ opt $(t) + C +$ cost $(t \to T]$. Hence, the computation at time T has a computational complexity of $O(T)$[1] and computing the entire opt () function has a complexity of $O\left(T^2\right)$.

This celebrated result has two main limitations. First, many data-sets are too large to allow such complexity. For example, low latency traders [12] look into latencies of the order of a millisecond. Therefore, if one samples stock quotes ~ 1000 times a second, analyzing the data will become slower than real time or completely infeasible in a matter of hours. To address this issue, the focus of this work is around finding ways to compute the same optimal value but doing it much faster. Another limitation of the dynamic programming approach is that it cannot handle streaming data. This issue is addressed in Section 4.

3 Accelerated DP

This section introduces the modifications to the DP algorithm to accelerate it together with the proofs for its correctness and performance. The first lemma states that if the cost function has the ITIP then it is monotone.

Lemma 2. *If the cost function has the ITIP them for every* $t_1 \leq t_2 \leq t_3 \leq t_4$

$$\textit{cost}\,(t_2 \to t_3] \leq \textit{cost}\,(t_1 \to t_4] \quad .$$

Proof. Using the fact that the cost function is non-negative and the inverse triangle inequality, it holds that:

$$\begin{aligned} \text{cost}\,(t_2 \to t_3] &\leq \text{cost}\,(t_1 \to t_2] + \text{cost}\,(t_2 \to t_3] + \text{cost}\,(t_3 \to t_4] \\ &\leq \text{cost}\,(t_1 \to t_3] + \text{cost}\,(t_3 \to t_4] \leq \text{cost}\,(t_1 \to t_4] \quad .\square \end{aligned}$$

The monotone property of the cost function is sufficient to save unnecessary computation. Algorithm DPskip (Algorithm 1 when setting *skip* to true and *prune* to false) demonstrates that. The key idea is that for every $t_1 < t_2$

$$\text{opt}\,(t_1) + \text{cost}\,(t_2 \to T] + C \leq \text{opt}\,(t_1) + \text{cost}\,(t_1 \to T] + C \quad . \tag{2}$$

Therefore, if evidence exists that the left-hand-side of (2) is larger then the value of opt (T) then clearly the right-hand-side is larger too and computation of these values can safely be skipped. DPskip does provide speedup over the baseline but theoretically, its complexity remains $O\left(T^2\right)$. The next step is to show that the opt function is monotone too:

Lemma 3. *If the cost function has the ITIP then for every* $t_1 < t_2$

$$\textit{opt}\,(t_1) \leq \textit{opt}\,(t_2) \quad .$$

[1] Computing the cost function is assumed to take $O(1)$ steps. More details can be found in Section 3.

Proof. It is sufficient to prove the statement when t_2 is the next sample time after t_1 since the rest can be proven by induction. Let t_2 be the next time stamp after t_1. There exists $t \leq t_2$ such that

$$\text{opt}\,(t_2) \quad = \quad \text{opt}\,(t) + C + \text{cost}\,(t \to t_2] \geq \text{opt}\,(t) + C + \text{cost}\,(t \to t_1] \geq \text{opt}\,(t_1) \quad .\square$$

Further analysis of the opt function reveals that if there is $t_1 < t_2$ such that opt (t_2) is much smaller then the value if t_1 was the last segmentation point before t_2, then there must be a segment starting in between these points, and this is independent of the future. In other words, t_1 is a barrier such that when evaluating the value of opt (T), for any $T \geq t_2$, there is no need to consider any past value earlier then t_1. This is stated in the following lemma:

Lemma 4. *If the cost function has the ITIP and $t_1 < t_2$ are such that*

$$\textit{opt}\,(t_2) \leq \textit{opt}\,(t_1) + \textit{cost}\,(t_1 \to t_2] - C$$

then for every $T \geq t_2$:

$$\textit{opt}\,(T) = \min_{t_1 \leq t < T} \textit{opt}\,(t) + \textit{cost}\,(t \to T] + C \quad .$$

Proof. Let $t < t_1 < t_2 \leq T$. From the definition of ITIP (Definition 1) the following is true:

$$\begin{aligned} \text{opt}\,(t) + C + \text{cost}\,(t \to T] \quad &\geq \quad \text{opt}\,(t) + C + \text{cost}\,(t \to t_1] + \text{cost}\,(t_1 \to t_2] + \text{cost}\,(t_2 \to T] \\ &\geq \quad \text{opt}\,(t_1) + \text{cost}\,(t_1 \to t_2] + \text{cost}\,(t_2 \to T] \\ &\geq \quad \text{opt}\,(t_2) + C + \text{cost}\,(t_2 \to T] \quad . \end{aligned}$$

Therefore

$$\begin{aligned} \text{opt}\,(T) \quad &= \quad \min_{t<T} \text{opt}\,(t) + C + \text{cost}\,(t \to T] \\ &= \quad \min\left(\min_{t<t_1} \text{opt}\,(t) + C + \text{cost}\,(t \to T]\,, \min_{t_1 \leq t < T} \text{opt}\,(t) + C + \text{cost}\,(t \to T]\right) \\ &= \quad \min\left(\text{opt}\,(t_2) + C + \text{cost}\,(t_2 \to T]\,, \min_{t_1 \leq t < T} \text{opt}\,(t) + C + \text{cost}\,(t \to T]\right) \\ &= \quad \min_{t_1 \leq t < T} \text{opt}\,(t) + C + \text{cost}\,(t \to T] \quad . \end{aligned}$$

□

Therefore, the barrier allows pruning the computation, not only for the current iteration but also for future iterations. The algorithm which uses this property, DPprune, is presented in Algorithm 1 when *prune* is set to true and *skip* is set to false.

The number of operations for every iteration in the loop For $T = 1, \ldots$ is $O\,(T - \text{barrier})$. Therefore, if barrier $\geq T - \Delta$ for some constant Δ, each iteration requires $O\,(\Delta)$ and the overall complexity of the algorithm is $O\,(T\Delta)$, that is, linear in T. However, this condition need not always hold. For example, assume that cost $(t_1 \to t_2] =$

$\max(0,\ t_2 - t_1 - 1)$ and $C = 2$. It is easy to verify that in this case $\text{opt}(T) = T + 1$. Therefore, if $t_1 < t_2$ then

$$\begin{aligned} \text{opt}(t_1) + \text{cost}(t_1 \to t_2] - C &= t_1 + 1 + t_2 - t_1 - 2 \\ &= t_2 - 1 < \text{opt}(t_2) \quad . \end{aligned}$$

Hence, in this example, the condition of Lemma 4 is never satisfied and no barrier will be found. Therefore, the ITIP is an insufficient condition in this case. However, the following shows that the strict version of the ITIP (SITIP) is a sufficient condition.

Theorem 1. *Assume that the cost function is (Δ, κ, n) SITIP then the complexity of the DPprune algorithm is $O(T\Delta)$.*

Proof. Assume that the cost function is (Δ, κ, n) SITIP and let $t_1, \ldots, t_n$ be such that $T - \Delta = t_1 \leq t_2 \leq \cdots \leq t_n = T$ and

$$\text{cost}(T - \Delta \to T] \geq \kappa + \sum_{i<n} \text{cost}(t_i \to t_{i+1}] \quad .$$

We have that

$$\text{opt}(T) - \text{opt}(T - \Delta) \quad \leq \quad nC + \sum_{i<n} \text{cost}(t_i \to t_{i+1}] \leq nC - \kappa + \text{cost}(T - \Delta \to T] \quad .$$

Therefore, if $C \leq \frac{\kappa}{(n+1)}$ then$nC - \kappa \leq -C$ and therefore

$$\text{opt}(T) - \text{opt}(T - \Delta) + C \leq \text{cost}(T - \Delta \to T] \quad .$$

This forces the barrier in DPprune to be at least $T - \Delta$. Therefore, each iteration is $O(\Delta)$ operations which will result in $O(T\Delta)$ complexity for the entire algorithm. Note that the last part of the algorithm, where the ends of each segments are outputted has $O(T)$complexity since prev$[t] < t$ for every $t > 0$. □

Note that in theory combining the pruning technique and skipping technique might lead to negative consequences since the skipping criterion might skip computing the cost at the barrier point which could lead to excess computation. However, in practice, as demonstrated in the empirical section, the skipping techniques provides additional $1.5 - 2x$ speedup on top of the pruning technique.

Another point to consider is the true complexity of computing the cost function. So far this computation was considered to have $O(1)$ complexity. However, this is not always the case. For example, finding the linear approximation that minimizes the L_1 loss requires solving a linear programming problem. However, when the L_2 loss is used, the computation can be done in $O(1)$ complexity. This can be done since in this case there are statistics which suffice to compute the approximation and its loss of the form $S(t_1, t_2] = \sum_{t_1 < t \leq t_2} z_t$ where z_t is some function of t and the observed value at time t. Bai and Perron [9] suggested a preprocessing step in which all the terms $S(0, t]$ are computed in $O(T)$ steps. Once all these values are available, it is possible to compute $S(t_1, t_2]$ since $S(t_1, t_2] = S(0, t_2] - S(0, t_1]$. The approach of Bai and Perron [9] creates numerical stability problems when T is large since $S(0, t]$. will have

limited accuracy and hence if $0 \ll t_1$ but t_1 is close to t_2 the value of $S(0, t_2] - S(0, t_1]$ may deviate significantly from $\sum_{t_1 < t \leq t_2} z_t$. Therefore, our implementation uses a different approach.

The DP algorithm calls the cost function in the following order: for every T, it first computes the costs is a sequence $\text{cost}(T-1 \to T], \text{cost}(T-2 \to T], \ldots$. To compute $\text{cost}(T-1 \to T]$ terms of the sort of $S(T-1, T] = z_T$ are needed. Since $S(t, T] = S(t+1, T] + z_{t+1}$, by caching $S(t+1, T]$, it is possible to compute $S(t, T]$ in a single operation. This approach uses $O(1)$ time, $O(1)$ memory and is more stable. When skipping is used, the complexity is no longer $O(1)$ it requires summing over all elements that were skipped. Nevertheless, it is used in the experiments due to its better stability. An implementation of this method is provided in Section 5.

4 Streaming data and confidence intervals

In many cases the goal is to find breakpoints and segments while the data is streaming in. For example, consider a system that monitors traffic: such a system is expected to report on changes in traffic patterns as close as possible to when they happen such that adjustments can be made to traffic lights for example. The original DP solution is not capable of doing so since every new entry has the potential of changing the entire segmentation of the data and thus it is impossible to commit to any part of it. Moreover, the amount of computation per entry grows over time and hence it is impossible the get real-time performance without using heuristics. However, Lemma 4 shows that if the cost function is SITIP, then barriers will emerge in the data. This will guarantee that the computation has fixed complexity and that guarantees on breakpoints that happen before the barrier can be made. This section expands the discussion about these subjects.

Recall, the Lemma 4 proves that if the cost function has the ITIP and $t_1 < t_2$ are such that

$$\text{opt}(t_2) \leq \text{opt}(t_1) + \text{cost}(t_1 \to t_2] - C \tag{3}$$

then for every $T \geq t_2$:

$$\text{opt}(T) = \min_{t_1 \leq t < T} \text{opt}(t) + \text{cost}(t \to T] + C \ .$$

Assume that when computing $\text{opt}(t_2)$ a $t_1 < t_2$ is found such that (3) holds. Therefore, regardless of any future information, all the segments that begin before t_1 must end before t_2. Using the notation of Algorithm 1 it holds that $\forall T > t_2, \ \text{prev}[T] \geq t_1$. This leads to the following conclusions that are summarized in the following lemma:

Lemma 5. *If the cost function has ITIP and (3) holds for $t_1 < t_2$ then,*

1. *If $t < t_1$ is such that $\forall t' \leq t_2, \ \text{prev}[t'] \neq t$ then regardless of any future information, there is no break-point at time t.*
2. *If $t < t_1$ is such that $\forall t < t' \leq t_2, \ \text{prev}[t'] \geq t$ then regardless of any future information, there is a break-point at time t and the optimal segmentation up-to time t can be determined.*

Table 1: **Description of the data used in the experiments**

Name	Source	Description	# Samples
DJA	[13]	Daily Closing Value of the Dow Jones Average 1960-2013. The data was converted to log-scale.	13583
ECG	[14, 15]	This is file e0103 from the data-set which contains a 2 hour recording of an ECG signal recorded at 250Hz	1800000
Dodgers	[16]	Loop sensor data collected for the Glendale on ramp for the 101 North freeway in Los Angeles	50400

3. *If $t < t_1$ is such that $\forall t < t' \leq t_2$, $\mathrm{prev}\,[t'] \geq t - \tau$ then regardless of any future information, there is a break-point between time $t - \tau$ and time t.*

According to this lemma, it is possible to determine properties of the optimal segmentation before the entire sequence has been revealed.

Proof. 1) Assume that there is a break-point at time t. Therefore, there is $t' > t$ such that $\mathrm{prev}\,[t'] = t$. However, from the assumption of the lemma, this requires that $t' > t_2$ but for $t' > t_2$ Lemma 4 shows that $\mathrm{prev}\,[t'] \geq t_1 > t$ and therefore, there could not be a break-point at time t.

2) Assume that there is no break-point at time t. Therefore, there is $t' > t$ such that $\mathrm{prev}\,[t'] < t$. If such t' exists, it has to be greater than t_2 and Lemma 4 shows that $t' > t_2$ could not have this property either. Therefore, a break-point must occur at t. The property of the optimal segmentations that was used to show that it can be found using dynamic programming is that if there is a break-point at time t then the optimal segmentation must contain, as a subset, the optimal segmentation up-to time t.

3) Repeating the same argument as above, for every $t' > t$, it is true that $\mathrm{prev}\,[t'] \geq t - \tau$. Let $T > t_2$ and consider the optimal segmentation to time T. Assuming that $t - \tau > 0$ then there exists n such that $\mathrm{prev}^n\,[T] \geq t - \tau$ but $\mathrm{prev}^{n+1}\,[T] < t - \tau$. Therefore, it must be that $t \geq \mathrm{prev}^n\,[T] \geq t - \tau$.

□

Lemma 5 suggests that by modifying the DPprune algorithm, it is possible to determine for each point before the current barrier whether there is a potential of having a break-point at this point and how far is it from the last break-point. If the SITIP exists, and C is not too large, the barrier is within Δ time steps from the current time and therefore, it is possible to determine these properties for every point that is at least Δ time steps from the current time. This calculation can be made with $O\,(1)$ operations for every data point by holding a simple data structure. The details are omitted due to space limitations. The analysis here may explain the success of the SWAB algorithm [11]. However, the algorithm presented here has significant advantages: its performance is provable and it does not require any tuning (beside the parameter C).

These results can be interpreted also in terms of confidence intervals. Bai and Perron [9] developed confidence intervals for segmentation. The assumption is that the data is generated from a piecewise linear model and the question is how would

the location of the breakpoints change if the data is generated with finer resolution. The assumption is that the duration of the process being inspected is fixed but there is uncertainty regarding the values of the process in-between sample points. While this is true, there is even greater uncertainty about the future. Lemma 5 shows how to build such confidence intervals and proves their correctness. In Section 5 an example of possible way of using these confidence intervals in streaming mode is demonstrated.

5 Implementation

In this section we present a complete implemntation of the algorithms presented in this work. We use F# for this code since it allows for a concise code that is high performing and available on many platforms. The code presented here implements the following:

- Computing the segment costs for piecewise linear segmentation with the squared loss
- Computing the segments (Algorithm 1)
- Computing the confidence intervals

```
//
    ================================================================

/// A circular array that allows us to add new
    elements by reusing
/// unused past elements
type Buffer<'a>(capacity) =
    let data = Array.zeroCreate<'a> capacity
    member m.Reset i = m.[i] <- Unchecked.defaultof<'a
        >
    member m.Item with get i       = data.[i%capacity]
                  and  set i value = data.[i%capacity]
                       <- value
    member m.Range(a, b)  = [ for i = a to b do yield
        m.[i] ]
    member m.Remove(a, b) = [a..b] |> Seq.iter(m.Reset
        )

//
    ================================================================

// Optimized mean error calculation that remembers the
     state of
// previous calls. This is very efficient when the
    requested
```

```
// interval is one greater than the previously
   requested interval
type IncrementalMeanError() =
    let mutable sum, sumSquared = 0.0,0.0
    let mutable lastI, lastJ    = 0,0
    member x.Cost getData i j =
        while not(i = lastI && j = lastJ) do
            if (i < lastI && j = lastJ) then
                lastI <- lastI - 1
                let v = getData lastI
                sum        <- sum + v
                sumSquared <- sumSquared + v*v
            else
                sum <- 0.0; sumSquared <- 0.0
                lastI <- j; lastJ <- j
        let range = j - i;
        max 0.0 (sumSquared - sum * sum / float(range)
           )
```

```
/// Calculate the optimal segmentation for a data
    sequence
/// C    - the segmentation penalty
/// cost - a function that gives the loss between two
    datapoints
/// dataSize - the size of the data if fixed, or the
    buffer size
///            for streamed data
type Segmenter(C, cost, dataSize) =
    let opt       = Buffer<float>(dataSize)
    let prev      = Buffer<int>(dataSize)

    // A chain of indexes to previous optimal segment
       breaks
    member m.Prev = prev

    // Update the optimal breakpoints for latest data
       point
    member m.Update barrier T =
        opt.[T] <- if T = barrier then 0.0 else Double
           .MaxValue
        let rec sweep t = function
        | lastCost when t < barrier -> barrier
        | lastCost when skip(t, lastCost) ->
            sweep(t-1) lastCost
        | lastCost ->
            let computedCost = cost t T
            if (prune(t, computedCost)) then t
            else
                let candidate = opt.[t] + computedCost
                    + C
                if candidate < opt.[T] then
                    opt.[T] <- candidate
                    prev.[T] <- t
                sweep (t-1) computedCost
        and skip(t,lastCost) = lastCost + opt.[t] + C
           > opt.[T]
        and prune(t,lastCost) = lastCost >= opt.[T] -
           opt.[t] + C
        sweep (T-1) 0.0

    /// Find the optimal break points for a fixed
       dataset
    static member OptimalBreaks C cost data =
        let dataSize = Array.length data
        let s = Segmenter(C, cost, dataSize)
```

```
// Update the optimal breakpoints for all data
    elements
[0..dataSize-1] |> Seq.fold (s.Update) 0 |>
   ignore

// Walk the prev buffer backwards to obtain
   the optimal
// segmentation
let rec endpoints index = seq {
    if index > 0 then
        yield  index
        yield! endpoints(s.Prev.[index]) }
endpoints (dataSize-1)
```

```
/// Transform a stream of datapoints to a new stream
   that
/// indicates at which index there could possibly be a
    segment
/// break and the maximum distance to the begining of
   the
/// segment each point belongs to.
/// C    - the segmentation penalty
/// cost - a function that gives the loss between two
   datapoints
/// bufferSize - the buffer size for the streamed data
type StreamSegmenter(C, cost, bufferSize) =
    let data                  = Buffer<float>(
       bufferSize)
    let possibleBreak         = Buffer<bool>(
       bufferSize)
    let maxDistanceToBeginning = Buffer<int>(
       bufferSize)
    let getData i = data.[i]
    let segmenter = Segmenter(C, cost getData,
       bufferSize)
    let prev = segmenter.Prev

    // Update the maximum distance to a breakpoints
    let updateConfidenceInterval t =
        let p = prev.[t]
        for i = t downto p do
            maxDistanceToBeginning.[i] <-
                max maxDistanceToBeginning.[i] (i - p)
        possibleBreak.[p] <- true

    // This function takes the previous state and a
       new
    // datapoint, updates the internal buffers and
       returns
    // a new state to start from for the next
       datapoint
    // arrives
    let segmentStream state datapoint =
        let ((T,barrier), _) = state
        data.[T] <- datapoint
        let newBarrier = segmenter.Update barrier T
        updateConfidenceInterval T

        // When we reach a new barrier we can emit
           endpoints
```

```
    // behind it since they can no longer be
       affected
    if (newBarrier <> barrier) then
        // It is only safe to emit data points
           between the
        // last barrier and the new barrier.
        let range = barrier, (newBarrier-1)
        let breakpoints = maxDistanceToBeginning.
           Range range
                        |> Seq.zip(possibleBreak.
                           Range range)
        maxDistanceToBeginning.Remove range
        possibleBreak.Remove range
        ((T+1, newBarrier), breakpoints)
    else
        ((T+1, barrier), Seq.empty)

// Transform an observable sequence of data points
    to an
// delayed observable sequence of (possible,
   distance) pairs.
member private m.SegmentStream =
    let gather (obs:IObservable<_*_ seq>) = obs.
       SelectMany snd
    let T,barrier = 0,0
    let initialState = ((T,barrier),Seq.empty)
    Observable.scan segmentStream initialState
    >> gather

/// Collect possible breakpoints and distances to
   those points
static member PossibleBreaks(C, cost, bufferSize)
   =
    StreamSegmenter(C, cost, bufferSize).
       SegmentStream
```

```
//
    ----------------------------------------------------------------

/// Usage examples:
//
    ----------------------------------------------------------------

let boxWave next =
    Seq.init 100 (fun i -> i + next*100)
    |> Seq.map(fun x -> if (x/100)%2 = 0 then 3.0 else
        -3.0)

//
    ----------------------------------------------------------------

let finding_exact_breakpoints_in_fixed_data =
    let data = [0..9] |> Seq.collect boxWave |> Seq.
       toArray
    let meanError = IncrementalMeanError().Cost(fun i
       -> data.[i])
    Segmenter.OptimalBreaks 0.01 meanError data
    |> Seq.iter(printfn "Line segment starts at %d")

//
    ----------------------------------------------------------------

// Usage examples:
//
    ----------------------------------------------------------------

let numberOfElements = 1000
let addIndex i x = (i,x)
let meanError = IncrementalMeanError().Cost

//
    ----------------------------------------------------------------

let finding_exact_breakpoints_in_a_stream =
    let data = [0..9] |> Seq.collect boxWave
    let isDistanceZero(index, (possible, dist)) = dist
        = 0
    let exactBreakpoints =
        data.ToObservable()
        |> StreamSegmenter.PossibleBreaks(0.5,
           meanError, 1000)
        |> Observable.mapi addIndex
```

```
        |> Observable.filter isDistanceZero
        |> Observable.map fst
    exactBreakpoints.ForEachAsync(
        fun x -> printfn "Line segment starts at %d" x
           )

//
   ---------------------------------------------------------------

let
   finding_possible_breakpoints_in_a_stream_with_confidence
    =
    let sine i = sin(0.02*Math.PI*float(i))
    let smoothData = Array.init numberOfElements sine
    let fuzzyBreakpoints =
        smoothData.ToObservable()
        |> StreamSegmenter.PossibleBreaks (0.5,
           meanError, 1000)
        |> Observable.mapi addIndex

    fuzzyBreakpoints.ForEachAsync(fun (i, (possible,
       distance)) ->
        let may = if (possible) then "may" else "doesn
           't"
        printfn "A segment %s start at index %A,\n\
                 the begining of the current segment\n
                    \
```

Table 2: **Time to segment entire data-set.** Duration of execution are measured in seconds.

Data-set	Cost	# segments	DP	DPskip	DPprune	DPcombine
DJA	Mean	20	8 ± 0.5	1.8 ± 0.1	1.5 ± 0.2	0.7 ± 0.1
DJA	L2	13	$12 + 0.6$	3.4 ± 2.1	3 ± 1.6	1.5 ± 0.7
ECG	Mean	17450	138195.8	27028.8	25.1	18.5
ECG	L2	16588	200525	47471.7	36.5	24.2
Dodgers	Mean	9194	114 ± 3.5	18 ± 1	0.2 ± 0.02	0.13 ± 0.025
Dodgers	L2	8645	170 ± 3.3	26 ± 1.8	0.3 ± 0.2	0.2 ± 0.05

6 Empirical evaluation

Keogh and Kasetty [17] argued that when evaluating algorithms for analyzing time series it is important to use data-sets from different domains since they may represent different characteristics. Therefore, the data-sets used are from 3 different domains: medical, financial and engineering data. The data is described in Table 1. The different algorithms were applied to each data-set with two cost functions. The *Mean* cost function finds the best constant approximation for a segment and evaluates the sum of squared distances between the approximation and the measurement. The L_2 cost functions finds the best linear approximation for a segment and evaluates the sum of squared distances between the approximation and the measurement. Note that both cost function have the ITIP. Reference implementation is provided in Section 5.

For each file, the algorithms have been applied to prefixes of different lengths. For the DJA and Dodgers data-set the experiment was repeated 10 times.[2] The value for C was chosen to be half of the range between the minimal value and the maximal value of the DJA and Dodgers data-sets and 0.1 for the ECG data-set after the data was converted to the range $[0, 1]$.[3]

Table 2 presents the time it takes the different algorithms to process the different data-sets. When applied to these data-sets the order of the algorithms from fastest to slowest is always DPcombine than DPprune, followed by DPskip and DP. All these differences are statistically significant with $p < 0.01$. Overall, the DP algorithm is $8 - 8200$ times slower than DPcombine.

DPcombine and DPprune take a longer time to process the DJA data-set than the Dodgers data-set, despite the fact that the later data-set is 4 times larger. This is explained by the length of the segments in each data-set. In DJA, the average length of a segment is $680 - 1045$ samples while in the Dodgers data-set, the average length of a segment is about 6 samples only. Therefore, the pruning has a much larger impact in this case.

Figure 1 presents the time to segment the different data-sets given different sizes of prefixes of the data. In all data-sets, the contribution of the new method becomes

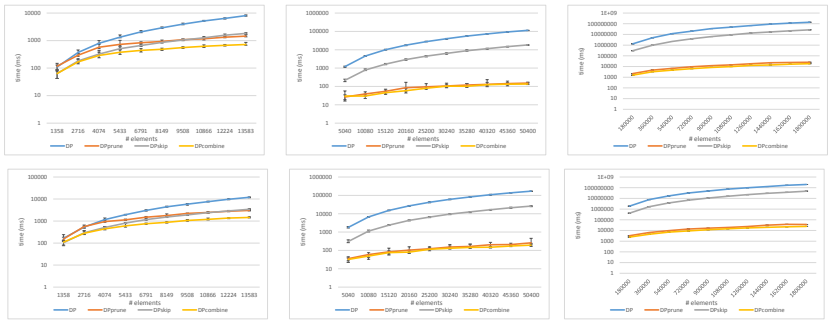
[2]The non-accelerated versions of the DP algorithm take about a week to complete and therefore repeating this experiment was prohibitive.

[3]The values of C were chosen arbitrarily. For the ECG data-set we eye-balled a small prefix of the data to set C such that it will capture the R wave and sometimes the T wave as well.

```
barrier ← 0;
opt[0] ← 0;
for T ← 1 to ... do
    lastComputedCost ← 0 ;
    opt[T] ← ∞ ;
    for t ← T − 1 to barrier do
        if skip & lastComputedCost + opt[t] + C > opt[T] then continue;
        lastComputedCost ← cost (t → T];
        if prune & lastComputedCost ≥ opt[T] − opt[t] + C then
            barrier ← t;
            break;
        end
        candidate ← opt[t] + lastComputedCost + C;
        if candidate < opt[T] then
            opt[T] ← candidate;
            prev[T] ← t;
        end
    end
end
/* Output the ends of all segments                    */
t ← T;
while t > 0 do
    output "a segment ends at time " t;
    t ← prev[t];
end
```

Algorithm 1: Time series segmentation algorithm. When the *skip* parameter is true, it will use the monotonicity described in Lemma 2 to skip cost computations. When the *prune* parameter is set it will use the barriers described in Lemma 4 to early prune the computation. The baseline DP algorithm has both *skip* and *prune* set to false. A reference implemetation of the DPprune algorithm is provided in the supporting material.

Figure 1: **Time to segment different data-sets.** The top row presents the time to segment using the *mean* cost function while the bottom row uses the *L2* loss function. The different datatsets DJA, Dodgers are presented left to right. The X axis represents the size of the problem and the Y axis is the time to complete measured in milliseconds in logarithmic scale. The blue, orange, gray and yellow lines represent DP, DPprune, DPskip and DPcombine respectively. It is clear from these graphs that the DPcombine algorithm is orders of magnitude faster than the DP algorithm.



larger as the data-set size increases. When comparing the contribution of skipping calculations to the contribution of pruning, it seems as if on smaller data-sets skipping sometimes has a larger impact. However, as the data-set grows in size, pruning dominates skipping significantly.

7 Conclusions

The theoretical study resulted in an accelerated segmentation algorithm which has linear complexity and, as expected, creates big speed-ups when applied to data ($8 - 8200\times$). Moreover, the accelerated algorithms opened the door for segmentation in the streaming scenario. However, several challenges remain for future research such as, creating a parallel version of the algorithm an even greater challenge is to apply the technique presented here or techniques similar to it, to accelerate other dynamic programming tasks. It is also interesting to see if the method can be extended to cases in which the ITIP may not hold [18].

References

[1] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. Compressing historical information in sensor networks. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 527–538. ACM, 2004.

[2] Siew Ann Cheong, Robert Paulo Fornia, Gladys Hui Ting Lee, Jun Liang Kok, Woei Shyr Yim, Danny Yuan Xu, and Yiting Zhang. The Japanese economy in crises: A time series segmentation study. *Economics*, 5, 2012.

[3] F. Sufi, Q. Fang, and I. Cosic. ECG RR Peak Detection on Mobile Phones. In *IEEE EMBS*, 2007.

[4] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment*, 1(2):1542–1552, 2008.

[5] Björn Geißler, Alexander Martin, Antonio Morsi, and Lars Schewe. Using piecewise linear functions for solving minlps. In *Mixed Integer Nonlinear Programming*, pages 287–314. Springer, 2012.

[6] Eamonn J Keogh and Michael J Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *KDD*, volume 98, pages 239–243, 1998.

[7] Mano Ram Maurya, Raghunathan Rengaswamy, and Venkat Venkatasubramanian. Fault diagnosis using dynamic trend analysis: A review and recent developments. *Engineering Applications of Artificial Intelligence*, 20(2):133–146, 2007.

[8] Richard Bellman and Robert Roth. Curve fitting by segmented straight lines. *Journal of the American Statistical Association*, 64(327):1079–1084, 1969.

[9] Jushan Bai and Pierre Perron. Computation and analysis of multiple structural change models. *Journal of applied econometrics*, 18(1):1–22, 2003.

[10] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. Segmenting time series: A survey and novel approach. *Data mining in time series databases*, 57:1–22, 2004.

[11] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An online algorithm for segmenting time series. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 289–296. IEEE, 2001.

[12] Joel Hasbrouck and Gideon Saar. Low-latency trading. *Journal of Financial Markets*, 16(4):646–679, 2013.

[13] Samuel H. Williamson. Daily closing value of the Dow Jones average, 1885 to present. MeasuringWorth, 2014.

[14] A Taddei, G Distante, M Emdin, P Pisani, GB Moody, C Zeelenberg, and C Marchesi. The European ST-T database: standard for evaluating systems for the analysis of ST-T changes in ambulatory electrocardiography. *European heart journal*, 13(9):1164–1172, 1992.

[15] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. Physiobank, physiotoolkit, and physionet components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

[16] Dodgers loop sensor data set. Freeway Performance Measurement System (PeMS), `http://pems.dot.ca.gov/`.

[17] Eamonn Keogh and Shruti Kasetty. On the need for time series data mining benchmarks: a survey and empirical demonstration. *Data Mining and knowledge discovery*, 7(4):349–371, 2003.

[18] Daniel Lemire. A better alternative to piecewise linear time series segmentation. In *SDM*. SIAM, 2007.