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 K segments: asymptotically $O(NK)$ time in the worst case, and $O(N \log K)$ time in the best case. In practice, other 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

TODO `fpop::multiBinSeg` (Maidstone, Hocking, Rigaill, and Fearnhead 2017).

TODO `wbs::sbs` (Baranowski and Fryzlewicz 2019).

TODO `changepoint::cpt.mean(method="BinSeg")` (Killick and Eckley 2014; Killick, Haynes, and Eckley 2022).

TODO `ruptures.Binseg` (Truong, Oudre, and Vayatis 2020).

TODO `fastcpd` package (Li and Zhang 2024) (nice related work table).

TODO **blockcpd** package TODO (Prates and Leonardi 2021a,b) .

2. Models and software

We assume there is a sequence of N data $x_1, \ldots, x_N \in \mathbb{R}$, in which we want to find abrupt

changes. We assume that each data point $i \in \{1, \dots, N\}$ has a corresponding weight $w_i > 0$. Binary segmentation can be interpreted as attempting to find model parameters θ which minimize a given weighted loss function $w_i \ell(\theta, x_i)$. 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.

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

$$L_{j,e} = \min_{\theta} \sum_{i=j}^{e} w_i \ell(\theta, x_i). \tag{1}$$

The minimization problem is over the model parameter θ (segment-specific center and possibly scale).

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. Let $\delta_{j,e}, \tau_{j,e} = \min, \arg\min_{t \in \{j,\dots,e-1\}} D_{j,e}(t)$ be the best loss difference/split on segment (j, e).

Efficient computation. 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 the cumulative median algorithm of Drouin, Hocking, and Laviolette (2017).

2.1. Algorithm pseudo-code

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 $t \in \{j, \dots, e-1\}$.

Initialization. The initialization of the algorithm is as follows. Let $\mathcal{L}_1 = L_{1,n}$ be the loss with one segment, and let $\delta_1, t_1 = f_{1,n}$ be the best loss difference and split point for the entire data. We initialize the set of segments to split as $\mathcal{S}_1 = \{(1, N, d_1, t_1)\}$.

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

$$j_k^*, e_k^*, d_k^*, t_k^* = \operatorname*{arg\,min}_{(j,e,d,t)\in\mathcal{S}_{k-1}} d. \tag{2}$$

Ideally, the minimization above takes $O(1)$ constant time, if the set of splittable segments is a red-black tree (C++ STL multiset). As a baseline, we also consider a linked list, which results in a sub-optimal $O(k)$ linear time search (see Section **??** for details).

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

$$\mathcal{L}_k = \mathcal{L}_{k-1} + d_k^*. \tag{3}$$

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

$$\mathcal{N}_k = \{(j_k^*, t_k^*), (t_k^* + 1, e_k^*)\}. \tag{4}$$

Fourth, we subset the two new segments, to consider only the segments that could possibly be split:

$$\mathcal{V}_k = \{(j, e, f_{j,e}) \mid (j, e) \in \mathcal{N}_k,\ j < e\}. \tag{5}$$

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

$$\mathcal{S}_k = [\mathcal{S}_{k-1} \setminus (j_k^*, e_k^*, d_k^*, t_k^*)] \cup \mathcal{V}_k. \tag{6}$$

TODO time.

2.2. 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 (**??**) is re-defined as

$$f_{j,e}^m = \min, \operatorname*{arg\,min}_{t\in\{j+m-1,\dots,e-m\}} D_{j,e}(t). \tag{7}$$

Second, the set of splittable segments (5) is re-defined as

$$\mathcal{V}_k^m = \{(j, e, f_{j,e}) \mid (j, e) \in \mathcal{N}_k,\ e - j + 1 \geq 2m\}. \tag{8}$$

2.3. C++ features

We used several features of C++ to implement **binsegRcpp** efficiently. In binary segmentation, a segment is split in each iteration, which results in up to two new segments that must be considered to split (see Section **??** for details). Computing the cost of splitting a segment of size N is $O(N \log N)$ time for L1/Laplace losses, and $O(N)$ 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 therefore propose using two different kinds of C++ Standard Template Library containers to store and retreive the cost of segments which have previously been considered, but not yet split (Table 4). First, we propose using the `multiset` container (red-black tree), which can be used to achieve best case time complexity. The `multiset` keeps segments sorted by their cost values, so offers 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. Second, we propose using the `list` container (doubly-linked list), which is used as a baseline, to demonstrate the benefits of the multiset container. 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.

A C++ virtual class `Container` is used to represent the methods that must be provided by either type of container: `insert`, `get_best`. We used a C macro, `CMAKER`, to declare the two sub-classes:

```
#define CMAKER(CONTAINER, INSERT, GET_BEST)...
CMAKER(multiset, insert, segment_container.begin())
CMAKER(list, push_back, std::min_element(
 segment_container.begin(),segment_container.end()))
```

The code above is expanded by the C pre-processor, to two declarations of `Container` sub-classes.

Another C macro is used to declare the loss functions which require the cumulative median:

```
#define ABS_DIST(NAME, VARIANCE)...
ABS_DIST(l1, false)
ABS_DIST(laplace, true)
```

The loss function which requires a variance estimate is the Laplace negative log likelihood; the simple L1 loss is obtained without a variance estimate. A `class absDistribution` implements the common operations for the two loss functions, including the cumulative median in $O(N \log N)$ time using the algorithm of Drouin *et al.* (2017).

Another C macro is used to declare the loss functions which require the cumulative sum:

Parameter	Description	Definition
$M =$`mean`	model parameter	T/W
`var`	model parameter	$S/W + M(M - 2T/W)$

Table 1: TODO

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

Table 2: TODO

```
#define CUM_DIST(NAME, COMPUTE, ERROR, VARIANCE)...
CUM_DIST(mean_norm, RSS, , false)
CUM_DIST(poisson,
 (mean>0) ? (mean*N - log(mean)*sum) : ( (sum==0) ? 0 : INFINITY ),
 if(round(value)!=value)return ERROR_DATA_MUST_BE_INTEGER_FOR_POISSON_LOSS;
 if(value < 0)return ERROR_DATA_MUST_BE_NON_NEGATIVE_FOR_POISSON_LOSS;,
 false)
CUM_DIST(meanvar_norm,
 (var>max_zero_var) ? (RSS/var+N*log(2*M_PI*var))/2 : INFINITY,
 , true)
```

The code above declares three loss functions based on the cumulative sum: `mean_norm` is the residual sum of squares (RSS), `poisson` loss is for count data, and `meanvar_norm` is the normal model with change in mean and variance. The `COMPUTE` parameter specifies the loss, in terms of six variables: `mean`, `var`, `N`, `sum`, `squares`, `max_zero_var` (Table **??**). These variables are computed in a `class CumDistribution` which contains the logic common to these loss functions (cumulative sum, etc). Note that the code supports cross-validation, in the sense that the input train set can be divided into a subtrain set (used to compute model parameters) and a validation set (used to evaluate the computed model parameters). The `mean` and `var` variables are model parameters, computed using the subtrain set, whereas the other variables depend on the data (subtrain or validation set). The `ERROR` parameter is an optional block of code, that is used to return an error status code, if there are unusual input data.

To define the loss functions based on the cumulative sum TODO

Additionally, the `CUM_DIST` and `ABS_DIST` macros generate code that populates a static `unordered_map` which can be queried to obtain a list of distributions which are supported by the C++ code:

```
R> binsegRcpp::get_distribution_info()$dist

[1] "meanvar_norm" "poisson"      "mean_norm"    "laplace"      "l1"
```

To illustrate the performence benefits of using log-time containers were used to efficiently , virtual classes, static variables, function pointers, templates, macros).

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
max segs	yes	yes	no	yes	yes	yes
dim	one	one	one	multi	multi	multi
correct	yes	no	yes	yes	yes	
losses	5	6	1	1	10	5
language	C++	C	C	C++	Python	C++
storage	STL multiset	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	$O(N \log N)$	$O(N^3)$	$O(N \log N)$	$O(N \log N)$	$> O(N \log N)$	$O(N \log N)$
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 3: Properties (rows) of five binary segmentation libraries (columns). Properties are weights (can specify observation-specific positive weights for the loss), 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 (time complexity in best case data), worst (time complexity in worst case data), CV (support for cross-validation), params (parameters are returned).

case/splits:	one		best/equal		worst/unequal	
operation:	insert	argmin	insert	argmin	insert	argmin
list	$O(1)$	$O(n)$	$O(S \log S)$	$O(S)$	$O(S)$	$O(S)$
heap/multiset	$O(\log n)$	$O(1)$	$O(S)$	$O(S^2)$	$O(S)$	$O(S)$
Find new split			$O(N \log S)$		$O(NS)$	

Table 4: container properties

Loss computation. We run each algorithm on `N.data` points up to `max.segments = max.changes+1`. `binsegRcpp::binseg` result is a list which contains a data table with `max.segments` rows and column `loss` that is the square loss. `changepoint::cpt.mean` result is a list of class `cpt.range` with method `logLik` which returns the square loss of one of the models. `wbs::sbs` result is a list which contains a data frame with `N.data-1` rows and `CUSUM` and `min.th` columns TODO. `fpop::multiBinSeg` result is a list with element `J.est`, which is a vector of `max.changes` square loss decrease values. `ruptures.Binseg.predict` result is a vector of segment ends for one model size, which can be passed to `sum_of_costs` method to compute the square loss.

Three packages have implemented the normal change in mean and variance model. `binsegRcpp::binseg` loss values are the normal negative log likelihood (NLL). The `changepoint::logLik` function returns two times the NLL. The `ruptures.Binseg` loss is related via this equation,

$$\text{NLL} = (\text{rupturesLoss} + N[1 + \log(2\pi)])/2 \tag{9}$$

3. Storage and retreival of 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))
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 M, this representation of model parameters uses M rows, so is asymptotically linear, $O(M)$. 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` and `after.mean=6` values are the means of the resulting segments, before and after that split. The other columns, `invalidates.index` and `invalidates.after`, indicate the previous means which are no longer valid, after 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`).

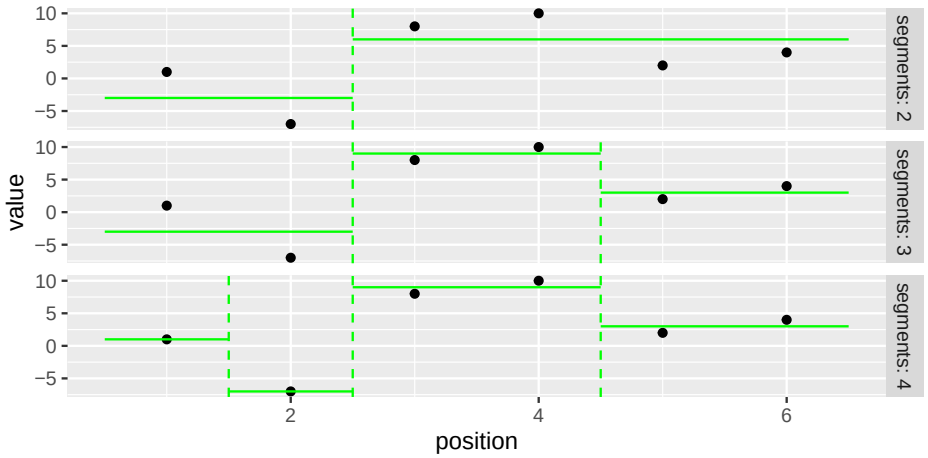
In general, for any number of segments K, we can compute a table of estimated means (one row for each segment), using the following algorithm. First, consider only the first K rows of the `splits` table. Next, set values in `before.mean` and `after.mean` columns to missing (`NA`), based on values in `invalidates.index` and `invalidates.after` columns. Finally, sort the `end` values, and use the corresponding non-missing mean values. For K segments, this algorithm is $O(KlogK)$ 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.

```
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 figure above shows the data points in black, along with the segmentation model in green (dashed vertical change-points, and solid horizontal segment means).

3.1. Comparison with changepoint

To do 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 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.

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 0, 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.

3.2. 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
```

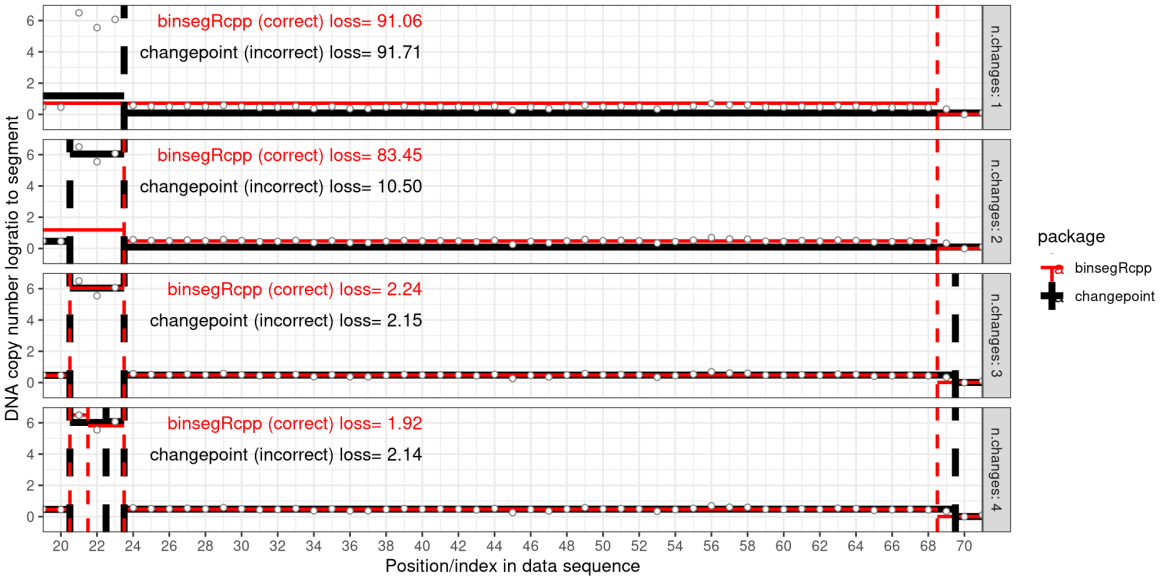


Figure 1: For a real cancer DNA copy number data set with 273 observations, we show the first four change-points models for two different R packages that provide code for binary segmentation: **binsegRcpp** and **changepoint**.

```
2:     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. Illustrations

TODO

```
R> data("quine", package = "MASS")
```

TODO

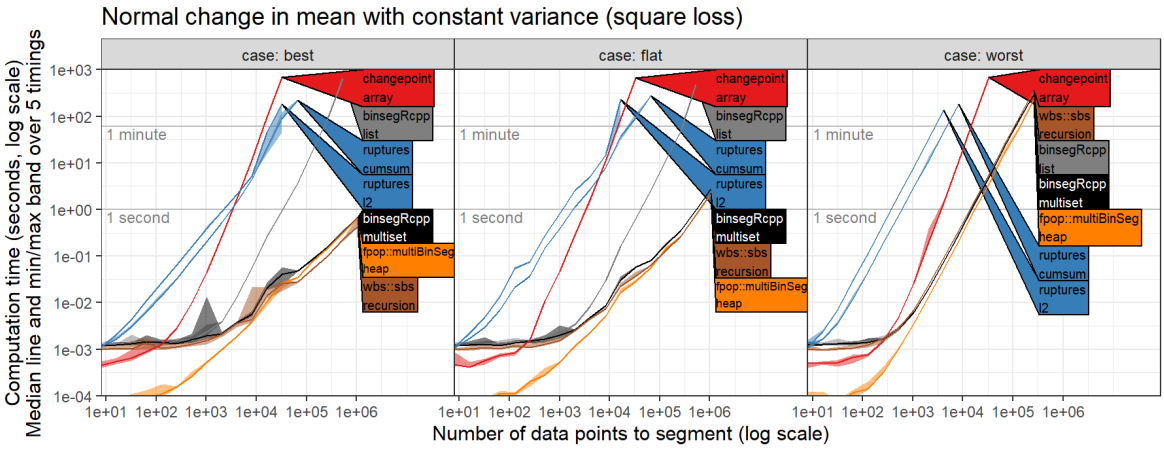


Figure 2: Timings using square loss.

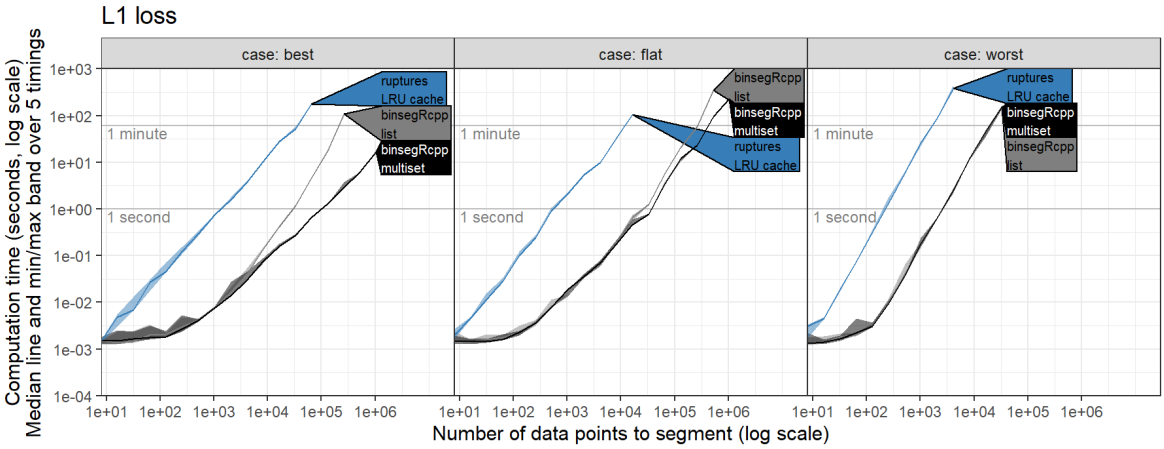


Figure 3: Timings using L1 loss.

5. Summary and discussion

- As usual ...

References

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.

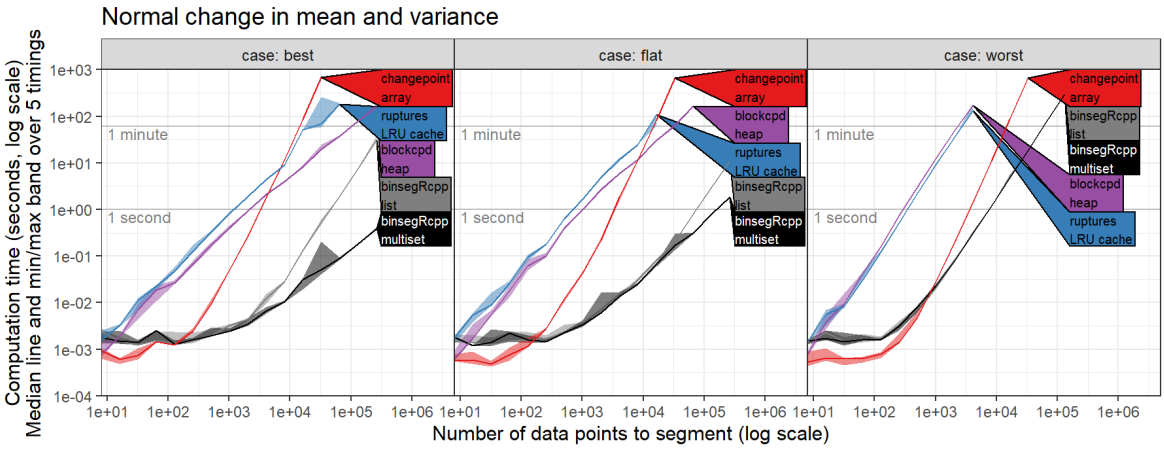


Figure 4: Timings using Gaussian change in mean and variance model.

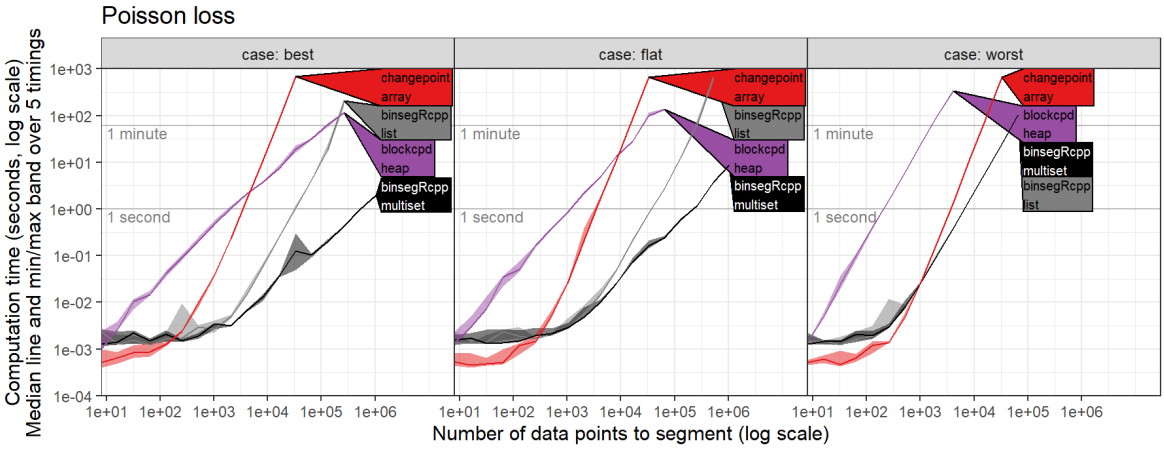


Figure 5: Timings using Poisson loss.

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, 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.

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

Prates L, Leonardi F (2021b). "TODO title." Pre-print arXiv:2111.10187.

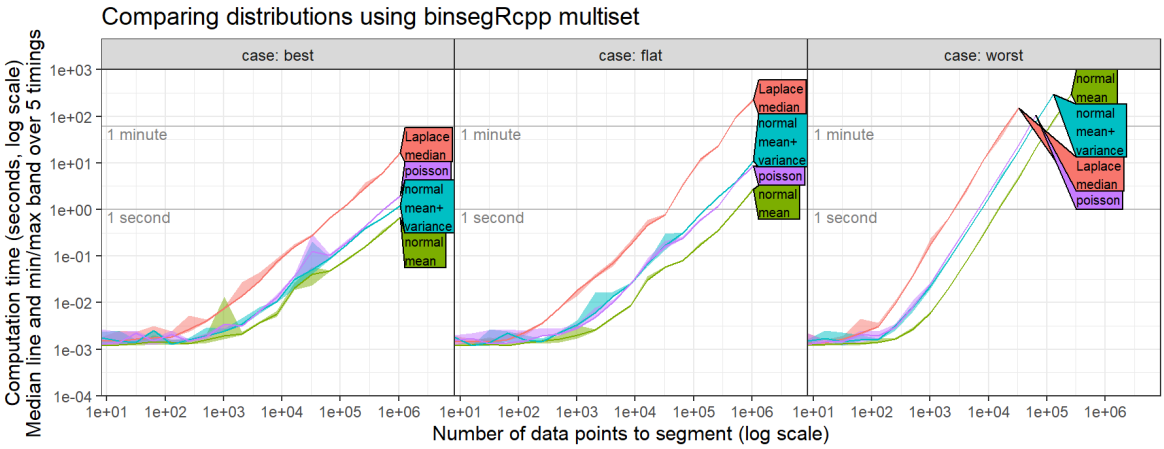


Figure 6: Timings using binsegRcpp multiset with several different loss functions.

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.

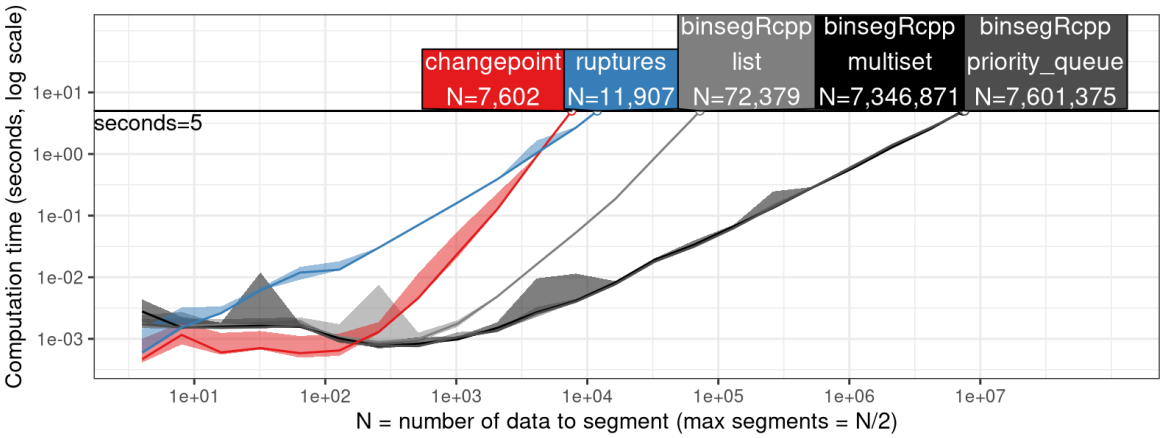


Figure 7: Comparing computation time of **binsegRcpp** with three containers, and two baselines: **changepoint** and **ruptures**.

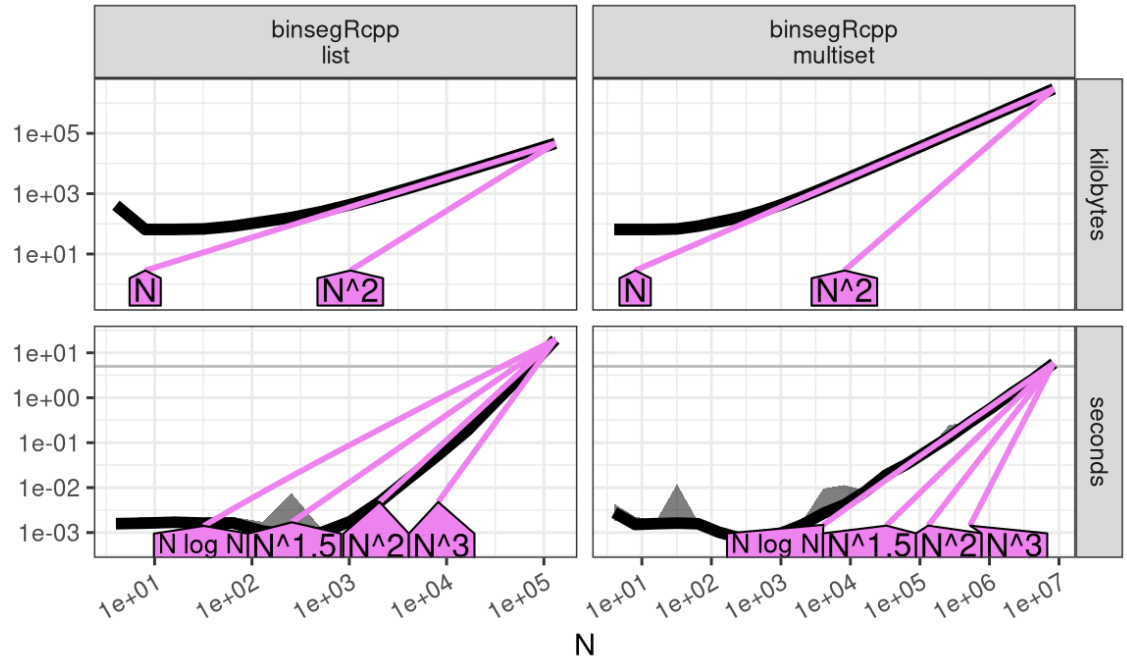


Figure 8: Estimation of asymptotic time (seconds) and memory (kilobytes) usage for **binsegRcpp**. Black curves are empirical measurements, and violet curves are references. These data suggest that both list and multiset containers are $O(N)$; list is $O(N^2)$ time and multiset is $O(N \log N)$ time.

A. More technical details

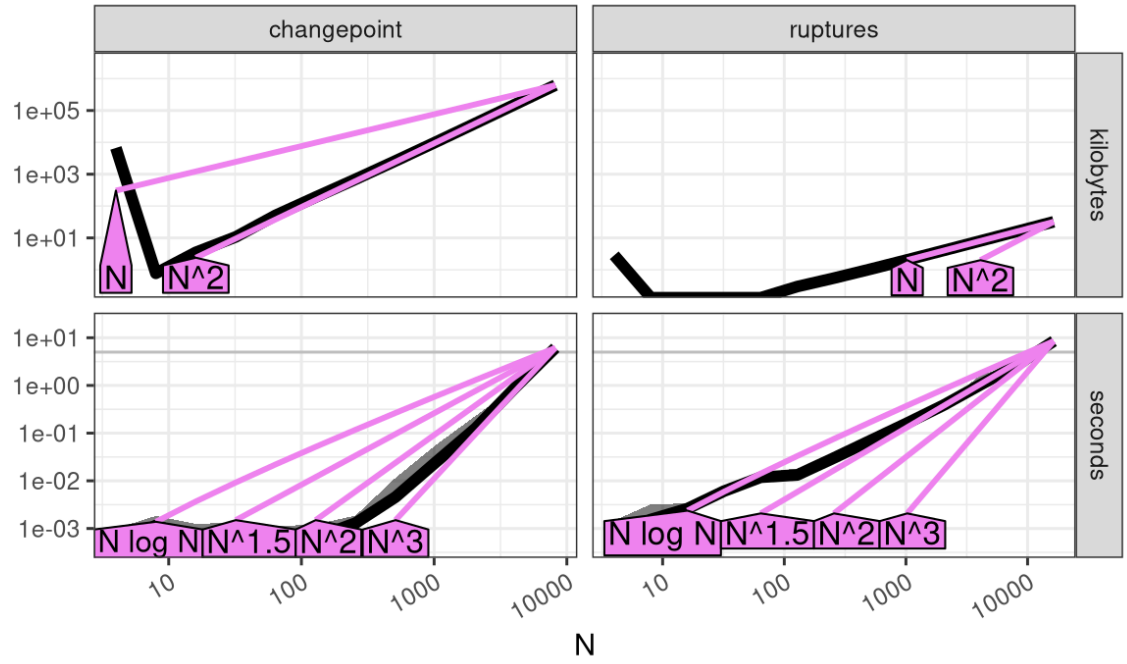


Figure 9: Estimation of asymptotic time (seconds) and memory (kilobytes) usage for **changepoint** and **ruptures**. Black curves are empirical measurements, and violet curves are references. These data suggest that **changepoint** is $O(N^2)$ space and $O(N^3)$ time; **ruptures** is $O(N)$ space and $O(N^{1.5})$ time.

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