Journal of Statistical Software
MMMMMM YYYY, Volume VV, Issue II. doi: 10.18637/jss.v000.i00

Comparing binsegRcpp with other implementations of binary segmentation for change-point detection

Toby Dylan Hocking
Université de Sherbrooke

Abstract

Binary segmentation is a classic algorithm for detecting change-points in sequential data. In theory, using a simple loss function like Normal or Poisson, binary segmentation should be extremely fast for N data and S segments: asymptotically $O(NS)$ time in the worst case, and $O(N \log S)$ time in the best case. In practice, existing implementations can be asymptotically slower, and can return incorrect results. We propose **binsegRcpp**, an R package which provides a correct C++ implementation, with the expected asymptotic time complexity. We discuss several important C++ coding techniques, and include detailed comparisons with other implementations of binary segmentation: **ruptures**, **fpop**, **changepoint**, **blockcpd**, and **wbs**.

Keywords: C++, R, binary segmentation, change-point.

1. Introduction: previous software for change-point detection

Change-point detection is an important problem in sequential data analysis, that occurs in a variety of different applications, such as genomics (Hocking, Schleiermacher, Janoueix-Lerosey, Boeva, Cappo, Delattre, Bach, and Vert 2013), neuroscience (Jewell, Hocking, Fearnhead, and Witten 2019), and medical monitoring (Fotoohinasab, Hocking, and Afghah 2020), among others. The focus of this article is change-point detection in the off-line setting, in which we assume there is a sequence of N observed data, and we want an algorithm to identify the precise locations of any abrupt changes that may be present (Truong, Oudre, and Vayatis 2020). We assume there is user-specified maximum number of segments S, and we would like to compute a sequence of models from 1 to S segments (0 to $S-1$ change-points).

Using the classical dynamic programming algorithm of Auger and Lawrence (1989), optimal change-points for a given cost function, and a sequence of model sizes $k \in \{1, \dots, S\}$, can be computed in $O(N^2 S)$ time. More recently, the pruned dynamic programming algorithms

package	binsegRcpp	changepoint	wbs	fpop	ruptures	blockcpd
function	binseg	cpt.mean	sbs	multiBinSeg	Binseg	fit_blockcpd
version	2022.3.29	2.2.3	1.4	2019.8.26	1.1.9	1.0.0
weights	yes	no	no	no	no	no
min len	yes	yes	no	no	yes	yes
max segs	yes	yes	no	yes	yes	yes
dim	one	one	one	multi	multi	multi
correct	yes	no	yes	yes	yes	yes
losses	5	6	1	1	10	5
language	C++	C	C	C++	Python	C++
storage	heap	arrays	recursion	heap	LRU cache	heap
space	$O(S)$	$O(S^2)$	$O(S)$	$O(S)$	$O(S)$	$O(S)$
cumsum	yes	yes	yes	yes	no	yes
best	$O(N \log N)$	$O(N^3)$	$O(N \log N)$	$O(N \log N)$	$O(N^{1.4})$	$O(N \log N)$
worst	$O(N^2)$	$O(N^3)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
CV	yes	no	no	no	no	no
params	all	one	no	no	no	one

Table 1: Properties (rows) of six binary segmentation libraries (columns). Properties are weights (can specify observation-specific positive weights for the loss), min len (can specify minimum segment length parameter), max segs (can specify maximum number of segments parameter), dim (dimensions of input data), correct (does code return correct model parameters?), losses (number of loss functions implemented), language (programming language to implement loss), storage (technique used to store cost of splitting a segment), space (space complexity in number of segments S), cumsum (cumulative sum used), best and worst case time complexity (using N data and $S = N$ segments), CV (support for cross-validation), params (parameters returned, one or all model sizes).

of Maidstone, Hocking, Rigaill, and Fearnhead (2017) can be used to compute the same sequence of S optimal models, with the same $O(N^2 S)$ worst case time complexity, but faster $O(NS)$ time best case time complexity. Other dynamic programming algorithms such as PELT (Killick, Fearnhead, and Eckley 2012) and DuST (Truong and Runge 2024) compute a single model in best case linear time $O(N)$, worst case quadratic time $O(N^2)$. They may be used as a sub-routine to efficiently compute a range of S change-point models, using the CROCS algorithm (Liehrmann, Rigaill, and Hocking 2021), which again results in best case $O(NS)$ time, worst case $O(N^2 S)$ time. Li and Zhang (2024) propose a change-point algorithm based on dynamic programming, and provide comparisons with several previous algorithms.

Binary segmentation is a heuristic algorithm for computing a sequence of models from 1 to S change-points (Scott and Knott 1974). Using a simple loss function like the square loss, the best case of binary segmentation is $O(N \log S)$ time (Hocking 2024), which is much faster than dynamic programming. The worst case of binary segmentation is $O(NS)$ time, which is the same speed as the best case of dynamic programming.

1.1. Existing implementations of binary segmentation

There are several free/open-source implementations of binary segmentation, most of which are R packages with C/C++ code (Table 1). We compared our proposed implementation to these baselines in terms of functionality, correctness, and efficiency. None of these existing implementations support cross-validation, nor observation-specific weights, which are unique features that we propose in **binsegRcpp**.

The **changepoint** package in R contains the `cpt.mean` function which implements several different change-point detection algorithms, including binary segmentation via `method="BinSeg"` (Killick and Eckley 2014; Killick, Haynes, and Eckley 2022). Six loss functions are supported, along with min segment length parameter, max segments parameter, and returning the segment-specific parameters for one model (not all). The implementation does not always return correct results, and requires more time/space than expected. For S segments, an $S \times S$ matrix is used to represent the segment end positions, whereas only a vector of S values is required. Therefore, the time complexity is $O(NS^2)$, much slower than the expected $O(N \log S)$. Likewise, the space complexity is $O(S^2)$ rather than the expected $O(S)$.

The **wbs** package in R contains the `sbs` function, which implements the square loss for univariate data (Baranowski and Fryzlewicz 2019). Since it uses recursive function calls, it does not support the max segments parameter S, and instead always computes the full path of models (from 1 to $N-1$ change-points). It does not implement min segment length, nor returning the segment mean parameters.

The **fpop** package in R contains the `multiBinSeg` function, which implements the square loss for multi-dimensional data (Maidstone *et al.* 2017). It implements the max segments parameter, but not min segment length, nor returning the segment mean parameters.

The **ruptures** package in python contains the `Binseg` class, which implements binary segmentation for 10 loss functions (Truong *et al.* 2020). It supports min length parameter and max segments parameter. The cumulative sum is not used in the implementation of the square loss, which is normally required to achieve optimal best case time complexity. When running on N best case data, with $S = N$ segments, we have observed approximately $O(N^{1.4})$ time complexity (see Results section), which is slower than the expected/optimal $O(N \log N)$ time complexity.

The **blockcpd** package in R contains the `fit_blockcpd` function, which implements several algorithms for change-point detection, including binary segmentation via `method="hierseg"` (Prates and Leonardi 2021; Prates, Lemes, Hünemeier, and Leonardi 2021). It supports input of min segment length and maximum segments, and output of segment mean parameters for one model size (not all).

2. Statistical model, distributions and loss functions

We assume there is a sequence of N data $x_1, \dots, x_N \in \mathbb{R}$, in which we want to find abrupt changes. For each index $i \in \{1, \dots, N\}$, we assume $x_i \sim \mathcal{D}(\theta_i)$, meaning the observed data value x_i is drawn from some distribution $\mathcal{D}$ with un-observed parameter θ_i, typically the mean (and perhaps variance/scale). We would like to compute a sequence of parameter values, $\theta_1, \dots, \theta_N$, which is piecewise constant. We would like to find a sequence of change-points, such that each segment has a parameter θ which minimizes a loss function $\ell(\theta, x)$ that depends on the distribution $\mathcal{D}$. The loss functions that we consider to minimize can be interpreted as maximizing the log likelihood of several common distributions (Normal, Poisson, Laplace).

Furthermore, we assume that each data point $i \in \{1, \dots, N\}$ has a corresponding weight $w_i > 0$, which is used in the loss function via $w_i \ell(\theta, x_i)$. For example, minimizing the square loss $w_i\ell(\theta, x_i) = w_i(\theta - x_i)^2$ corresponds to maximizing the normal log-likelihood, when the variance is uniform.

Choice of observation-specific weights. Weights can be used to efficiently handle two particular kinds of data. First, in the case of data which are sampled uniformly across space/time, typically the weights are constant ($w_i = 1$ for all i), but other weights can be used for data which are sampled non-uniformly (for example daily time series with some missing days). Second, in the case of data with runs of identical values, a run-length encoding can be used to convert the raw/unweighted data sequence into a compressed/weighted data sequence that can be more efficiently processed. For example, in genomics, DNA sequence reads are aligned to a reference genome, and a coverage profile is computed by counting the number of aligned reads at each reference position. Since each DNA sequence read is about 100 bases long, the coverage profile has runs of identical values, which can be converted to a run-length encoding for $\approx 100\times$ speedups.

We consider five change-point models with loss functions ℓ, each of which have a corresponding value for the `distribution.str` argument of the proposed function, `binsegRcpp::binseg` (Table 2). The five models can be divided in two categories: absolute error and cumulative sum.

2.1. Loss functions based on absolute error

Two models have loss functions that involve the absolute error function, and the Laplace distribution. The probability density function of the Laplace distribution, for data $x \in \mathbb{R}$, with location parameter $\mu \in \mathbb{R}$, and scale parameter $b > 0$, is

$$\frac{1}{2b}\exp\left(\frac{|\mu - x|}{-b}\right). \tag{1}$$

For `distribution.str="l1"`, the segment-specific parameter $\theta = \mu \in \mathbb{R}$ is the median. In this case, maximizing the probability corresponds to minimizing the negative log likelihood, which simplifies to the absolute error loss function,

$$\ell_{\texttt{l1}}(\mu, x) = |\mu - x|. \tag{2}$$

For `distribution.str="laplace"`, the segment-specific parameter $\theta = [\mu, b] \in \mathbb{R}^2$ is a vector with two values, the median μ and scale b. In this case, the loss function we use is the negative log likelihood,

$$\ell_{\texttt{laplace}}([\mu, b], x) = \log(2b) + |\mu - x|/b. \tag{3}$$

For a segment from j to e ($1 \leq j \leq e \leq N$), the estimated parameters are

$$\mu^*_{j,e} = \arg\min_{\mu} \sum_{i=j}^{e} w_i|\mu - x_i| \tag{4}$$

$$b^*_{j,e} = \frac{\sum_{i=j}^{e} w_i|\mu^* - x_i|}{\sum_{i=j}^{e} w_i}. \tag{5}$$

Note that there could be a range of $\mu^*_{j,e}$ values which minimize the absolute error (for example with an even number of data and uniform weights). In that case, we define $\mu^*_{j,e}$ as the value

`distribution.str`	Loss	Distrib.	Center	Scale	Params	Complexity
`mean_norm`	Square	Normal	yes	no	1	$O(N)$
`meanvar_norm`	Neg. Log Lik.	Normal	yes	yes	2	$O(N)$
`poisson`	Neg. Log Lik.	Poisson	yes	yes	1	$O(N)$
`l1`	Absolute	Laplace	yes	no	1	$O(N \log N)$
`laplace`	Neg. Log Lik	Laplace	yes	yes	2	$O(N \log N)$

Table 2: Different change-point models provided by `binsegRcpp::binseg`. All models support a change in center parameter; two models have an additional scale parameter; change in Poisson rate parameter affects both mean and variance. Complexity is time for computing best split on a segment with N data.

in the middle of the range of values which minimize the absolute error (this is the traditional definition of the sample median).

In the proposed C++ code, a `class absDistribution` implements the common operations for the `l1` and `laplace` loss functions, including parameter estimation via a cumulative median algorithm, which is required to efficiently compute the optimal split point on a segment. In particular, when we search for the best split on a segment with $s = e - j + 1$ data, we need to compute the sequence of parameter estimates $\mu^*_{j,j}, \dots, \mu^*_{j,e-1}$ before the split, and $\mu^*_{j+1,e}, \dots, \mu^*_{e,e}$ after the split. Computing these parameter estimates and corresponding loss values can be done in $O(s \log s)$ time, using a modified version of the L1 loss minimization code proposed by Drouin, Hocking, and Laviolette (2017).

2.2. Loss functions based on cumulative sums

There are three models which use the cumulative sum trick to find the best split on a segment with s data in $O(s)$ time. In particular, for a sequence of N data, we compute the vectors $W, X, Y \in \mathbb{R}^{N+1}$ in linear $O(N)$ time via:

$$\text{Cumulative weights: } W_0 = 0, \qquad \forall t \in \{1, \dots, N\},\, W_t = \sum_{i=1}^t w_i = W_{t-1} + w_i, \tag{6}$$

$$\text{Cumulative sum: } X_0 = 0, \qquad \forall t \in \{1, \dots, N\},\, X_t = \sum_{i=1}^t w_i x_i = X_{t-1} + w_i x_i, \tag{7}$$

$$\text{Cumulative squares: } Y_0 = 0, \qquad \forall t \in \{1, \dots, N\},\, Y_t = \sum_{i=1}^t w_i x_i^2 = Y_{t-1} + w_i x_i^2. \tag{8}$$

For `distribution.str="mean_norm"`, the normal change in mean model, the segment-specific parameter $\theta \in \mathbb{R}$ is the mean. For data $x \in \mathbb{R}$, mean $\mu \in \mathbb{R}$, and variance $\sigma^2 > 0$, the normal probability density function is

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{(\mu - x)^2}{-2\sigma^2}\right]. \tag{9}$$

Maximizing the normal probability is equivalent to minimizing the negative log-likelihood, which simplifies to the square loss,

$$\ell_{\texttt{mean\_norm}}(\mu, x) = (\mu - x)^2. \tag{10}$$

For a segment from j to e ($1 \leq j \leq e \leq N$), we use the notation $W_{j,e} = W_j - W_{e-1}$ for the sum of the weights over the segment, and similarly for $X_{j,e}$ and $Y_{j,e}$. Then the mean on the segment from j to e can be computed in constant $O(1)$ time via

$$\mu^* = X_{j,e}/W_{j,e}. \tag{11}$$

For `distribution.str="meanvar_norm"`, the normal change in mean and variance model, the segment-specific parameter $\theta = [\mu, \sigma^2] \in \mathbb{R}^2$ is a vector with two values, the mean μ and variance σ^2. The Gaussian negative log likelihood is used as the loss function,

$$\ell_{\texttt{meanvar\_norm}}([\mu, \sigma^2], x) = [\log(2\pi\sigma^2) + (\mu - x)^2/\sigma^2]/2. \tag{12}$$

For a segment from j to e ($1 \leq j \leq e \leq N$), the variance estimate is computed in constant $O(1)$ time via

$$\sigma^{2*} = Y_{j,e}/W_{j,e} - \mu^*(\mu^* - 2X_{j,e}/W_{j,e}). \tag{13}$$

For `distribution.str="poisson"`, the Poisson model, the segment-specific parameter $\theta = \mu \geq 0$ is the rate. The Poisson probability mass function for a data value $x \in \{0, 1, 2, \dots\}$ is

$$\mu^x e^{-\mu}/(x!). \tag{14}$$

Maximizing the probability is equivalent to minimizing the negative log likelihood, which simplifies to the Poisson loss,

$$\ell_{\texttt{poisson}}(\mu, x) = \mu - x \log \mu. \tag{15}$$

The Poisson rate parameter μ is estimated via the sample mean (11).

3. Algorithm

This section gives details of the binary segmentation algorithm.

3.1. Efficient computation of optimal split point on a segment

The iterative algorithm involves computing the min loss over candidate segments, each of which is defined by a start j and end e position ($1 < j < e < N$).

$$L_{j,e} = \min_\theta \sum_{i=j}^{e} w_i \ell(\theta, x_i). \tag{16}$$

The minimization problem is over the model parameter θ.

Best loss difference. Since different segments have different loss values before and after splitting, the algorithm must choose the next segment to split by minimizing a loss difference value. Let $D_{j,e}(t) = L_{j,t} + L_{t+1,n} - L_{j,e} < 0$ be the loss difference after splitting segment (j, e) after t. The $L_{j,e}$ term is the loss before the split, and $L_{j,t} + L_{t+1,n}$ is the loss after the split, which is necessarily smaller. We define the best loss difference/split on segment (j, e) as

$$f_{j,e} = \min, \operatorname*{arg\,min}_{t \in \{j,\dots,e-1\}} D_{j,e}(t). \tag{17}$$

Splits:	one split, iteration k		S equal splits		S unequal splits	
operation:	insert	argmin	insert	argmin	insert	argmin
list	$O(1)$	$O(k)$	$O(S)$	$O(S^2)$	$O(S)$	$O(S)$
heap/multiset	$O(\log k)$	$O(1)$	$O(S \log S)$	$O(S)$	$O(S)$	$O(S)$
Compute loss	—		$O(N \log S)$		$O(NS)$	

Table 3: Asymptotic time complexity for container operations (insert new segment which could be split, argmin to find best segment to split next). At iteration k, list takes $O(k)$ time, whereas heap/multiset takes $O(\log k)$ time. Best case time for finding S splits in N data is $O(N \log S)$, which can be achieved by heap/multiset, because insert $O(S \log S)$ time is faster. However, best case time can not be achieved by list, because argmin $O(S^2)$ time is slower.

Efficient computation of optimal split point. To compute the loss differences $D_{j,e}(j), \dots, D_{j,e}(e-1)$, we need to compute the sequence of values $L_{j,j}, \dots, L_{j,e-1}$ (loss values for possible new segments before candidate split points), and the sequence of values $L_{j+1,e}, \dots, L_{e,e}$ (loss values for possible new segments after candidate split points). These sequences can be computed efficiently for the five loss functions that we implemented in `binsegRcpp`. The time complexity of computing these sequences depends on the loss function, and the size of the segment to split, $s = e - j + 1$. For the case of negative log likelihood of Normal and Poisson distributions, the cumulative sum trick can be used to compute these sequences in linear $O(s)$ time. For the case of the L1 loss and negative log likelihood of Laplace distribution, these sequences can be computed efficiently in log-linear $O(s \log s)$ time using a cumulative median algorithm. To implement the cumulative median, we used a modified version of the L1 loss minimization code proposed by Drouin *et al.* (2017).

3.2. Containers for efficient retreival of next segment to split

The binary segmentation algorithm needs to keep track of a set of segments that can be split, $\mathcal{S}_k$ for iteration k. Each segment $(j, e, d, t) \in \mathcal{S}_k$ is represented by the start/end positions (j, e) along with the corresponding min loss difference $d < 0$, as well as the best split point $t \in \{j, \dots, e-1\}$.

In binary segmentation, a segment is split in each iteration, which results in up to two new segments that must be considered to split. Computing the cost of splitting a segment of size s is $O(s \log s)$ time for L1/Laplace losses, and $O(s)$ time for the normal/Poisson losses. After computing the cost for two new segments, the cost of all splittable segments must be considered, in order to identify the min cost segment to split next. We propose using three different kinds of C++ Standard Template Library containers to represent $\mathcal{S}_k$ (set of splittable segments at iteration k). Each container has a different implementation of storage and retreival of the cost of segments which have previously been considered, but not yet split (Table 3). First, we propose using the `multiset` container (red-black tree) and `priority_queue` container (heap), which can be used to achieve best case time complexity. These containers keep segments sorted by their cost values, so offer constant $O(1)$ time retreival of the min cost segment to split next; insertion takes logarithmic $O(\log n)$ time in the number of segments n currently stored in the container.

List container as baseline. Second, we propose using the `list` container (doubly-linked list), which is used as a baseline, to demonstrate the benefits of the other containers. The `list` keeps segments stored in the order in which they were inserted, so offers constant $O(1)$ time insertion of new elements, and linear $O(n)$ time retreival of the min cost segment to split, for a container with n segments currently stored. In the best case run-time of binary segmentation, there are equal sized splits at each iteration, with two new segments that could be split, but only one that is split (and the other one which stays in the set $\mathcal{S}_k$). Therefore the best case size of the set of splittable segments at iteration k is linear, $|\mathcal{S}_k| = O(k)$, and the `list` container therefore takes $O(k)$ time to find the next segment to split. Going up to S segments, the list therefore takes $O(S^2)$ time, which can dominate the overall time for large S, and prevent the code from achieving the best case $O(N \log S)$ time.

3.3. Algorithm pseudo-code

This section gives the update rules for the binary segmentation algorithm.

Initialization. The initialization of the algorithm is as follows. Let $\mathcal{L}_1 = L_{1,n}$ be the loss with one segment, and let $f_{1,n}$ be the best loss difference and split point for the entire data. We initialize the set of segments to split as $\mathcal{S}_1 = \{(1, N, f_{1,n})\}$.

Recursive update rules. Let $S \in \{2, \dots, N\}$ be the user-specified max number of segments to consider. The binary segmentation algorithm recursively computes the following quantities for each iteration $k \in \{2, \dots, S\}$. First, the best segment to split is computed by greedy minimization of the loss decrease over all splittable segments.

$$j_k^*, e_k^*, d_k^*, t_k^* = \operatorname*{arg\,min}_{(j,e,d,t)\in\mathcal{S}_{k-1}} d. \tag{18}$$

Ideally, the minimization above takes $O(1)$ constant time (independent of the number of segments that could be split), if the set of splittable segments is stored in an efficient container (for example, C++ Standard Template Library multiset or priority_queue). As a baseline, we also consider a linked list, which results in a sub-optimal $O(k)$ linear time search at iteration k (see Section 3.2 for details).

In the second update at iteration k, the total loss is computed by adding the best loss decrease d_k^* to the best loss L_{k-1} at the previous iteration,

$$\mathcal{L}_k = \mathcal{L}_{k-1} + d_k^*. \tag{19}$$

Third, we construct a set of two new segments to consider:

$$\mathcal{N}_k = \{(j_k^*, t_k^*), (t_k^* + 1, e_k^*)\}. \tag{20}$$

New segments in N_k could be too small to split, if there is only one data point on that segment. We therefore consider only the segments that are large enough to be split:

$$\mathcal{V}_k = \{(j, e, f_{j,e}) \mid (j, e) \in \mathcal{N}_k,\ j < e\}. \tag{21}$$

Finally, we update the set of splittable segments, by removing the currently split segment, and inserting the new segments that could be split:

$$\mathcal{S}_k = [\mathcal{S}_{k-1} \setminus (j_k^*, e_k^*, d_k^*, t_k^*)] \cup \mathcal{V}_k. \tag{22}$$

For an efficient implementation, the time complexity of these remove/insert operations should be sub-linear in the container size, which is true of all three containers that we consider. The list container offers constant time insertion/removal; the multiset and priority_queue containers offer constant time removal and log-time insertion.

3.4. Modification for min segment length parameter

Sometimes there is prior knowledge that there should be no segments with fewer data points than $m \in \{1, 2, \dots\}$. In that case, m is referred to as a min segment length parameter, and there are two simple modifications to the equations above. First, the best split (17) is re-defined as

$$f_{j,e}^m = \min, \operatorname*{arg\,min}_{t\in\{j+m-1,\dots,e-m\}} D_{j,e}(t). \tag{23}$$

Second, the set of splittable segments (21) is re-defined as

$$\mathcal{V}_k^m = \{(j, e, f_{j,e}) \mid (j, e) \in \mathcal{N}_k,\, e - j + 1 \geq 2m\}. \tag{24}$$

4. Demonstration and comparison of code

In this section, we show example code for computing binary segmentation models using several software packages. We aim to emphasize how **binsegRcpp** makes it easy to retreive segment-specific model parameters. To simplify the discussion in this section, we use “segment mean” to represent arbitrary segment-specific model parameters. For example, segment-specific model parameters can be the mean/variance of the normal distribution, or median/scale of the Laplace distribution.

A unique feature that is provided by **binsegRcpp**, but not provided by other implementations of binary segmentation, is efficient storage and retrieval of arbitrary segment mean parameters. In contrast, `changepoint::cpt.mean` returns only the segment means for the selected model, and other implementations do not return any segment means (but the returned change-point variables could be used to compute estimates).

For example, we consider a simple example with six data points.

```
R> ex6_df <- data.frame(value = c(1, -7, 8, 10, 2, 4))
```

The code below can be used to compute models up to a max of 4 segments, using the square loss (normal change in mean, constant variance).

```
R> ex6_binsegRcpp <- binsegRcpp::binseg(
+    "mean_norm", ex6_df$value, max.segments = 4)
R> ex6_binsegRcpp$splits[, .(segments, end, before.mean, after.mean,
+    invalidates.index, invalidates.after)]
```

```
   segments   end before.mean after.mean invalidates.index invalidates.after
      <int> <int>       <num>      <num>             <int>             <int>
1:        1     6           3         NA                NA                NA
2:        2     2          -3          6                 1                 0
3:        3     4           9          3                 2                 1
4:        4     1           1         -7                 2                 0
```

The output above is a table with four rows, one for each model size. For a max number of segments S, this representation of model parameters uses S rows, so is asymptotically linear space, $O(S)$. The first row represents the trivial model: one segment across all data points, with `end=6` equal to the index of the last data point. The model parameter of the trivial model is `before.mean=3`, which is the mean of all of the data. The other rows represent the segments that result from the recursive splitting algorithm. For example, in the second row, `end=2` means that the first change-point results in a split between data points 2 and 3. The `before.mean = -3` is the mean up to data point 2, and `after.mean=6` is the mean from data point 3 to the end. The other columns, `invalidates.index` and `invalidates.after`, indicate the previous means which are no longer valid, for models with the given split. For example, in the second row, `invalidates.index=1` identifies the first row, and `invalidates.after=0` identifies the mean before the split (not after), which implies that `before.mean=3` is no longer valid, after performing the split in the second row. In the third row, we see a new split at `end=4`, resulting in new parameters `before.mean=9` and `after.mean=3`, which invalidate the previous mean in row 2 after the split (`after.mean=6`), while the other mean in row 2 remains valid (`before.mean = -3`).

In general, after running binary segmentation up to S segments, the model parameters are stored in a table with S rows. For any number of segments $k \in \{1, \dots, S\}$, we can compute a table of k estimated means (one row for each segment), using the following algorithm. First, consider only the first k rows of the `splits` table. Next, set values in `before.mean` and `after.mean` columns to missing (`NA`), based on values in columns `invalidates.index` and `invalidates.after`. Finally, sort the `end` values, and use the corresponding non-missing mean values. For k segments, this algorithm is $O(k \log k)$ time due to the sort. It is implemented in the `coef` method, as shown in the example code below:

```
R> (ex6_segments <- coef(ex6_binsegRcpp, 2:4))
```

```
   segments start   end start.pos end.pos  mean
      <int> <int> <int>     <num>   <num> <num>
1:        2     1     2       0.5     2.5    -3
2:        2     3     6       2.5     6.5     6
3:        3     1     2       0.5     2.5    -3
4:        3     3     4       2.5     4.5     9
5:        3     5     6       4.5     6.5     3
6:        4     1     1       0.5     1.5     1
7:        4     2     2       1.5     2.5    -7
8:        4     3     4       2.5     4.5     9
9:        4     5     6       4.5     6.5     3
```

The table above contains one row per segment, for three model sizes (two to four segments). It can be used to plot the segment means and change-point variables, using the code below.

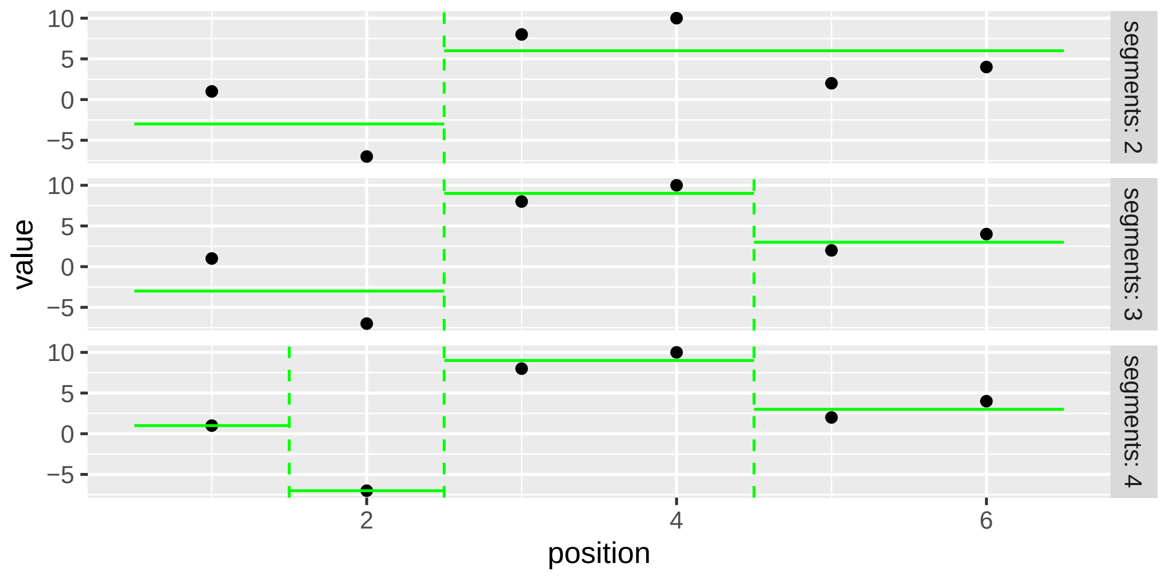

Figure 1: Change-point models with 2 to 4 segments computed by **binsegRcpp**. Data points are shown in black, along with the model in green (dashed vertical change-points, and solid horizontal segment means).

```
R> library(ggplot2)
R> ex6_df$position <- 1:nrow(ex6_df)
R> ggplot() + geom_point(aes(position, value), data = ex6_df) +
+    geom_segment(aes(start.pos, y = mean, xend = end.pos, yend = mean),
+      color = "green", data = ex6_segments) +
+    geom_vline(aes(xintercept = start.pos), color="green",
+      linetype = "dashed", data = ex6_segments[1 < start]) +
+    facet_grid(segments ~ ., labeller = label_both)
```

The code above results in Figure 1.

4.1. Comparison with changepoint

To attempt the analogous computation with **changepoint**, we can use the code below.

```
R> ex6_changepoint <- changepoint::cpt.mean(ex6_df$value,
+    method = "BinSeg", Q = 3, penalty = "Manual", pen.value = 0)
R> changepoint::cpts.full(ex6_changepoint)

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

The output above is the matrix of change-points, one row for each iteration of binary segmentation. One issue with the output above is that the change-points are incorrect in rows

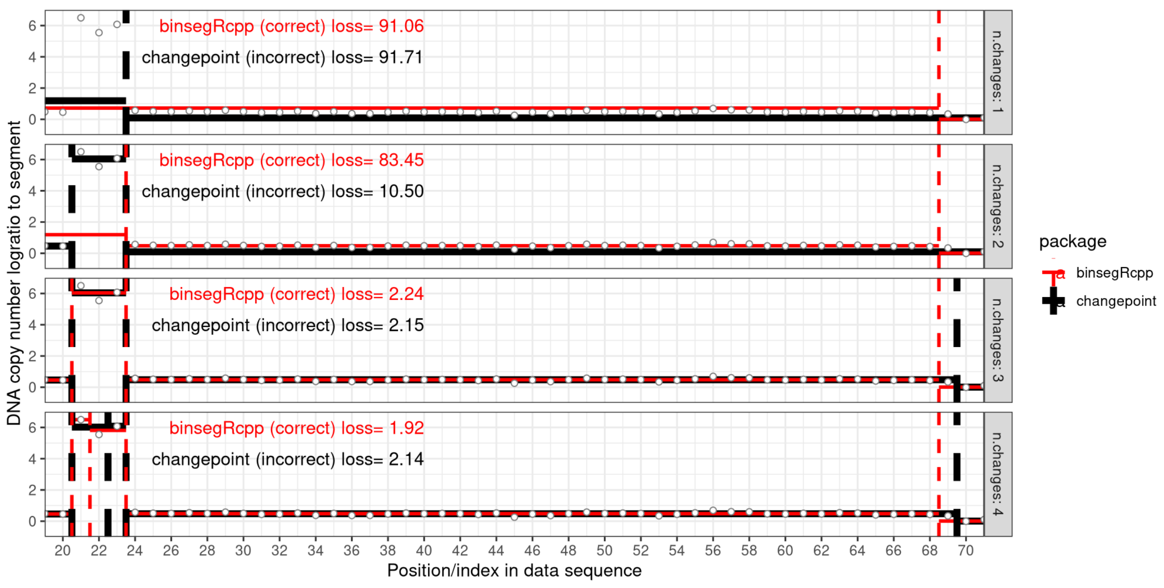

Figure 2: For a real cancer DNA copy number data set with 273 observations, we show the first four change-points models for two different R packages that provide code for binary segmentation: **binsegRcpp** and **changepoint**.

2 and 3. For example, row 2 represents the model with two change-points, which appear as 0 and 2 in the matrix above. However, for the model with two change-points, the expected result is segments that end at data points 2, 4, and 6. These results indicate that the implementation of binary segmentation in **changepoint** can yield incorrect change-point detection results. Incorrect results can also be observed in real data sets, such as in Figure 2, which shows that **changepoint** computes models with sub-optimal change-points and loss values.

Another issue with **changepoint** is that the storage of change-points is relatively inefficient. For a maximum number of change-points Q, it uses a $Q \times Q$ matrix of integers to represent the change-points, so the storage space is quadratic $O(Q^2)$. In contrast, **binsegRcpp** uses only linear space, $O(Q)$, which is asymptotically more efficient. Therefore, **changepoint** can only handle models with relatively small numbers of change-points, whereas **binsegRcpp** can efficiently store arbitrarily large change-point models.

A final issue is illustrated via the code below,

```
R> changepoint::param.est(ex6_changepoint)
```

```
$mean
[1]  1 -3  6
```

The result above shows the estimated segment means, for which there are two issues. First, there are only three values, so these should be the segment means for the model with two change-points. However, these three values are incorrect (the correct segment means are -3, 9, and 3). Another issue is that the segment means are only reported for one model size, and it is not possible to retreive means for other model sizes.

4.2. Comparison with wbs

Like **binsegRcpp**, the **wbs** package provides a correct implementation of binary segmentation. However, it is missing some key features, such as the ability to specify a max number of segments. The **wbs** code below computes the full path of binary segmentation models.

```
R> ex6_wbs <- wbs::sbs(ex6_df$value)
R> data.table::data.table(ex6_wbs$res)[order(-min.th)]
```

```
       s     e   cpt      CUSUM    min.th scale
   <num> <num> <num>      <num>     <num> <num>
1:     1     6     2 -10.392305 10.392305     1
2:     3     6     4   6.000000  6.000000     2
3:     1     2     1   5.656854  5.656854     2
4:     5     6     5  -1.414214  1.414214     3
5:     3     4     3  -1.414214  1.414214     3
```

The `res` table above is similar to the `splits` table from **binsegRcpp**. It has one row per split in binary segmentation, so there are five rows in this example (because there are six data points). The `cpt` column is the index of the last data point before the detected change-point, which is consistent with the `end` column from **binsegRcpp**. The **wbs** package implements binary segmentation using recursive function calls in C (depth-first search, which means it is impossible to specify a max number of segments). To compute the change-points for some number of change-points Q, the `res` table must be sorted, and only the top Q rows be kept. For example, the model with $Q = 2$ change-points can be represented using the first two rows of the table above (change-points at 2 and 4). However, the **wbs** package does not support retreival of segment-specific model parameters such as segment means. If the user wants segment mean estimates, they must be computed using the returned change-points.

4.3. Cross-validation

A unique feature of **binsegRcpp** is efficient cross-validation, which can be useful for selecting the number of change-points. The data input to `binseg()` is called the train set, which is divided into a subtrain set (used to compute model parameters) and a validation set (used to evaluate the computed model parameters). The assignment of data to each set is controlled by the `is.validation.vec` argument, which must be a logical vector (same length as the data to segment). For example, we can simulate a data set with two change-points:

```
R> set.seed(8)
R> two_changes <- c(rnorm(7, 1), rnorm(10, 3), rnorm(5, 0))
```

Then we can create a vector to indicate that every other data point should be assigned to either subtrain or validation:

```
R> (is.validation.vec <- rep(c(TRUE, FALSE), l=length(two_changes)))
```

```
 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[13]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

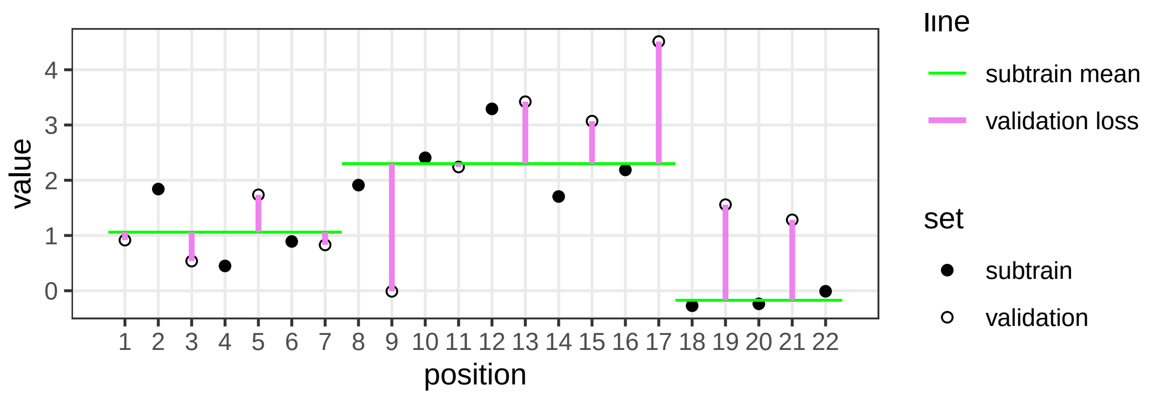

Figure 3: Change-point model parameters (segment means and change-point positions) are computed using the subtrain set; validation set is used for selecting the number of change-points.

Then we can run binary segmentation with cross-validation via the code below:

```
R> valid_fit <- binsegRcpp::binseg("mean_norm", two_changes,
+    is.validation.vec = is.validation.vec)
R> valid_fit
```

```
binary segmentation model:
    segments   end          loss validation.loss
       <int> <int>         <num>           <num>
 1:        1    11  1.424746e+01        21.89464
 2:        2     8  5.446692e+00        23.44001
 3:        3     3  2.563496e+00        18.00127
 4:        4     1  1.651273e+00        20.91210
 5:        5     6  1.232687e+00        24.03317
 6:        6     5  3.771919e-01        21.40443
 7:        7     4  2.546014e-01        20.41229
 8:        8     7  1.387041e-01        19.83415
 9:        9     2  4.060015e-02        20.33371
10:       10    10  5.868399e-04        20.86757
11:       11     9 -6.702972e-15        20.87759
```

The output above includes a table with one row per model size, and a column for the validation loss, which is minimized at 3 segments (as expected). The validation loss is computed as the total loss of the segment means with respect to the overlapping validation data points, as shown in Figure 3.

5. Empirical time complexity analysis

This section presents results from empirical timings of the proposed **binsegRcpp** package, along with baselines.

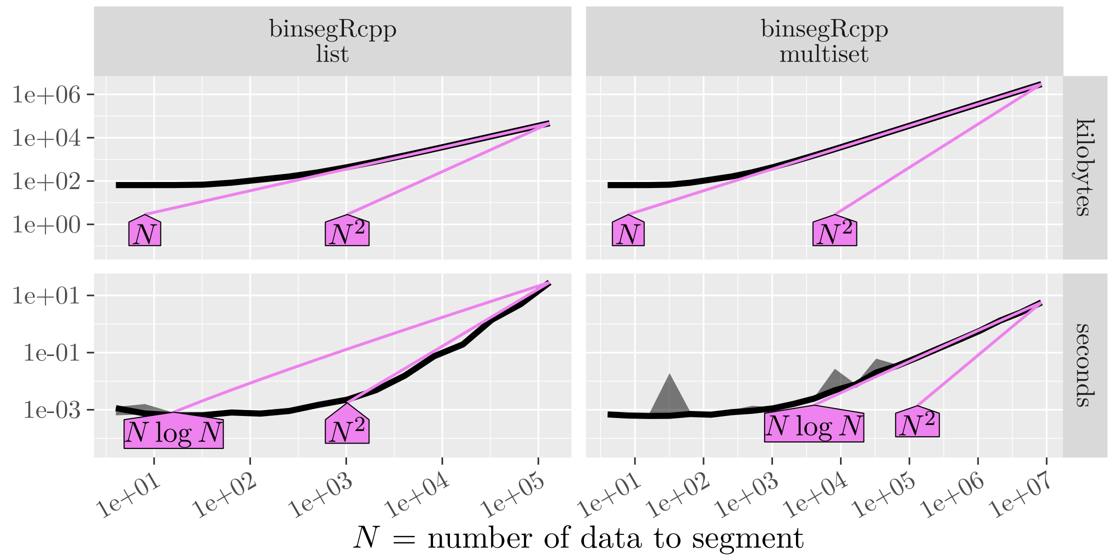

Figure 4: Estimating best-case asymptotic complexity for time (seconds) and memory (kilobytes). Black curves are empirical measurements for **binsegRcpp** with two containers, and violet curves are references. These data suggest that both list and multiset containers are $O(N)$ space list is $O(N^2)$ time and multiset is $O(N \log N)$ time. Square loss and best case data were used (integers from 1 to N, result in equal splits).

5.1. Best case time using multiset but not list container

First, we wanted to empirically validate the expected time complexity of the proposed C++ code using the Standard Template Library containers (Table 3). We expected the multiset container to achieve the best case $O(N \log S)$ time for N data and S segments; we expected the list container to be slower, $O(S^2)$ time, even for best case data. To test this expectation empirically, we used **binsegRcpp** with synthetic data sequences of varying sizes $N \in \{2, 4, 8, \dots\}$, and a maximum number of segments $S = N/2$. For each N, the data sequence we used was $1, \dots, N$, which results in equal splits with the square loss: one segment of size N split into two of size $N/2$, each of which is split into two of size $N/4$, etc. In this simulation, $S = O(N)$, so we expected time complexity of $O(N \log N)$ for the multiset, and $O(N^2)$ for the list. In Figure 4, it is clear that the empirical timings using the list container closely align with the $O(N^2)$ reference line, whereas the empirical timings using the multiset container closely align with the $O(N \log N)$ reference line. These data indicate that **binsegRcpp** with the multiset container indeed achieves the best case asymptotic time complexity, $O(N \log S)$. Futhermore, these data indicate that the best case time complexity can not be attained, if list container is used. In addition, Figure 4 clearly shows the linear space complexity of both containers, $O(N)$.

5.2. Other packages which do not achieve best case time

Next, we wanted to empirically verify the best case asymptotic time complexity of two baselines, **changepoint** and **ruptures**. We expected that these baselines should have the asymptotically optimal time complexity, $O(N \log S)$ (only different from **binsegRcpp** by constant

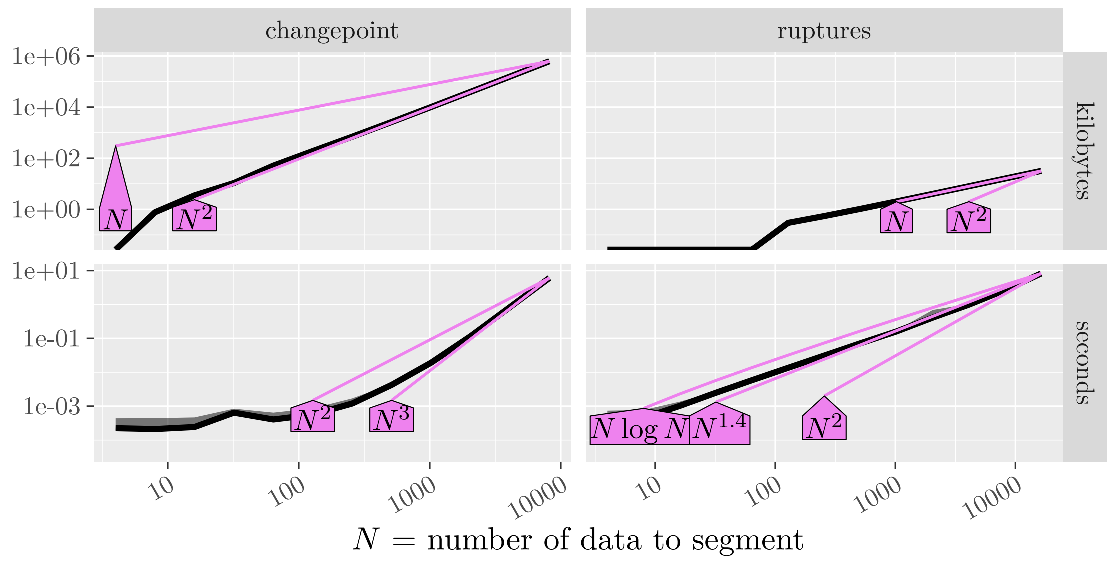

Figure 5: Estimating best-case asymptotic complexity for time (seconds) and memory (kilobytes). Black curves are empirical measurements for **changepoint** and **ruptures**, and violet curves are references. These data suggest that **changepoint** is $O(N^2)$ space and $O(N^3)$ time; **ruptures** is $O(N)$ space and $\approx O(N^{1.4})$ time. Square loss and best case data were used (integers from 1 to N, result in equal splits).

factors). To test this hypothesis, we ran these baselines in the same computational experiment described in the previous paragraph (square loss, best case data, $S = N/2$). We observed in Figure 5 that **changepoint** takes $O(N^2)$ space, which is a larger asymptotic complexity class than expected, $O(N)$. Similarly, we observed in Figure 5 that **changepoint** takes $O(N^3)$ time, which is again larger than the expected $O(N \log N)$ complexity. We observed in Figure 5 that **ruptures** empirical timings align well with a $O(N^{1.4})$ asymptotic reference, which is clearly larger than $O(N \log N)$, but smaller than $O(N^2)$. Contrary to our expectations, these data indicate that the implementations of binary segmentation in both **changepoint** and **ruptures** do not achieve the best case asymptotic time complexity.

5.3. Comparing timings and throughput with baselines

In Figure 6, we show the previously described empirical timings (Figures 4–5) on the same axes, for comparison. Whereas there are not very large timing differences for small data sizes (all work in less than one second for up to 1000 data), there are large differences for data sizez of several thousand data or larger. We highlight the estimated throughput at 5 seconds, which is the data size N that each package can handle in that time limit. It can be seen that **binsegRcpp** has about the same timings, using either `multiset` or `priority_queue` containers (as expected because they both offer log-time retreival of the next segment to split). Also, **binsegRcpp** with `list` container is about 100× slower (as expected because of the linear time retreival of the next segment to split). It can be seen that **ruptures** and **changepoint** are significantly slower, with about 1000× smaller throughput than **binsegRcpp** (thousands versus millions of data possible in a time limit of 5 seconds). These data indicate that for

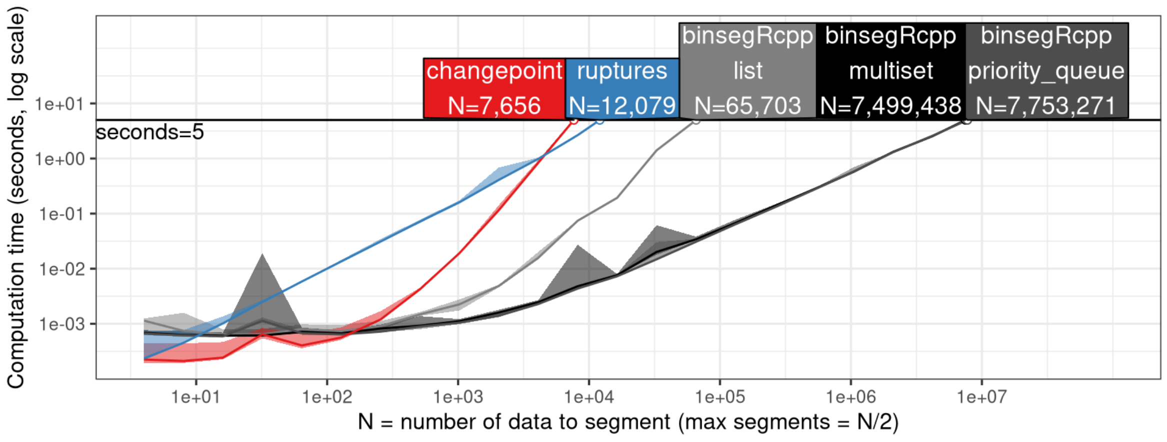

Figure 6: Comparing best-case computation time of **binsegRcpp** with three containers, and two baselines: **changepoint** and **ruptures**. Square loss and best case data were used (integers from 1 to N, result in equal splits).

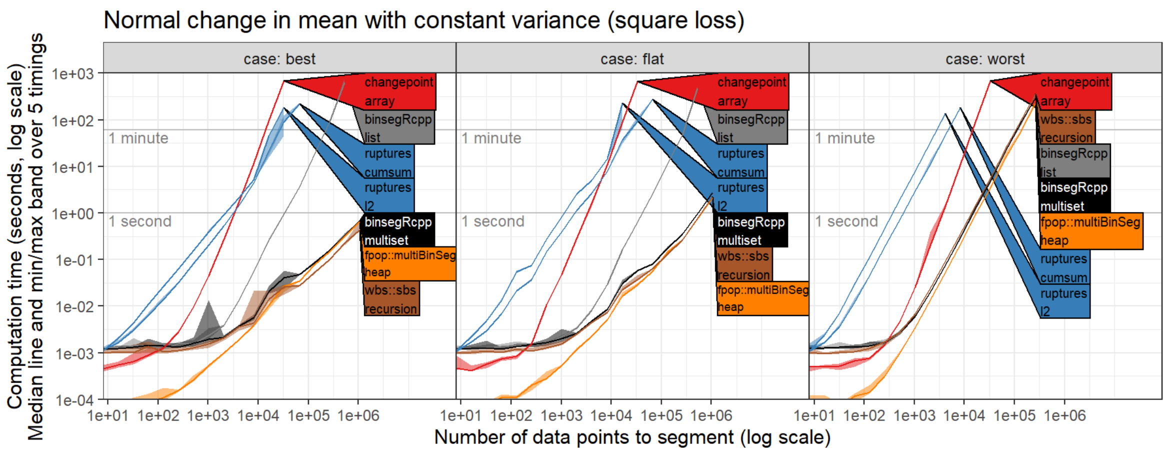

Figure 7: Timings using square loss.

large data (several thousand or more), **binsegRcpp** is preferable for speed.

5.4. Other packages which achieve best case time

We conducted empirical timings on other implementations of binary segmentation (`wbs::sbs`, `fpop::multiBinSeg`, `blockcpd::heap`). In Figure 7, we show timings for the square loss in three different data sets (best case, worst case, flat). It is clear that `fpop::multiBinSeg` and `wbs::sbs` have very similar asymptotic timings as **binsegRcpp**. In the worst case, all packages show the same slope (indicating quadratic time), except **changepoint** (which has a larger slope, indicating cubic time).

We also investigated the Poisson model (Figure 9), and the normal change in mean and variance model (Figure 8), which are implemented by `blockcpd::heap`. It is clear that `blockcpd::heap` has the same asymptotic slope, but much larger constant factors, when compared with **binsegRcpp**.

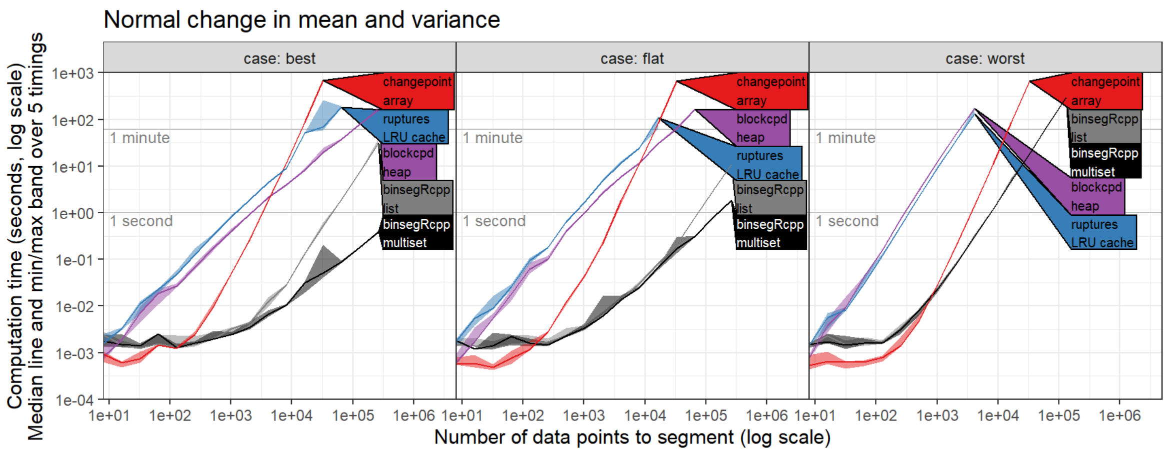

Figure 8: Timings using Gaussian change in mean and variance model.

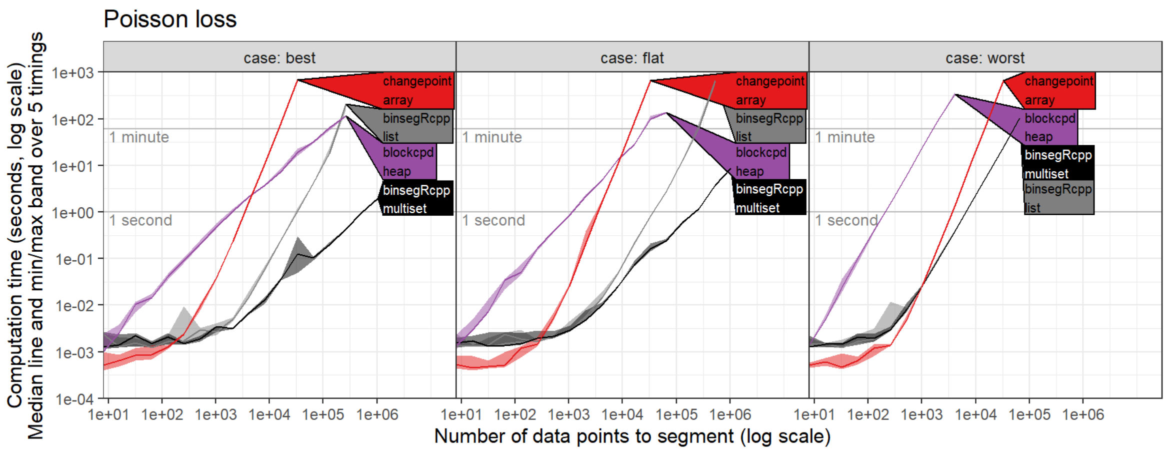

Figure 9: Timings using Poisson loss.

6. Summary and discussion

▮ As usual …

References

Auger I, Lawrence C (1989). "Algorithms for the optimal identification of segment neighborhoods." *Bull Math Biol*, **51**, 39–54.

Baranowski R, Fryzlewicz P (2019). *wbs: Wild Binary Segmentation for Multiple Change-Point Detection.* R package version 1.4, URL https://CRAN.R-project.org/package=wbs.

Drouin A, Hocking T, Laviolette F (2017). "Maximum Margin Interval Trees." In I Guyon, UV Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, R Garnett (eds.), *Advances* *in Neural Information Processing Systems 30*, pp. 4947–4956. Curran Associates, Inc. URL http://papers.nips.cc/paper/7080-maximum-margin-interval-trees.pdf.

Fotoohinasab A, Hocking T, Afghah F (2020). "A Graph-constrained Changepoint Detection Approach for ECG Segmentation." In *2020 42nd Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC)*, pp. 332–336. doi:10.1109/EMBC44109.2020.9175333.

Hocking TD (2024). "Finite Sample Complexity Analysis of Binary Segmentation." Preprint arXiv:2410.08654, under review at Canadian Journal of Statistics.

Hocking TD, Schleiermacher G, Janoueix-Lerosey I, Boeva V, Cappo J, Delattre O, Bach F, Vert JP (2013). "Learning smoothing models of copy number profiles using breakpoint annotations." *BMC Bioinformatics*, **14**(164).

Jewell SW, Hocking TD, Fearnhead P, Witten DM (2019). "Fast nonconvex deconvolution of calcium imaging data." *Biostatistics*, **21**(4), 709–726. ISSN 1465-4644. doi:10.1093/biostatistics/kxy083. https://academic.oup.com/biostatistics/article-pdf/21/4/709/33893963/kxy083.pdf.

Killick R, Eckley IA (2014). "changepoint: An R Package for Changepoint Analysis." *Journal of Statistical Software*, **58**(3), 1–19. URL https://www.jstatsoft.org/v58/i03/.

Killick R, Fearnhead P, Eckley IA (2012). "Optimal detection of changepoints with a linear computational cost." *Journal of the American Statistical Association*, **107**(500), 1590–1598.

Killick R, Haynes K, Eckley IA (2022). *changepoint: An R package for changepoint analysis.* R package version 2.2.3, URL https://CRAN.R-project.org/package=changepoint.

Li X, Zhang X (2024). "fastcpd: Fast Change Point Detection in R." Pre-print arXiv:2404.05933.

Liehrmann A, Rigaill G, Hocking TD (2021). "Increased peak detection accuracy in over-dispersed ChIP-seq data with supervised segmentation models." *BMC Bioinformatics*, **22**(323).

Maidstone R, Hocking T, Rigaill G, Fearnhead P (2017). "On optimal multiple changepoint algorithms for large data." *Statistics and Computing*, **27**. URL https://link.springer.com/article/10.1007/s11222-016-9636-3.

Prates L, Lemes RB, Hünemeier T, Leonardi F (2021). "Population based change-point detection for the identification of homozygosity islands." Pre-print arXiv:2111.10187.

Prates L, Leonardi F (2021). *blockcpd: Change Point Detection for Multiple Aligned Independent Time Series.* R package version 1.0.0.

Scott A, Knott M (1974). "A cluster analysis method for grouping means in the analysis of variance." *Biometrics*, **30**, 507–512.

Truong C, Oudre L, Vayatis N (2020). "Selective review of offline change point detection methods." *Signal Processing*, **167**, 107299. ISSN 0165-1684. doi:https://doi.org/10.1016/j.sigpro.2019.107299. URL https://www.sciencedirect.com/science/article/pii/S0165168419303494.

Truong C, Runge V (2024). “An Efficient Algorithm for Exact Segmentation of Large Compositional and Categorical Time Series.” *Stat*, **13**(4), e70012.

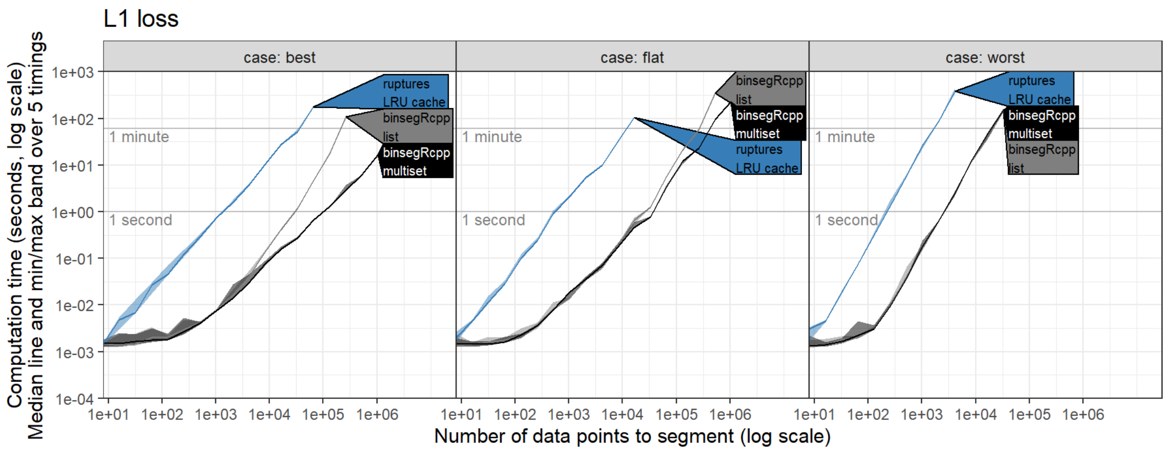

Figure 10: Timings using L1 loss.

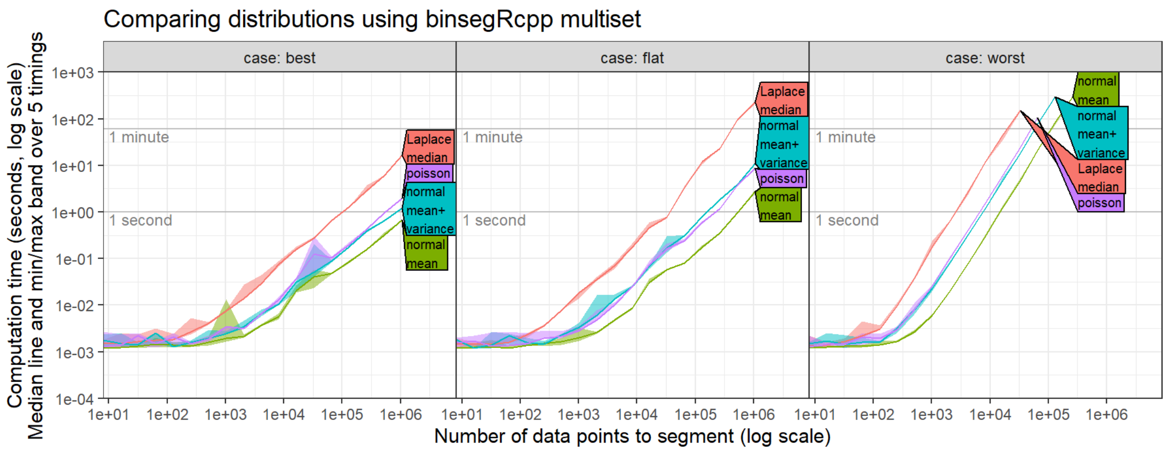

Figure 11: Timings using binsegRcpp multiset with several different loss functions.

A. More empirical timings

We also performed computational experiments involving the L1 loss (Laplace change in median), which was implemented by **ruptures**. Figure 10 shows that **ruptures** has slightly larger asymptotic slope, and much larger constant factors, than **binsegRcpp** (in the best case).

Figure 11 shows **binsegRcpp** timings for four loss functions, and three cases (best, worst, flat). It can be seen that the Laplace median model (L1 loss) has slightly larger asymptotic slope in the best and flat cases (but same quadratic slope in the worst case).

Affiliation:

Toby Dylan Hocking
Université de Sherbrooke
E-mail: Toby.Hocking@R-project.org

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