cpop: Detecting changes in piecewise-linear signals

Paul Fearnhead
Lancaster Universtity

Daniel Grose
Lancaster University

Abstract

Changepoint detection is an important problem with applications across many application domains. There are many different types of changes that one may wish to detect, and a wide-range of algorithms and software for detecting them. However there are relatively few approaches for detecting changes-in-slope in the mean of a signal plus noise model. We describe the R package, **cpop**, available on the Comprehensive R Archive Network (CRAN). This package implements CPOP, a dynamic programming algorithm, to find the optimal set of changes that minimises an L_0 penalised cost, with the cost being a weighted residual sum of squares. The package has extended the CPOP algorithm so it can analyse data that is unevenly spaced, allow for heterogeneous noise variance, and allows for a grid of potential change locations to be different from the locations of the data points. There is also an implementation that uses the CROPS algorithm to detect all segmentations that are optimal as you vary the L_0 penalty for adding a change across a continuous range of values.

Keywords: changepoints, change-in-slope, dynamic programming, piecewise linear models, structural breaks.

1. Introduction

The detection of change in sequences of data is important across many applications, for example changes in volatility in finance (**?**), changes in genomic data that represent copy number variation (**?**), changes in calcium imaging data that correspond to neurons firing (**?**) or changes in climate data (**?**), amongst many others. Depending on the application, interest can be in detecting changes in different features of the data, and there has been a corresponding wide-range of methods that have been developed. See **?**, **?**, **?** and **?** for recent reviews of changepoint methods and their applications.

For some applications we have data on a piece-wise linear mean function, and we wish to detect the times at which the slope of the mean changes. This is the change-in-slope problem: see the top-left plot of Figure **??** for example simulated data. This is a particularly challenging problem for the following reasons. First, a simple approach to detecting changes in slope is to take first differences of the data, as this transforms a change-in-slope into a change-in-mean, and then apply one of the many methods for detecting changes in mean. However this removes much of the information in the data about the location of changes and such approach can perform poorly. This can be seen by comparing the raw data in the top-left plot of Figure **??** with the first differenced data in the top-right plot Figure **??**. By eye it is easy to see the rough location of the changes in slope in the former, but almost impossible to see any changes in mean in the latter. Running the PELT change-in-mean algorithm **?** on the first

differenced data leads to poor estimates of the location of any changes. Second, the most common approach to detecting multiple changepoints is to use binary segmentation (**?**) or one of its variants (**??**). These repeatedly apply a test for a single change-in-slope. However **?** shows that such binary segmentation methods do not work for the change-in-slope problem as if you fit a single change-in-slope to data simulated with multiple changes, it will often detect the change at a location near the middle of a segment between changes. Third, dynamic programming algorithms that minimise an L_0 penalised cost, such as Optimal Partitioning (**?**) or PELT (**?**) cannot be applied to the change-in-slope problem due to dependencies in the model across changepoints from the continuity of the mean at each change.

Despite these challenges, there are three methods developed specifically for detecting changes-in-slope: Trend-filtering (**??**) which minimises the residual sum of squares of fit to the data plus an L_1 penalty on the changes-in-slope; NOT (**?**) that repeatedly performs a test for a single change-in-slope on subsets of the data and combines the results using the narrowest-over-threshold procedure; and CPOP (**?**) which uses a novel variant of dynamic programming to minimise the residual sum of squares plus an L_0 penalty, i.e. a constant penalty for adding each change. The difference between the L_1 penalty of trend-filtering and the L_0 penalty of CPOP is important in practice: as the former allows one to fit a single change in slope with multiple changes of the same sign. This can lead to either over-fitting the number of changes, or, if a large enough penalty is used to detect the changes accurately, over-smoothing the mean function: see the bottom row of plots in Figure **??** for an example. The main difference between CPOP and NOT is that the former fits all changes simultaneously. See **?** for an empirical comparison of the three methods.

The purpose of this paper is to describe the **cpop** package, which is written in R, and implements the CPOP algorithm. The latest version of the package was developed in response to an applied challenge, see Section **??**, where the data was unevenly spaced and the noise was not homoscedastic, aspects that previous implementations of change-in-slope algorithms could not handle. How the CPOP algorithm is extended to deal with these features is described in Section **??**, together with allowing the locations of the changes in slope to not coincide with the observations. This latter aspect can be helpful in reducing the computational cost of the CPOP algorithm for high frequency data by, e.g., searching for segmentations that only allow changes at a smaller grid of possible locations. Section **??** describes the basic functionality of the package, with the extensions to allow for unevenly spaced, heteroscedastic data described, and to specify the grid of potential change locations, in Section **??**. This latter section also shows how to impose a minimum segment length and how to implement CPOP within the CROPS algorithm (**?**) to obtain all segmentations as we vary the value of L_0 penalty. An application of CPOP to analyse decay of spectra from ocean models is shown in Section **??**.

1.1. R Packages for Changepoint Detection

There are both many different types of change that one may wish to detect, and many different approaches to detecting multiple changes. Consequently there are a wide range of change algorithms with associated packages in R. For example the **changepoint** package (**?**) implements dynamic programming algorithms, such as PELT, for detecting changes in mean, variance or mean and variance. Other dynamic programming algorithms include **fpop** and **gfpop** (**?**) that implements the functional optimal partitioning algorithm (**?**) for detecting changes in mean, with the latter package allowing for flexibility as to how the mean changes

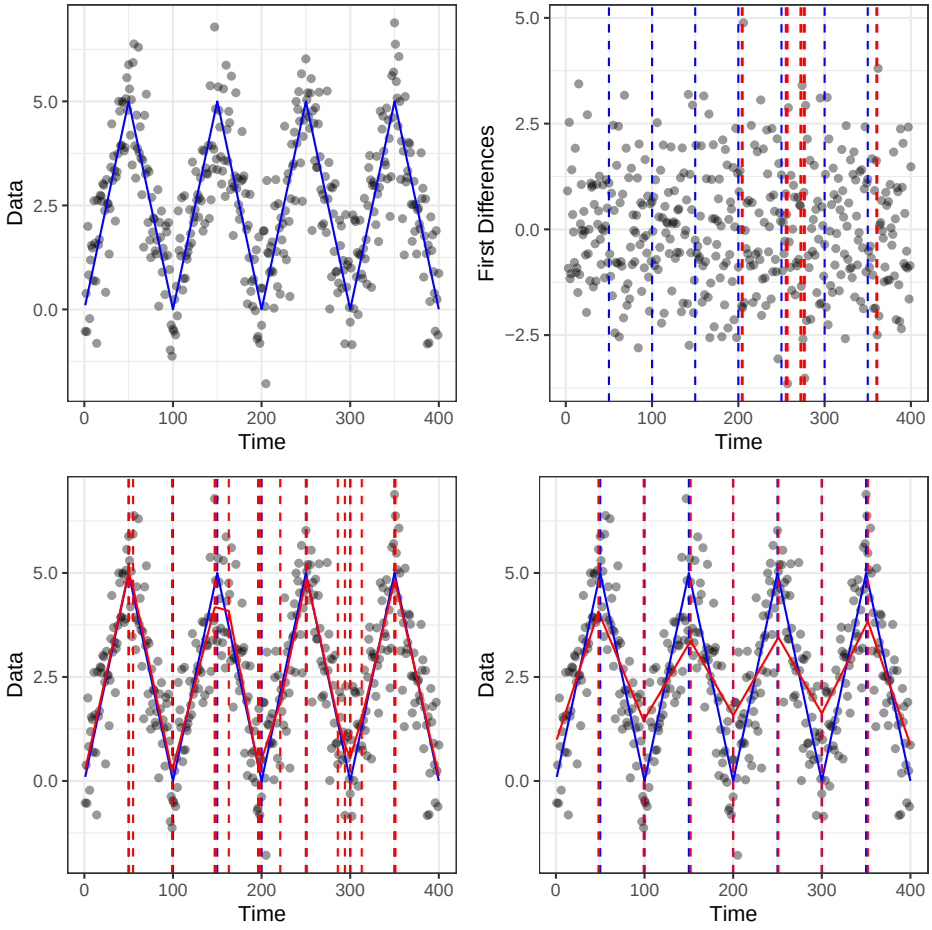

Figure 1: Example data simulated from a change-in-slope model (top left), and results from applying a change-in-mean algorithm to the first differences (top right) or from using trend-filtering (bottom row). In each case the true mean function (solid line) and change locations (vertical dashed lines) are shown in blue, and the estimates in red. For trend filtering we chose the L_1 penalty value based on cross-validation (bottom left) or so that it obtained the correct number of changes (bottom right). In the former case we over-estimate the number of changes, while in the latter we obtain a poor estimate of the mean.

(such as monotonically increasing) and for different loss functions for measuring fit to data. The **breakfast** package implements a range of methods based on recursively applying a test to detect a single change, for example wild binary segmentation (**?**) and IsolateDetect (**?**); whilst **mosum** (**?**) implements the MOSUM procedure. Packages **stepR** and **FDRseg** implement the multiscale approaches of **?**, **?** and **?**.

Separately there are packages that perform non-parametric change detection, for example **ecp** (**?**) implements the method of **?**, while **changepoint.np** implements the method of **?**. There are also methods for analysing multiple dimensional data streams, such as **InspectChangepoint** (**?**), and **changepoint.geo** (**?**); Bayesian methods, such as **bcp** (**?**); and methods that implement online procedures such as **CPM** (**?**) and **FoCUS** (**?**).

However, as mentioned above, there are more limited methods for specifically detecting changes-in-slope. The trend filtering algorithm can be implemented using the `trendfilter` function from the **genlasso** (**?**) package, and the NOT algorithm can be implemented using the **not** package or is available within **breakfast**. However current implementations of these do not allow for unevenly spaced, heterogeneous observations or minimum segment lengths. (Though there is flexibility within the **genlasso** package for implementing general lasso algorithms, and these can be constructed to fit a trend-filtering model to unevenly spaced data).

2. Detecting changes in slope

Assume we have data points $(y_1, x_1), \ldots, (y_n, x_n)$, ordered so that $x_1 < x_2 < \cdots < x_n$. In many applications x_i will be a time-stamp of when response y_i is obtained, whilst in, say, genomic applications, x_i may correspond to a location along the genome at which observation y_i is taken. We wish to model the response, y, as a signal plus noise where the signal is modelled as a continuous piecewise linear function of x. That is

$$y_i = f(x_i) + \epsilon_i, \tag{1}$$

where $f(x)$ is a continuous piecewise linear function, and ϵ_i is noise. If the function $f(x)$ has K changes in slope in the open interval (x_1, x_n), and these occur at x-values $\tau_1, \ldots, \tau_K$, and we define τ_0 and τ_{K+1} to be arbitrary values such that $\tau_0 \leq x_1$ and $\tau_{K+1} \geq n$ then we can uniquely define $f(x)$ on $[x_1, x_n]$ by specifying the values $f(\tau_i)$ for $i = 0, \ldots, K+1$. The function $f(x)$ can then be obtained via straight-line interpolation between the points $(\tau_i, f(\tau_i))$.

Our interest is in estimating the number of changes in slope, K, their locations, $\tau_1, \ldots, \tau_K$, and the underlying signal. The latter is equivalent to estimating $f(\tau_i)$ for $i = 0, \ldots, K+1$. To simplify notation we will denote these values by $\alpha_0, \ldots, \alpha_{K+1}$, i.e. $\alpha_i = f(\tau_i)$ for $i = 0, \ldots, K+1$. Also, for this and other quantities we will use the shorthand $\alpha_{i:j}$ for integers $i \leq j$ to be the ordered set of values, $\alpha_i, \ldots, \alpha_j$.

2.1. An L_0-Penalised Criteria

To estimate the number and locations of the changes-in-slope, and the underlying signal, we will first introduce a grid of x-values, $g_{1:N}$ with these ordered so that $g_i < g_j$ if and only if $i < j$. Our estimate for $f(x)$ will be restricted to piecewise-linear functions whose slope is only allowed to change at these grid-points. We will define our estimator of $f(x)$ as the function

that minimises a penalised cost that is a sum of the fit of the function to data, measured in terms of a weighted residual sum of squares, plus a penalty for each change-in-slope. That is we solve the following minimisation problem

$$\min_{K,\tau_{1:K}\in g_{1:N},\alpha_{0:K+1}} \left\{ \sum_{i=1}^{n} \frac{1}{\sigma_i^2} \left(y_i - \alpha_{j(i)} - (\alpha_{j(i)+1} - \alpha_{j(i)}) \frac{x_i - \tau_{j(i)}}{\tau_{j(i)+1} - \tau_{j(i)}} \right)^2 + K\beta \right\}, \qquad (2)$$

where $\beta > 0$ is a user chosen penalty for adding a changepoint, $j(i)$ is such that $\tau_{j(i)} \leq x_i < \tau_{j(i)+1}$, and $\sigma^2_{1:n}$ are user specified constants that are estimates of the variances of the noise $\epsilon_{1:n}$. The cost that we are minimising consists of two terms. The first is the measure of fit to the data, and is a residual sum of squares, but with the residuals weighted by the inverse of the variance of the noise for that observation. The expression in this term that depends on $\alpha_{0:K+1}$ and $\tau_{0:K+1}$ is just an expression for $f(x_i)$ given that $f(x)$ is defined as the linear interpolation between the points (τ_i, α_i) for $i = 0, \ldots, K+1$. The second term is the penalty for the number of changes-in-slope, with a penalty of β for each change.

This approach for estimating changes-in-slope was first proposed in **?**, but they assumed that the locations of the data points were evenly spaced, so $x_i = i$, the grid-points were equal to the locations of the data points, so $N = n$ and $g_{1:N} = x_{1:n}$, and that the noise was homogeneous so $\sigma_i^2 = \sigma^2$, for some constant σ^2, for all i.

Before we describe how to extend the approach in **?** to this more general estimator, we first give some comments on this and related approaches to estimating changes-in-slope. This approach is a common for estimating changes (**??**) and the cost is often termed an L_0 penalised cost. This is to contrast it with L_1 penalised costs, such as implemented in trend-filtering (**??**) which are of similar form, except that the cost for adding a change-in-slope is linear in the size of the change-in-slope. The advantage of using an L_1 penalised cost is that solving the resulting minimisation problem is simpler – however such an approach tends to over-estimate the number of changes, as shown in the introduction. An alternative approach to estimating changes-in-slope is to perform tests for a change-in-slope on data from randomly chosen intervals of x-values and to combine the results of these tests using the narrowest-over-threshold procedure of **?**. The current formulation and implementation of these alternative methods also make the simplifying assumptions of evenly spaced locations for the data, change-in-slope only at data point locations, and that the noise variance is constant.

The choice of β in (**??**) is important for accurate estimates, with lower values of β leading to larger estimates of K, the number of changes. If the noise is approximately Gaussian and independent, and the estimate of the noise variance is good, then $\beta = 2\log n$ is an appropriate choice (**?**). In general these assumptions will not hold, and often larger values of β are required to e.g. compensate for positively auto-correlated noise. Where possible we recommend evaluating sets of estimated changepoints obtained for a range of β values, and these can obtained in a computationally efficient manner using the CROPS algorithm of **?**.

2.2. Dynamic Programming Recursion

Solving (**??**) is non-trivial as it involves minimising a non-convex function. Furthermore, standard dynamic programming algorithms for change points (**?**), e.g. those that recurse based on conditioning on the location of the most recent changepoint, cannot be used because of the dependence across changepoints due to the continuity constraint. Thus we follow **?**

and develop a dynamic programming recursion that conditions both on the location of the changepoints and the value of the mean at that changepoint.

Remember that we are allowing changepoints only at grid-points, $g_{1:N}$. We introduce a set of functions, each associated with a grid-point, $F_t(\alpha)$ for $t = 1, \ldots, N$, defined as the minimum value of the penalised cost for data up to and including grid-point g_t conditional on the signal at g_t being α, i.e. $f(g_t) = \alpha$. To define this formally, define a set of segment cost functions

$$\mathcal{C}_{s,t}(\alpha', \alpha) = \sum_{i=n_s+1}^{n_t} \left(y_i - \alpha' - (\alpha - \alpha')\frac{x_i - g_s}{g_t - g_s} \right)^2,$$

which is the cost of fitting a linear signal to data points between the sth and tth grid-points, i.e. (x_i, y_i) with $g_s < x_i \leq g_t$, with the signal taking the value α' at g_s and α at g_t. In the summation we use the notation n_s to denote the index of the last data-point located at or before g_s. If $n_t = n_s$, that is there are no data-points between g_s and g_t then this cost is set to 0.

Using this definition, the L_0 penalised criteria (**??**) can be written as

$$\min_{K, i_{1:K} \in 1:N, \alpha_{0:K+1}} \left\{ \sum_{k=0}^{K} \mathcal{C}_{i_k, i_{k+1}}(\alpha_k, \alpha_{k+1}) + K\beta \right\}. \tag{3}$$

Using this definition can define the function $F_l(\alpha)$ for $l = 1, \ldots, N$ as

$$F_l(\alpha) = \min_{K, i_{1:K} \in 1:l-1, \alpha_{0:K}} \left\{ \sum_{k=0}^{K-1} \mathcal{C}_{i_k, i_{k+1}}(\alpha_k, \alpha_{k+1}) + K\beta + \mathcal{C}_{i_K, l}(\alpha_K, \alpha) \right\},$$

which is of the form of (**??**) but with $\tau_{K+1} = g_l$ and $\alpha_{K+1} = \alpha$, as for $F_l(\alpha)$ we are analysing only data up to g_l and we are fixing $\alpha_{K+1} = f(g_l) = \alpha$.

Using the same argument as in **?** we can then derive a recursion for $F_l(\alpha)$. For $l = 1, \ldots, N$,

$$F_l(\alpha) = \min_{k \in 0:(l-1)} \left\{ \min_{\alpha'} \left[F_k(\alpha') + \mathcal{C}_{k,l}(\alpha', \alpha) + \beta \right] \right\},$$

with $F_0(\alpha) = -\beta$. The idea of this recursion is that we condition on the location of the most recent changepoint, g_k, and the value of the signal at that changepoint, α'. Conditional on this information the optimal segmentation prior to the most recent changepoint is independent of the data since that changepoint, and the minimum of the penalised cost is $F_k(\alpha') + \mathcal{C}_{l-1,l}(\alpha', \alpha) + \beta$: the sum of the minimum cost for the data prior to g_k, plus the segment cost for the data from g_k to g_l plus the penalty for adding a changepoint. Finally we obtained $F_l(\alpha)$ by minimising over k and α'.

Solving this recursion is possible as the functions $F_l(\alpha)$ can be summarised as the pointwise minimum of a set of quadratics. For each k, we can solve the inner minimisation over α' analytically. Doing so for each k will define $F_l(\alpha)$ as the pointwise minimum of a large set of quadratics, and we can use a line search to then prune quadratics that do not contribute to this minimum (which is important to reduce the computational cost of the algorithm). See **?** for full details. As noted there, it is possible to further reduce the computational cost of solving the recursion by using PELT pruning ideas from **?**. This pruning enables us to reduce the search space of k within the recursion. Finally, whilst we have described how to find the

minimum value of the penalised cost, our main interest is in the locations of the changepoints and the value of the signal that gives that minimum cost. However extracting this information is trivial once we have solved the recursions – again see **?** for details.

The novelty relative to **?** is that for this recursion we have decoupled the grid of potential changepoints from the locations of the datapoints. Furthermore, our setting allows for unevenly spaced data and for the noise variance to be heterogeneous. These impact that definition of $\mathcal{C}_{l-1,l}(\alpha', \alpha)$ and how we perform the inner minimisation over α'. Full details are given in Appendix **??**.

One further extension of this approach to detecting changes is to allow for a minimum segment length. This can be done by optimising the penalised cost (**??**) only over sets of changepoints that are consistent with the prescribed minimum segment length. To minimise the cost subject to such a constraint involves adapting the dynamic programming recursion so that we only search over values of k for the location of the most recent changepoint that are consistent with the minimum segment length. However one drawback with imposing a minimum segment length for the change-in-slope problem is that it makes the PELT pruning ideas invalid. In our implementation of `cpop`, if a minimum segment length is specified we allow the algorithm to be run both with and without the PELT pruning. If run without PELT pruning, the algorithm will be slower but guaranteed to find the optimal segmentation under our condition. Running with PELT pruning is quicker but the algorithm may output a slightly sub-optimal segmentation. In practice we have observed that this happens rarely.

3. The cpop Package

The **cpop** package has functions to simulate data from a change-in-slope model, implement the CPOP algorithm to estimate the location of the changes, and various functions for summarising and plotting the estimates of the change locations and the mean function.

3.1. Generating Simulated Data

The `simchangeslope` function allows for simulating data from a change-in-slope model (**??**):

```
simchangeslope(x, changepoints, change.slope, sd = 1)
```

It takes the following arguments :

- `x` - A numeric vector containing the locations of the data.
- `changepoints` - A numeric vector of changepoint locations.
- `change.slope` - A numeric vector indicating the change in slope at each changepoint. The initial slope is assumed to be 0.
- `sd` - The residual standard deviation. Can be a single numerical value or a vector of values for the case of varying residual standard deviation. Default value is 1.

It returns a vector y of simulated values which correspond to the locations x. The mean function of the data goes through the origin – but to add an intercept we just add a constant

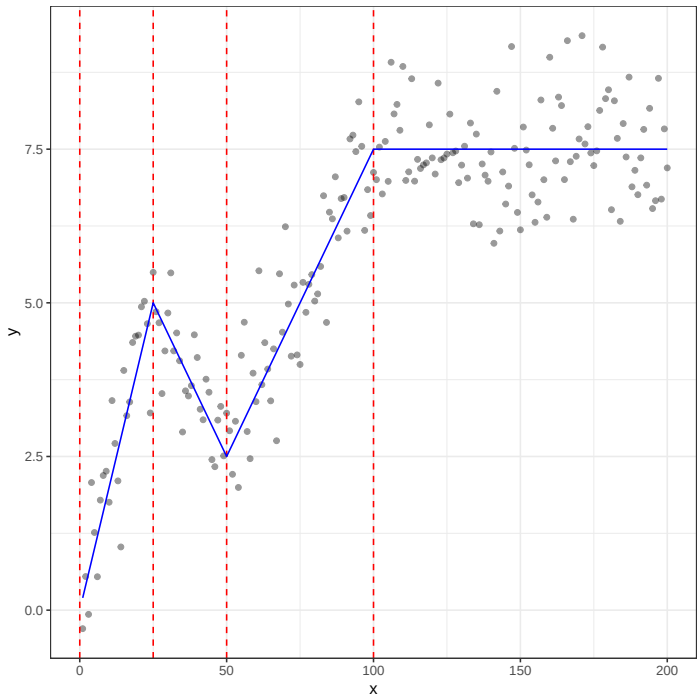

Figure 2: Simulated data (black dots) with true mean (blue dashed line) and changepoints (vertical red dashed lines).

to all output values. It is possible to get the value of the mean function at the x-values of the data by setting `sd=0`.

The following code demonstrates the `simchangeslope` function and displays the data along with the (true) line segments and the locations of the changes in slope (see Figure **??**).

```
R> library("cpop")
R> library("ggplot2")
R> changepoints <- c(0, 25, 50, 100)
R> change.slope <- c(0.2, -0.3, 0.2, -0.1)
R> x <- 1:200
R> sd <- 0.8
R> y <- simchangeslope(x, changepoints, change.slope, sd)
R> df <- data.frame("x" = x, "y" = y)
R> p <- ggplot(data = df, aes(x = x, y = y))
R> p <- p + geom_point(alpha = 0.4)
R> p <- p + geom_vline(xintercept = changepoints,
+                       color = "red",
+                       linetype = "dashed")
R> mu <- simchangeslope(x, changepoints, change.slope, sd = 0)
R> p <- p + geom_line(aes(y = mu), color = "blue")
R> p <- p + theme_bw()
R> print(p)
```

3.2. Determining Changes in Slope

The function `cpop` is used to determine the locations of changes in slope.

```
cpop(y, x = 1:length(y) - 1, grid = x, beta = 2 * log(length(y)),
    sd = 1, minseglen = 0, prune.approx = FALSE)
```

It takes the following arguments :

- `y` - A vector of length n containing the data.
- `x` - A vector of length n containing the times/locations of data points. Default value is NULL, in which case the locations are set to be $0, 1, \ldots, n-1$, corresponding to evenly spaced data.
- `grid` - An ordered vector of possible locations for the change points. If this is NULL, then this is set to x, the vector of times/locations of the data points.
- `beta` - A positive real value for the penalty, β in (**??**), incurred for adding a changepoint. The larger the penalty, the fewer changepoints will be detected.
- `sd` - Estimate of residual standard deviation. Can be a single numerical value if it is the same for all data points, or a vector of n values for the case of varying standard deviation.
- `minseglen` - The minimum allowable segment length, i.e. distance between successive changepoints. The default is that no minimum segment length is imposed.
- `prune.approx` - Only relevant if a minimum segment length is set. If True, `cpop` will use an approximate pruning algorithm that will speed up computation but may occasionally lead to a sub-optimal solution in terms of the estimated changepoint locations. If the minimum segment length is 0, then an exact pruning algorithm is possible and is used.

The `cpop` function returns an S4 object for which a number of generic methods, including `plot` and `summary`, are provided. The following demonstrates how the `cpop` function can be used to determine changes in slope, by analysing the data we simulated and plotted in Figure **??**. It uses the default penalty, $\beta = 2\log n$, and we assume that the true noise standard deviation, 0.8, is known. The `summary` function is used to provide an overview of the analysis parameters along with estimated changepoint locations, corresponding fitted line segments, and the (weighted) residual sum of squares (RSS) for each segment.

```
R> res <- cpop(y, x, sd = 0.8)
R> summary(res)

cpop analysis with n = 200 and penalty (beta)  = 10.59663

3  changepoints detected at x =
 22 52 95
fitted values :
  x0       y0  x1       y1     gradient   intercept      RSS
```

```
1  1 0.147335  22 4.844725  0.223685242 -0.07635023 10.07761
2 22 4.844725  52 2.717661 -0.070902123  6.40457180 10.38813
3 52 2.717661  95 7.303644  0.106650750 -2.82817758 25.09463
4 95 7.303644 200 7.563413  0.002473995  7.06861408 61.78303

overall RSS = 107.3434
cost = 199.514
```

The predicted change in slope (changepoint) locations and corresponding line segments can be displayed using `plot`. The `plot` function returns a **ggplot2** object which can be augmented to include additional features such as the true change in slope locations and line segments (see Figure **??**).

```
R> p <- plot(res)
R> p <- p + geom_vline(xintercept = changepoints[-1], color = "blue",
+                      linetype = "dashed")
R> p <- p + geom_line(aes(y = mu), color = "blue", linetype = "dashed")
R> print(p)
```

The last two lines of code add the true changepoint locations and the true mean function to the plot. For plotting the changepoint location we omit the first element of `changepoints`, which was 0, as that was included just to set the initial slope of the mean and does not correspond to a change-in-slope.

3.3. Other Functions

In addition to `plot` and `summary`, the **cpop** package provides functions to evaluate the fitted mean function at specified x-values, and to calculate the residuals of the fitted mean. The primary argument of these functions is `object` - An instance of a cpop S4 class as produced by the function `cpop`.

The function `changepoints(object)` creates a data frame containing the locations of the changepoints in terms of the their x-values.

```
R> changepoints(res)

  location
1       22
2       52
3       95
```

The function `estimate(object, x = object@x, ...)` with argument, `x`, that specifies the x-values at which the fit is to be estimated, creates a data frame with two columns containing the locations `x` and the corresponding estimates $\hat{y}$. The default value for `x` is the vector of x locations at which the cpop object was defined.

```
R> estimate(res, x = c(0.1,2.7,51.6))
```

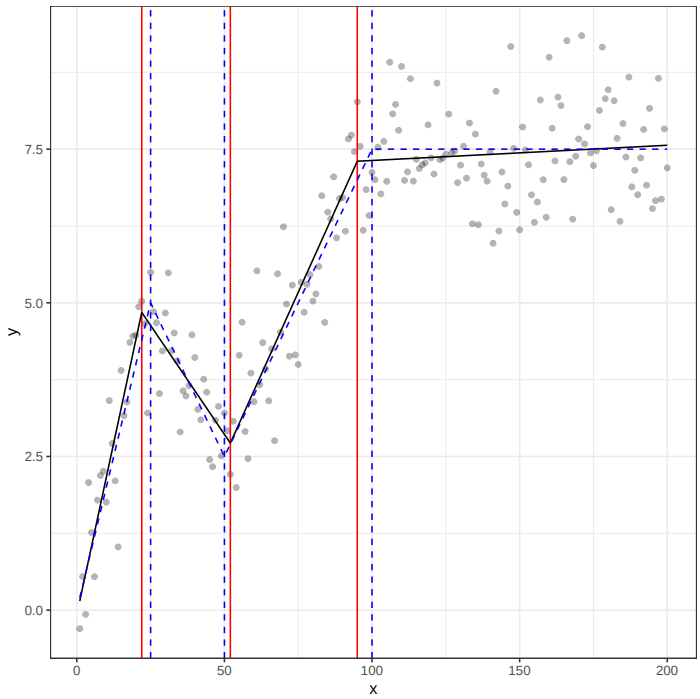

Figure 3: Example output of `cpop` for simulated data from Figure **??**. The true mean and changepoints are given in blue dashed lines, together with estimated mean (black full line) and changepoints (red full lines).

```
   x     y_hat
1  0.1 0.1473350
2  2.7 0.7289166
3 51.6 2.9473390
```

The function `fitted(object)` creates a data frame containing the endpoint coordinates for each line segment fitted between the detected changepoints. The data frame also contains the gradient and intercept values for each segment and the corresponding residual sum of squares (RSS).

```
R> fitted(res)
```

```
  x0       y0  x1       y1     gradient   intercept      RSS
1  1 0.147335  22 4.844725  0.223685242 -0.07635023 10.07761
2 22 4.844725  52 2.717661 -0.070902123  6.40457180 10.38813
3 52 2.717661  95 7.303644  0.106650750 -2.82817758 25.09463
4 95 7.303644 200 7.563413  0.002473995  7.06861408 61.78303
```

Finally, the function `residuals(object)` creates a single column matrix containing the residuals.

```
R> head(residuals(res))
```

```
           [,1]
[1,] -0.4484981
[2,]  0.1758944
[3,] -0.6632084
[4,]  1.2578339
[5,]  0.2215302
[6,] -0.7221359
```

4. Extensions of cpop

4.1. Irregularly Sampled Data

The **cpop** package allows for irregularly spaced data, both when simulating data and when running the CPOP algorithm. The only change to the previous code that we need to make is to change the definition of `x` that is input to `simchangeslope`.

```
R> x <- (1:200)^(2)/(200)
R> changepoints <- c(0, 25, 50, 100)
R> change.slope <- c(0.2, -0.3, 0.2, -0.1)
R> sd <- 0.8
R> y <- simchangeslope(x, changepoints, change.slope, sd)
```

To analyse the data we use `cpop` as before. (The only difference is that for evenly spaced data one can omit the `x` argument – but it must be included for unevenly spaced data.)

```
R> res <- cpop(y, x, sd = 0.8)
```

Figure **??** shows a plot of the simulated data and estimated changepoints and mean function.

4.2. Heterogeneous Data

To simulate heterogeneous data we just input a vector of the standard deviation of each data point. For example, we can produce a version of the simulation from Section **??** but with the noise standard deviation increasing with x.

```
R> x <- 1:200
R> sd <- x / 100
R> y <- simchangeslope(x, changepoints, change.slope, sd))
```

Here the values of `changepoints` and `change.slope` are as before.

It is interesting to compare two estimates of the changepoints, one where we assume a fixed noise standard deviation, and one where we assume the true noise standard deviation. For the former it is natural to set this value so the the average variance of the noise is correct.

```
R> res <- cpop(y, x, sd = sqrt(mean(sd^2)))
R> res.true <- cpop(y, x, sd = sd)
```

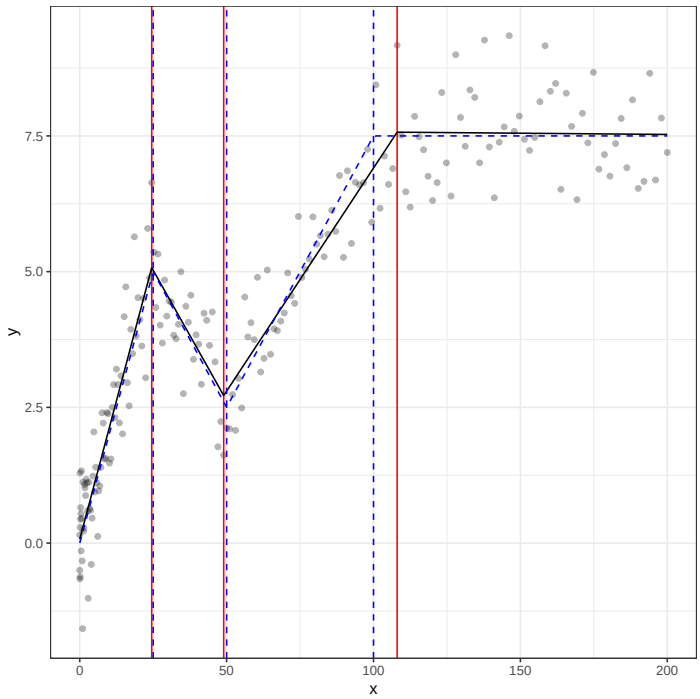

Figure 4: Example output of `cpop` for unevenly spaced simulated data. The true mean and changepoints are given in blue dashed lines, together with estimated mean (black full line) and changepoints (red full lines).

Here `res` contains the results where we assume a fixed noise standard deviation, and `res.true` where we use the true values. Figure **??** shows the results – and we can see that wrongly assuming homogeneous noise leads to detecting two false positive changepoints in regions where the noise variance is above what was assumed.

One practical issue is how can we estimate the noise variance in the heterogeneous case? In some situations there may be covariate information that tells the relative variance of the noise for different data points (for example due to some data being averages of multiple measurements). Alternatively if we know how the noise variance depends on x we can estimate this by (i) running CPOP assuming a constant variance; (ii) calculating the residuals of the fitted model; (iii) estimating how the noise variance varies with x by fitting an appropriate model to the residuals. An example of this scheme will be seen in Section **??**.

4.3. Choice of Grid

The computational cost for `cpop` increases with the size of the number of potential changepoint locations. To see the rate of increase we ran `cpop` with the default settings where the grid of potential changepoints is equal to the x values of the data, for data sizes varying from $n = 200$ to $n = 6400$. We considered two scenarios, one where we had a fixed number of changepoints and one where we had a fixed segment size of length 100. The average CPU cost across 10 runs of `cpop` for each data size are shown in Figure **??**. The suggest that the computational cost is increasing like $n^{2.5}$ when we have a fixed number of changes, and like $n^{1.7}$ when the number of changes increases linearly with n. By comparison, if we analyse each data set with

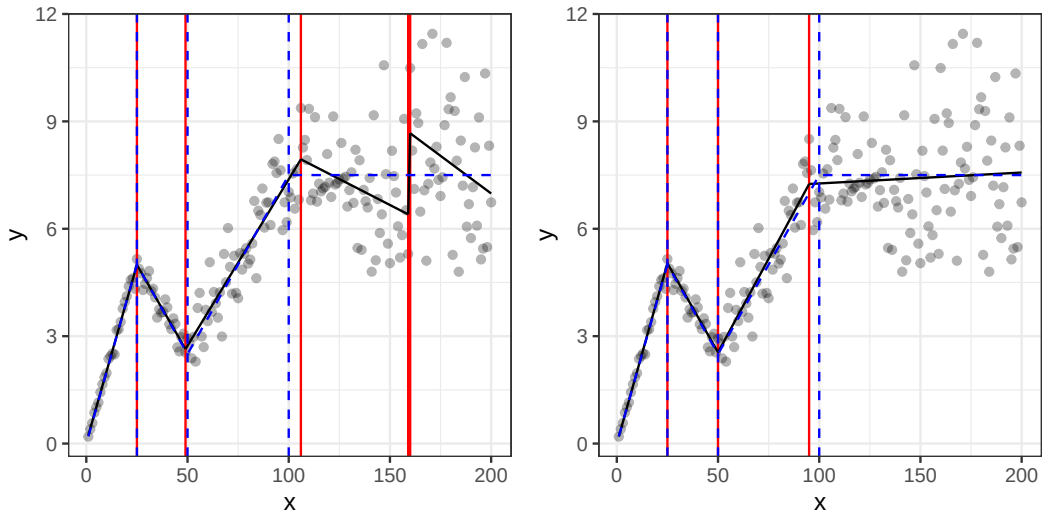

Figure 5: Example output of `cpop` for heterogenous noise: assuming a constant noise variance (left) and the true noise variance (right). The true mean and changepoints are given in blue dashed lines, together with estimated mean (black full line) and changepoints (red full lines).

a grid of 200 evenly spaced potential locations for the changes, the computational cost is roughly constant.

Thus for large data sets, we can substantially reduce the computational cost of running `cpop` by using a smaller grid of potential change locations. Obviously this comes with the drawback of a potential loss of accuracy with regards to the estimated changepoint locations. However one possible approach is to run `cpop` with a coarse grid, and then re-run the algorithm with a finer grid around the estimated changepoints.

To see this we implemented the scheme for a data set with $n = 6400$ and a fixed segment size of 200. We initially ran `cpop` with a grid with potential changes allowed every 16 observations.

```
R> x <- 1:6400
R> y <- simchangeslope(x, changepoints = 0:31*200,
+ change.slope = c(0.05,0.1*(-1)^(1:(31))), sigma = 1)
```

We use a smaller value for the penalty due to the smaller grid size, and the fact that this is a preliminary step to find roughly where the changes are: so the key is to avoid missing changes. Spurious changes can still be removed when we perform our final run of `cpop`.

```
R> res.coarse <- cpop(y, x, grid=1:399*16, beta = 2 * log(400))
```

In our example we find 38 changepoints with this coarse grid. We then introduce a finer grid around these putative changes: our new grid includes all x-values within 8 of each putative changepoint.

```
R> cps <- unlist(changepoints(res.coarse))
```

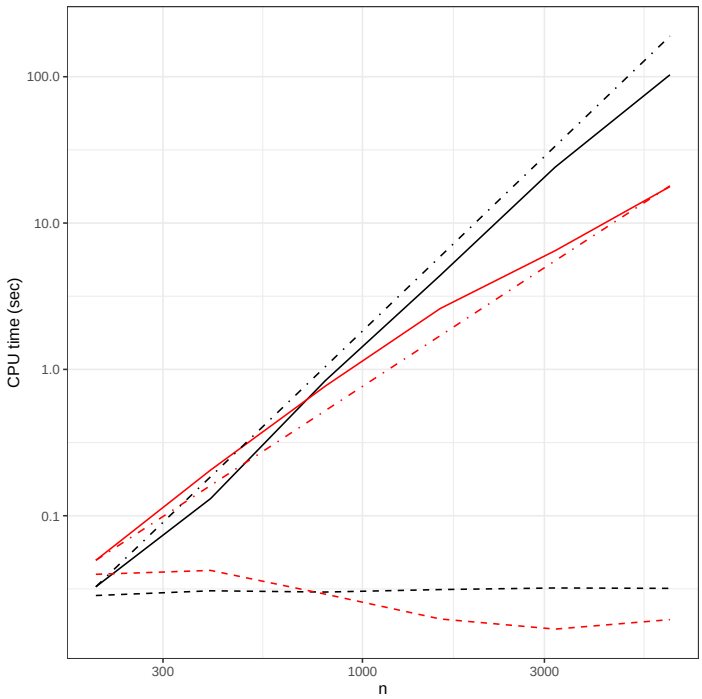

Figure 6: Empirical computational cost for `cpop` as a function of sample size, n, for a grid of size n (full lines) and of size 200 (dashed lines), and for data with a single changepoint (black) and for a linearly increasing number of changepoints (red). To aid interpretation straight lines for CPU cost proportional to $n^{1.7}$ (red dot-dashed) and $n^{2.5}$ (black dot-dashed) are shown.

```
R> grid <- NULL
R> for(i in 1:length(cps))
R> {
R>   grid <- c(grid, cps[i] + (-7):8)
R> }

R> res.fine <- cpop(y, x, grid, beta = 2 * log(length(x)))
```

This gives a computational saving of between 10 to 100 over the default running of `cpop`. We can evaluate the accuracy of the approach by then comparing the estimated changepoints to the estimates we obtain if we run the default setting of `cpop`. In this case, both runs estimate the same number of changepoints, with the maximum difference in the location of a change being two time-points. The slower, default running of `cpop` gives a segmentation with a marginally lower cost (of 7187.1 as opposed to 7188.0).

4.4. Imposing a Minimum Segment Length

The `cpop` function allows the user to specify a minimum segment length – and this is defined as the minimum x-distance allowed between two estimated changepoints. Specifying a minimum segment length can make the method more robust to point outliers or noise that is heavier-tailed than Gaussian: as minimising (**??**) can lead to over-fitting in such scenarios and this over-fitting tends to be through adding clusters of changepoints close together to fit the noise in the data. There are two disadvantages of imposing a minimum segment length. First it can cause true changes to be missed if they are closer together than the specified minimum segment length. Second `cpop` is slower when a minimum segment length is imposed.

To see these issues, we simulated data as in Section **??**, except that we assumed the noise was t_4 distributed. We cannot simulate such data directly with `simchangeslope`, so we need to first use `simchangeslope` to calculate the mean function, and then add the noise:

```
R> changepoints <- c(0, 25, 50, 100)
R> change.slope <- c(0.2, -0.3, 0.2, -0.1)
R> x <- 1:200
R> mu <- simchangeslope(x, changepoints, change.slope, sigma = 0)

R> y <- mu + rt(length(x), df = 4)
```

We then estimated the changepoint locations both without a minimum segment length, and with minimum segment lengths of 10, 30 and 40. To run `cpop` with a minimum segment length of 10:

```
R> res.min <- cpop( y, x, beta = 2*( log( length(y) ) ),
  +. minseglen = 10, sd = sqrt(2) )
```

The argument `sd = sqrt(2)` is because t_4 distributed noise has a variance of 2. Results from CPOP with different minimum segment lengths are shown in Figure **??**. If we do not impose a minimum segment length, then we estimate 11 changepoints, including three cluster of changes that overfit to the noise. By imposing a minimum segment length of 10 we avoid

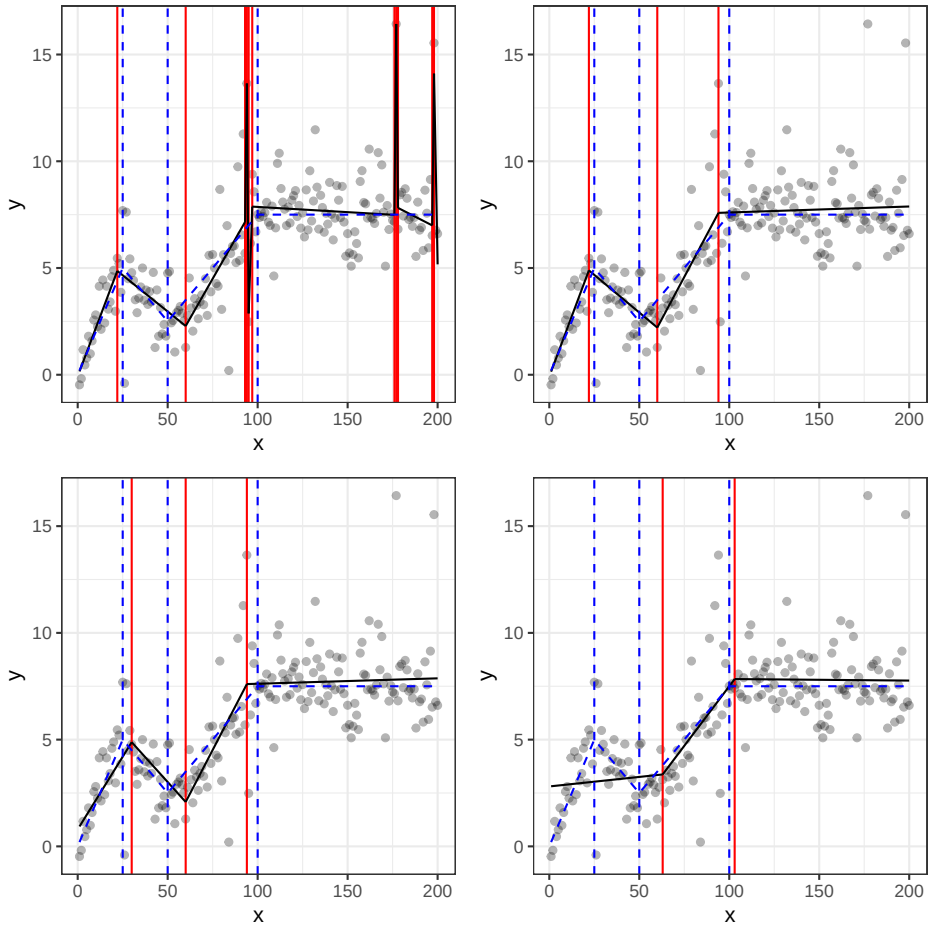

Figure 7: Results of analysing data with t_4 noise with no minimum segment length (top left) and minimum segment lengths of 10 (top right), 30 (bottom left) and 40 (bottom right). In each plot we show data (grey dots), true mean (blue dashed line), true changepoints (blue vertical dashed lines), estimated mean (black line) and estimated changepoints (red vertical liens)

the over-fitting. For this example, the computational cost of running `cpop` with the minimum segment length is about 15 times larger than when we do not assume a minimum segment length.

Assuming a minimum segment length of 30 or 40 shows what can happen when our minimum segment length assumption does not hold. A minimum segment length of 30 leads to estimates of the first two changes at time-points 30 and 60 – the closest possible given the assumption. As we increase the minimum segment length to 40 we miss the first changepoint all together.

4.5. Choice of Penalty

The choice of the penalty, `beta`, in `cpop` can have a substantial impact on the number of changepoints detected and the accuracy of the estimated mean function. This is common to all changepoint methods, where there will be at least one tuning parameter that specifies the evidence for a change that is needed before a change is added. The default choice of penalty, $2\log n$ where n is the data size, is based on assumptions that the noise is IID Gaussian with known variance. When these assumptions do not hold, it is recommended to look at the segmentations obtained as the penalty value is varied: this can be done efficiently using the CROPS algorithm of **?**.

The idea of CROPS is that it allows a penalised cost method to be implemented for all penalty values in an interval. This is implemented within the **cpop** package by the function:

```
cpop.crops(y, x = 1:length(y), grid = x, beta_min = 1.5 * log(length(y)),
  beta_max = 2.5 * log(length(y)), sd = 1, minseglen = 0, prune.approx = FALSE)
```

The arguments of `cpop.crops` are identical to those of `cpop` except that, rather than specifying a single penalty value (`beta`), the range of penalty values to be used is specified by `beta_min` and `beta_max`, which fix the smallest and largest penalty value to be used. The output is an instance of an S4 class that contains details of all segmentations found by minimising the penalised cost for some penalty value in the interval between `beta_min` and `beta_max`.

To see the use of `cpop.crops`, consider an application where we do not know the standard deviation of the noise. Under our criteria (**??**), the optimal segmentation with penalty $c^2\beta$ and standard deviation σ_i/c will be the same if we fix β and $\sigma_{1:n}$ but vary $c > 0$. Thus under an assumption that the noise is homogeneous, we can run `cpop` with `sd = 1` but for a range of $\beta \in [2\sigma_-^2 \log n, 2\sigma_+^2 \log n]$, and this will give us the optimal segmentations for $\beta = 2\log n$ as we vary noise standard deviation between σ_- and σ_+.

We simulated data as per Section **??** but with `sd = 1.5`. To run CPOP for a range of $\beta \in [7.5, 50]$ we use

```
R> res.crops <- cpop.crops(y, x, beta_min = 5, beta_max = 50)
```

For our example $2\log n = 10.6$, so this is equivalent to trying noise standard deviation in the range $[0.8, 2.2]$.

We can plot the location in the changepoints for each segmentation found by CPOP for $\beta \in [7.5, 50]$

```
R> plot(res.crops)
```

This is shown in Figure **??**, and shows that there are 6 different segmentations found. These are labelled with a penalty value which gives that segmentation (left axis) and the unpenalised cost, i.e. the weighted RSS, for that segmentation and penalty value (right axis).

Details of the segmentations can be obtained using `segmentations(res.crops)` . This gives a matrix with one row for each of the segmentations, and each row contains a value of β that gives that segmentation, the corresponding unpenalised cost, the penalised cost, the number of changepoints, and then the list of ordered changepoints. We can also obtain a list with the output from `cpop` corresponding to each segmentation, with `models( res.crops )`. For example, one approach to choose a segmentation from the output from `cpop.crops` is to find the segmentation that minimises a cost under a model where we assume the noise variance is unknown (**?**),

$$n \log \left(\frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{f}(x_i) \right)^2 \right) + 2K \log n.$$

Here $\hat{f}$ is the estimated mean function and K is the number of changepoints. This can be calculated as follows.

```
R> models <- cpop.crops.models(res.crops)
R> M <- length(models)
R> BIC <- rep(NA, M)
R> ncps <- segmentations(res.crops )[,4]
R> n <- length(y)
R> for(j in 1:M)
R> {
R>   BIC[i] <- n * log(mean((residuals(models[[j]]))^2))
+    + 2 * ncps[i] * log(n)
R> }
```

This uses that the fourth column of the matrix `segmentations( res.crops )` stores the number of changepoints in each segmentation, and that we can calculate $y_i - \hat{f}(x_i)$ using the `residual` function evaluated for the corresponding entry of `cpop.crops.models(res.crops)`. The segmentation which has the smallest value of `BIC` is shown in Figure **??**, and shows that this correctly chooses the segmentation with three changes.

As a final example, we performed a similar analysis but with correlated noise. This violates the assumption of IID noise that underpins the default choice of penalty, thus we run `cpop.crops` for a range of penalties.

We simulated data with $n = 500$ data points and 10 equally spaced changepoints.

```
R> n <- 500
R> x <- 1:n
R> mu <- simchangeslope(x, changepoints = 45*0:10,
+  change.slope = c(0.15,0.3*(-1)^(1:10)), sd = 0)
R> epsilon <- rnorm(n+2)
R> y <- mu + (epsilon[1:n] + epsilon[2:(n+1)] + epsilon[3:(n+2)]) / sqrt(3)
```

The noise is MA(3), and we simulate the data by first calculating the mean function, `mu`, and then adding the MA(3) noise.

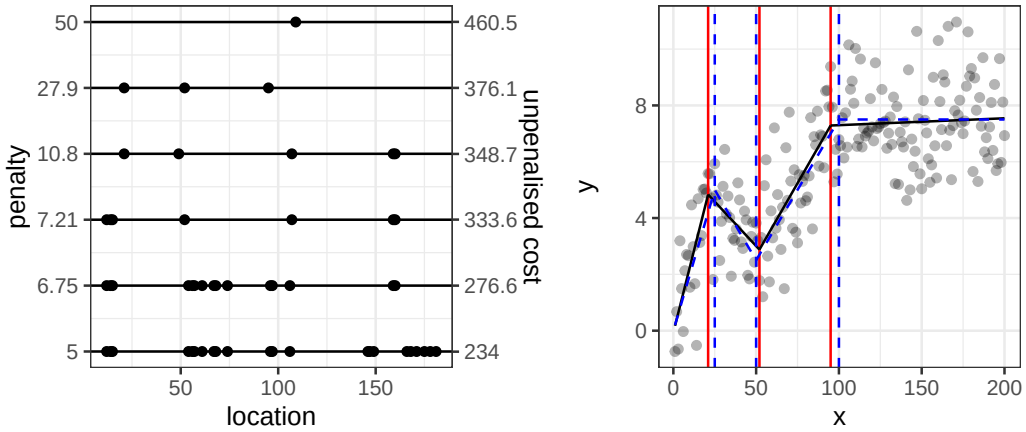

Figure 8: Example plot of output from `cpop.crops` (left), and best segmentation based on calculated `BIC` (right). For the left-hand plot each row shows a segmentation, with points at estimated changepoint location. The left-hand axis shows a penalty value that leads to that segmentation, and the right-hand axis gives the corresponding unpenalised cost. For the right-hand plot we show the true mean and changepoints (blue dashed lines), and estimated mean (black line) and changepoints (red lines).

We could continue as above, and choose between the segmentations by minimising a penalised cost under an appropriate model for the noise; this type of approach is suggested for change in mean models by **?**. A simpler, albeit more qualitative approach, is to plot the residual sum of squares of the segmentation against the number of changepoints (**????**). This avoids the need to specify a model for the residuals. The idea of this approach is that adding "true" changes should lead to a noticeably larger reduction in the residual sum of squares than adding "spurious" changes. Thus the best segmentation should correspond to an "elbow" in this plot.

```
R> res.crops <- cpop.crops(y, x, beta_min = 8, beta_max = 200)
R> segs <- segmentations(res.crops)
R> p <- ggplot(data = segs, aes(x = m))
R> p <- p + geom_line(aes(y = Qm))
R> p <- p + geom_vline(xintercept = 10, color = "red")
R> p <- p + xlab("No. of changepoints") + ylab("unpenalised cost")
R> plot(p)
```

This runs `cpop.crops` and then uses the fact that the output of `segmentations` includes columns that give the number of changepoints and the unpenalised cost of each segmentation. These columns are labelled `"m"` and `"Qm"` respectively. The plot gives a clear elbow, see Figure **??**, and this corresponds to a correct estimate of the number of changes.

5. Application

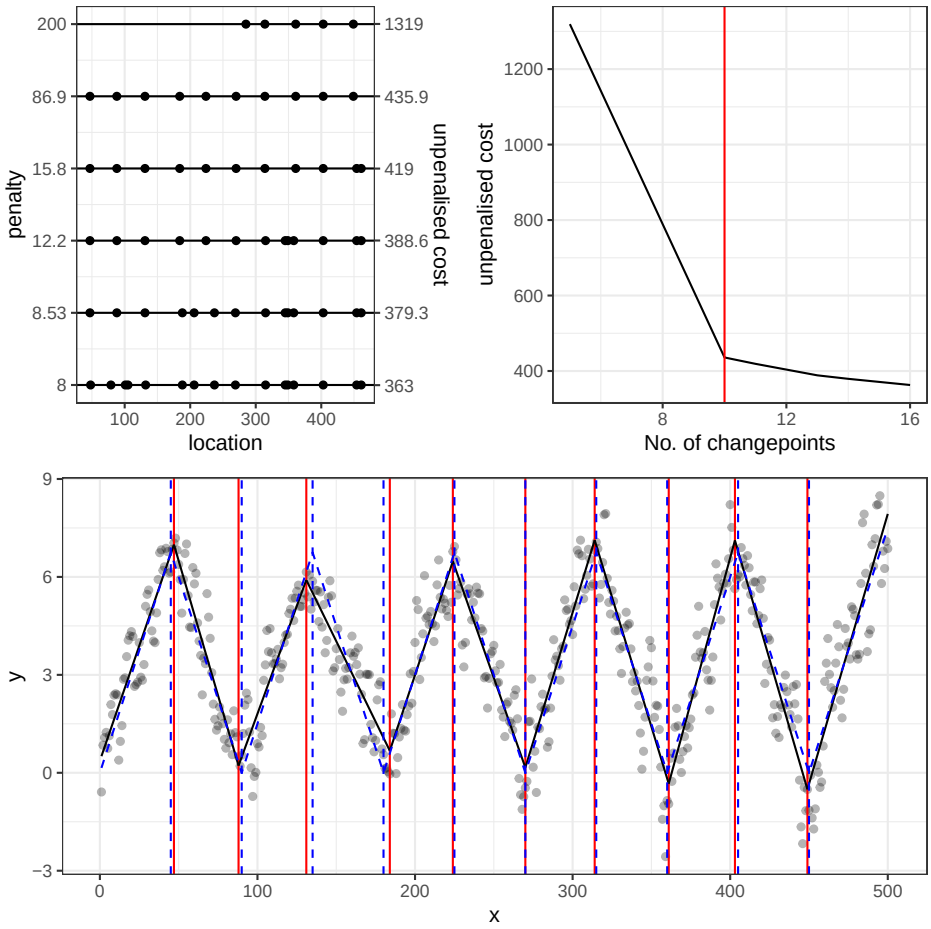

Figure 9: Correlated noise example. Output from `cpop.crops` (top left), unpenalised cost against number of changepoint (top right) and estimate from segmentation corresponding to "elbow" (bottom).

We now demonstrate an application of `cpop` on analysing power spectra of velocity as a function of wavenumber obtained from models of the Atlantic Ocean. The data is available in the **cpop** package and can be loaded with `data("wavenumber_spectra")`. It contains four spectra, corresponding to two different months (February and August) from two different runs of the model (2000 and 2100) corresponding to present and future scenarios: see Figure **??**. The data comes from **?**, and is available from **?**. See **?** for a fuller description of the data.

Interest lies in estimating the rate of decay of the log-spectra against log-wavenumber. We can do this by removing the first three data points (where the spectra is increasing) and then using `cpop` to fit a piecewise-linear curve to the remaining data. We perform an initial run of `cpop` assuming an estimated homogeneous noise variance on the data from August from the 2000 run.

```
R> data("wavenumber_spectra")
R> x <- log(wavenumber_spectra[-(1:3),1])
R> y <- log(wavenumber_spectra[-(1:3),4])

R> grid <- seq(from = min(x), to = max(x), length = 200)

R> sig2 <- mean( diff( diff(y) )^2 )/6

R> res <- cpop(y, x, grid, sd = sqrt(sig2), minseg = 0.2, beta = 2*log(200))
```

Here we estimate the noise variance, `sig2`, based on the variance of the double difference of the data. For regions where the mean is linear and the data is evenly spaced, taking the double difference will lead to a mean zero process. If the noise is IID with variance σ^2 then the double-differenced process will have a marginal variance that is $6\sigma^2$. Thus our estimate is the empirical mean of the square of the double-difference data divided by 6. The original data is evenly spaced in in terms of wavenumber, but as we take logs, `x` is unevenly spaced: so this estimator will be biased in our setting. However it will give a reasonable ball-park figure for an initial run of `cpop` – the residuals from which can then be used to get a better estimate of the noise variance. We use a evenly spaced grid for possible change-point locations. To avoid the potential for adding multiple changepoints between two observations, we set a minimum segment length of 0.09 (as the largest distance between consecutive `x` values is 0.08).

The output is shown in the top-right plot of Figure **??**, and appears to be over-fitting to the early part of the series. This is because the noise variance is heterogeneous, and decreasing with `x`. However, given our initial fit we can use the residuals to estimate the noise variance. The noise for the spectra is expected to be approximately inversely proportional to the wavenumber. By using a Taylor-expansion, we have the variance of the noise for the log-spectra should be approximately the variance of the noise of the spectra divided by the square of the mean of the spectra. As the mean of the spectra is roughly a power of the wavenumber, this suggests using a model for the variance, σ_x^2 say, depending on x as $\log \sigma_x^2 = a + bx$: so that the variance is proportional to some power of the wavenumber. We can estimate the parameters of this model by maximising the log-likelihood of Gaussian model for the residuals with this form for the variance.

```
R> r2 <- residuals(res)^2
R> loglik <- function(par)
```

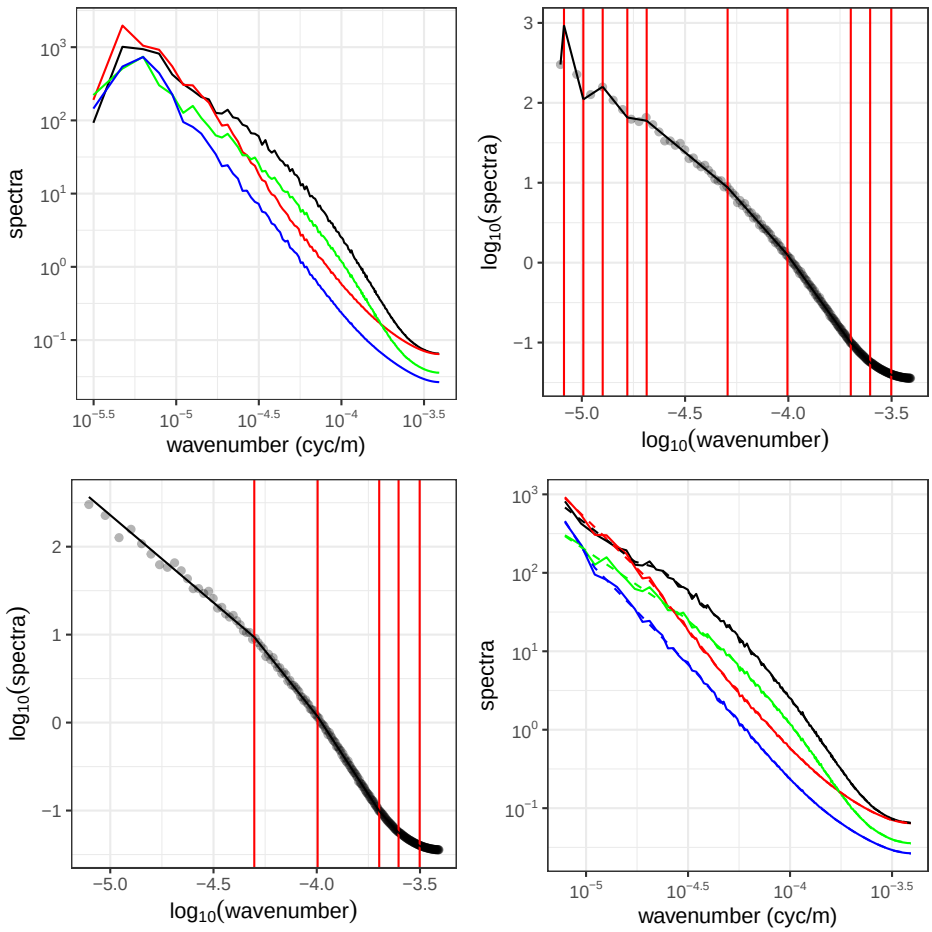

Figure 10: Application of `cpop` to `wavenumber_spectra` data. Log-log plot of raw data (top left) of horizontal wavenumber spectra of velocity for two months and two runs of an ocean model: February 2000 (black), August 2000 (red), February 2100 (green) and August 2100 (blue). Output from `cpop` applied to analyse the decay of spectra from August 2000, with y equal to log spectra and x equal to log wave number, with estimated homogeneous noise variance (top right) and estimated heterogeneous noise variance (bottom left). Log-log plot of fitted spectra for all four series (bottom right) with original data in full-lines and estimate in dashed lines.

```
R> {
R>  return(length(r2) * par[1] + par[2] * sum(x)
  + sum(r2 / (exp(par[1]+par[2]*x))))
R> }
R> est.hat <- optim(c(0,0), loglik)
R> sig2 <- exp(est.hat$par[1] + est.hat$par[2]*x)
R> res2 <- cpop(y, x, grid, sd = sqrt(sig2),
+               minseg = 0.2, beta =2 * log(200))
```

Here we have calculated the maximum likelihood estimates by using `optim` to minimise minus the log-likelihood. The resulting output from `cpop` is shown in the bottom left plot of Figure **??**. The first two changes could represent real regime transitions relating to the inviscid fluid physics that one would see in the real ocean (see Figure 6a of **?**), while the three changes for the largest values of `x` may relate to a breakdown in the numerical ocean model near the highest wavenumber of the ocean model grid (**?**). The estimates of the spectra for all four series, obtained by repeating this approach, is also shown in Figure **??**. For this application, the residuals from the fitted model appear to be uncorrelated and sub-Gaussian, so using the fit based on the default penalty choice is reasonable. Though one could also explore segmentations for other penalty choices using `cpop.crops` as in the previous section.

6. Acknowledgements

We would like to thanks Jessica Luo and Dan Whitt for access to and help with the wavenumber spectra data. Paul Fearnhead acknowledges funding from EPSRC grant EP/N031938/1.

A. Details of the Recursion

Here we describe how to calculate the inner minimisation

$$\min_{\alpha'} \left[F_k(\alpha') + \mathcal{C}_{l-1,l}(\alpha', \alpha) + \beta \right]$$

in the dynamic programming recursion – and how to do this so that the computational cost does not increase with the number of observations since the putative most recent changepoint. The function $F_k(\alpha')$ will be defined as the minimum of a set of quadratics. Denote this $q_i^{(k)}(\alpha')$ for $i = 1, \ldots, M_k$. Then we wish to solve

$$\min_{\alpha'} \left[\min_{i \in 1:M_k} \left\{ q_i^{(k)}(\alpha') + \mathcal{C}_{k,l}(\alpha', \alpha) \right\} + \beta \right] = \min_{i \in 1:M_k} \left\{ \min_{\alpha'} \left[q_i^{(k)}(\alpha') + \mathcal{C}_{k,l}(\alpha', \alpha) + \beta \right] \right\}.$$

So we only need to be able to calculate $\min_{\alpha'}[q(\alpha') + \mathcal{C}_{l-1,l}(\alpha', \alpha)]$, for any known quadratic $q(\alpha')$. In the following, we will denote the co-coefficients of $q(\alpha')$ by a, b and c, so

$$q(\alpha') = a + b\alpha' + c\alpha'^2.$$

Our approach will be to (i) calculate the co-coefficients of $\mathcal{C}_{k,l}(\alpha', \alpha)$ in constant time, through the use of summary statistics; (ii) calculate the co-coefficients of the sum $q_i^{(k)}(\alpha') + \mathcal{C}_{k,l}(\alpha', \alpha) +$

β; (iii) calculate the co-coefficients of the quadratic in α after we minimise with respect to α'. Steps (ii) and (iii) are trivial, but we give details below for completeness.

To simplify the exposition in the following we will use the convention that expressions that are of the form $0/0$ are equal to 0.

Define the following summary statistics for the data. These can be calculated prior to solving the dynamic programming recursion and enable the simple and quick calculation of $\mathcal{C}_{k,l}(\alpha', \alpha)$. These summary statistics are defined relative to the grid points – so a summary statistic with sub-script k will be based on all the data points for which $x_i \leq g_k$, and we define n_k to be the largest of observation such that $x_{n_k} \leq g_k$.

$$S_k^{(Y)} = \sum_{i=1}^{n_k} \frac{y_i}{\sigma_i^2}, \quad S_k^{(YY)} = \sum_{i=1}^{n_k} \frac{y_i^2}{\sigma_i^2}, \quad S_k = \sum_{i=1}^{n_k} \frac{1}{\sigma_i^2}$$

$$S_k^{(X)} = \sum_{i=1}^{n_k} \frac{x_i}{\sigma_i^2}, \quad S_k^{(XX)} = \sum_{i=1}^{n_k} \frac{x_i^2}{\sigma_i^2}, \quad S_k^{(XY)} = \sum_{i=1}^{n_k} \frac{x_i y_i}{\sigma_i^2}.$$

All summary statistics with sub-script 0, or that involve an empty sum, such that $n_k = 0$, are defined to be 0.

If we then define the co-coefficients of $\mathcal{C}_{k,l}(\alpha', \alpha)$, so that

$$\mathcal{C}_{k,l}(\alpha', \alpha) = A\alpha^2 + B\alpha\alpha' + C\alpha + D + E\alpha' + F\alpha'^2,$$

then tedious algebra gives that these coefficients are defined in terms of the summary statistics as

$$\begin{aligned}
A &= \frac{S_k^{(XX)} - S_l^{(XX)}}{(g_k - g_l)^2} - 2g_l \frac{S_k^{(X)} - S_l^{(X)}}{(g_k - g_l)^2} + g_l^2 \frac{S_k - S_l}{(g_k - g_l)^2}, \\
B &= 2(g_k + g_l) \frac{S_k^{(X)} - S_l^{(X)}}{(g_k - g_l)^2} - 2\frac{S_k^{(XX)} - S_l^{(XX)}}{(g_k - g_l)^2} - 2g_k g_l \frac{S_k - S_l}{(g_k - g_l)^2}, \\
C &= 2g_l \frac{S_k^{(Y)} - S_l^{(Y)}}{g_k - g_l} - 2\frac{S_k^{(XY)} - S_l^{(XY)}}{g_k - g_l}, \\
D &= S_k^{(YY)} - S_l^{(YY)}, \\
E &= 2\frac{S_k^{(XY)} - S_l^{(XY)}}{g_k - g_l} - 2g_k \frac{S_k^{(Y)} - S_l^{(Y)}}{g_k - g_l}, \\
F &= \frac{S_k^{(XX)} - S_l^{(XX)}}{(g_k - g_l)^2} - 2g_k \frac{S_k^{(X)} - S_l^{(X)}}{(g_k - g_l)^2} + g_k^2 \frac{S_k - S_l}{(g_k - g_l)^2}.
\end{aligned}$$

Adding $q_i^{(k)}(\alpha') + \beta$ to $\mathcal{C}_{k,l}(\alpha', \alpha)$ just changes the coefficients of powers of α' – that is D increases by $a + \beta$, E increases by b and F increases by c. Minimising the resulting quadratic with-respect to α' gives a quadratic of the form $a' + b'\alpha + c'\alpha^2$ where

$$a' = D + a + \beta - \frac{(E + b)^2}{4(F + c)},$$

$$b' = C - \frac{(E + b)B}{2(F + c)},$$

$$c' = A - \frac{B^2}{4(F+c)}.$$

Affiliation:

Paul Fearnhead
Department of Mathematics and Statistics
Lancaster University
Lancaster,
LA1 4YF, UK
E-mail: p.fearnhead@lancaster.ac.uk

Daniel Grose
Department of Mathematics and Statistics
Lancaster University
Lancaster,
LA1 4YF, UK
E-mail: dan.grose@lancaster.ac.uk