Journal of Statistical Software

June 2017, Volume 78, Code Snippet 2. doi: 10.18637/jss.v078.c02

Locally Adaptive Tree-Based Thresholding Using the treethresh Package in R

Ludger Evers
University of Glasgow

Tim Heaton
University of Sheffield

Abstract

This paper introduces the **treethresh** package offering accurate estimation, via thresholding, of potentially sparse heterogeneous signals and the denoising of images using wavelets. It gives considerably improved performance over other estimation methods if the underlying signal or image is not homogeneous throughout but instead has distinct regions with differing sparsity or strength characteristics. It aims to identify these different regions and perform separate estimation in each accordingly. The base algorithm offers code which can be applied directly to any one-dimensional potentially sparse sequence observed subject to noise. Also included are functions which allow two-dimensional images to be denoised following transformation to the wavelet domain. In addition to reconstructing the underlying signal or image, the package provides information on the believed partitioning of the signal or image into its differing regions.

Keywords: CARTs, wavelets, thresholding, sparsity, denoising, heterogeneous, partition.

1. Methodology

The **treethresh** (Evers and Heaton 2009, 2017) package is intended to allow improved estimation, via thresholding, of signals and images that have been observed subject to noise. The basic underlying algorithm is suitable for application on any underlying signal which may be potentially sparse but is particularly justified if we believe that this sparsity may vary along the course of the signal. There are many instances where such sparsity may be observed but one that is especially significant occurs when denoising via wavelets. Here although the original image may not itself be sparse, upon transformation to the wavelet domain it is expected to have a parsimonious representation. For this reason we also provide two specific algorithms to perform wavelet denoising. We illustrate our method both in the context of single sequence estimation and wavelet denoising using simulated data (for clarity of exposition) as well as a real world example.

The underlying *TreeThresh* algorithm is based upon the *EbayesThresh* approach (Johnstone and Silverman 2004, 2005a,b) which demonstrated that signal estimation could be considerably improved by adapting to the global strength of the particular signal under observation and the selection of a suitable strength-specific threshold. *TreeThresh* attempts to improve upon this methodology by also allowing adaptivity to potential local variations in the strength along the course of the signal. This idea has been applied before by Heaton (2009) whereby signal strength was estimated to vary smoothly by kernel or spline smoothing although here we consider more abrupt changes in this strength. Specifically *TreeThresh* aims to partition the observed data into disjoint heterogeneous regions corresponding to potential changes in the strength level within the signal. Within each of these partitions, it is expected that there is a similar level of signal strength and so *EbayesThresh* can be applied to give a suitable threshold in each separately.

The performance of *TreeThresh* has been demonstrated in both simulation studies and applications to practical examples where we have found it to outperform *EbayesThresh*, see Evers and Heaton (2009) for more details. Furthermore, both in single sequence estimation and image denoising via wavelets, our methodology is able to provide the user with a partition of the signal into disjoint regions with differing sparsity characteristics which may, in itself, be of interest.

The **treethresh** package is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=treethresh`. It has been developed to depend upon the additional packages **EbayesThresh** (Silverman 2010) for the base thresholding; and **wavethresh** (Nason 2010) to perform the wavelet decompositions and reconstructions needed for image denoising. These packages are required dependencies of **treethresh** in R and do not require independent installation.

1.1. Model

We will first explain the methodology as applied to single, potentially sparse, sequences before demonstrating the application to wavelet image denoising in Section 3. Suppose we have, after possible rescaling to obtain unit variance, observed a sequence $\mathbf{X} = (X_i)_{i\in\mathcal{I}}$ satisfying

$$X_i = \mu_i + \epsilon_i, \quad \text{for } i \in \mathcal{I},$$

where $\boldsymbol{\mu} = (\mu_i)_{i\in\mathcal{I}}$ is a possibly sparse signal (i.e., some/most of the μ_i are believed to be zero), the ϵ_i are independent $N(0,1)$ noise, and $\mathcal{I}$ is a possibly multidimensional index domain. Being a generalization of *EbayesThresh*, the *TreeThresh* method is based on assuming a mixture between a point mass at zero (denoted δ_0) and a signal with density $\gamma(\cdot)$ as the prior distribution for the μ_i:

$$f_{\text{prior}}(\mu_i) = (1 - w_i)\delta_0 + w_i\gamma(\mu_i).$$

However, in contrast to the *EbayesThresh* method, the mixing weights w_i depend on the index i, i.e., the underlying signal can be heterogeneous (in the sense of not being everywhere equally sparse). We assume there is a partition of the index space $\mathcal{I} = P_1 \cup \ldots \cup P_p$ (with $P_k \cap P_l = \emptyset$ for $k \neq l$) such that the weights within each region P are (almost) constant.

This yields

$$l(\mathbf{w}) = \sum_{i\in\mathcal{I}} \log f(x_i|w_i) = \sum_{i\in\mathcal{I}} \log\left((1 - w_i)\phi(x_i) + w_i(\gamma \star \phi)(x_i)\right)$$

as the marginal log-likelihood of the observed $\mathbf{x}$. Here, ϕ denotes the density of the standard normal distribution and $\gamma \star \phi$ the convolution of γ and ϕ. In commonality with *EbayesThresh*, the **treethresh** package uses a double exponential distribution[1] with fixed scale parameter a (set to 0.5 by default) as $\gamma(\cdot)$ permitting this convolution to be written in closed-form. Other possibilities exist but Silverman (2010) uses this as the default in their **EbayesThresh** package having found it to be the best performing prior for a wide range of sparse signals as well as having shown that its heavy tails provide theoretical benefits (for further details see Johnstone and Silverman 2005a).

1.2. Estimation

The *TreeThresh* procedure contains two steps, both of which are implemented in the package. Firstly, given the observed data, we estimate a suitable partition $\mathcal{P}$ splitting the signal into disjoint regions according to potential variations in strength and simultaneously estimate the mixing weight w_i in each. Secondly, with these mixing weights we select a suitable mixing-weight specific threshold for the accompanying partition $t(w_i)$ before applying our chosen thresholding rule.

Partitioning and mixing weight estimation

The partitioning is found using an algorithm that resembles those used in recursive partitioning algorithms such as *classification and regression trees* (CARTs; Breiman, Friedman, Olshen, and Stone 1984). In brief, we initially create a nested sequence of increasingly fine partitions. Cross-validation is then used to prune (remove) some of these partitions and identify the "best" overall partition of the signal. Section 2 gives a more detailed description of the specific algorithm.

Having identified an "optimal" partition, we then estimate the mixing weights in each region by maximizing the log-likelihood of the observations it contains. This mixing weight estimation corresponds to carrying out the *EbayesThresh* algorithm for each region separately.

Thresholding

Having created our partition and identified the mixing weights w_i in each region, we can use these to estimate the underlying signal μ_i via thresholding with a weight specific threshold. The thresholding can be done in a variety of ways but the exact choice of technique is much less important than the threshold level $t(w_i)$. We provide three such choices for the threshold function but all use the same threshold value – that obtained from the posterior median method. This threshold value has been shown to have strong theoretical properties (Johnstone and Silverman 2004, 2005a).

Posterior median: The posterior median of μ_i given $X_i = x$ is shown in Figure 1 as a function of x (solid line). As pictured, it has a thresholding property where for $x \in [-t_{\hat{w}_i}, t_{\hat{w}_i}]$ the posterior median is zero. For the mathematical details see e.g., Johnstone and Silverman (2005b, Section 6.1).

Hard thresholding: Alternatively, we allow use of the $t_{\hat{w}_i}$ obtained from the posterior me-

[1] I.e., a distribution on the real line with density $\gamma(u) = \frac{a}{2}\exp(-a|u|)$.

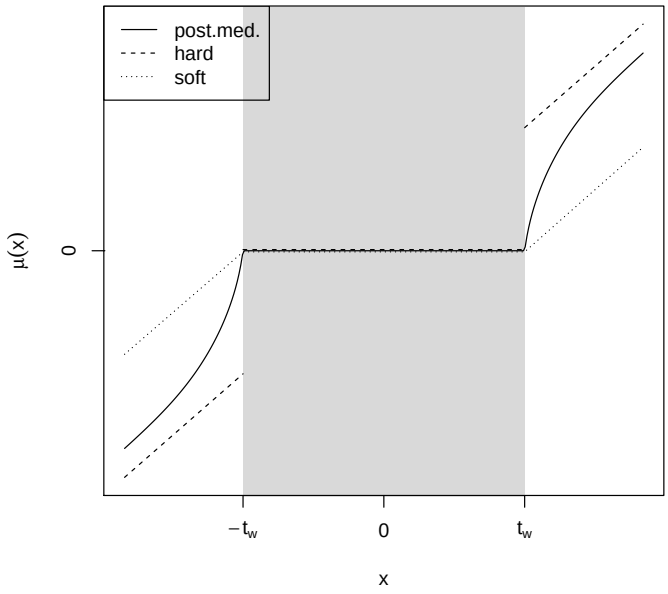


Figure 1: Comparison of the three thresholding rules.

dian within the hard thresholding rule

$$\hat{\mu}_i^{\text{hard}}(x) = \begin{cases} x & \text{for } x < -t_{\hat{w}_i} \\ 0 & \text{for } -t_{\hat{w}_i} \leq x \leq t_{\hat{w}_i} \\ x & \text{for } x > t_{\hat{w}_i} \end{cases}$$

This threshold rule is illustrated with a dashed line in Figure 1. Such hard thresholding is discontinuous at $-t_{\hat{w}_i}$ and at $t_{\hat{w}_i}$.

Soft thresholding: Finally, the package gives the option of the soft thresholding rule

$$\hat{\mu}_i^{\text{soft}}(x) = \begin{cases} x + t_{\hat{w}_i} & \text{for } x < -t_{\hat{w}_i} \\ 0 & \text{for } -t_{\hat{w}_i} \leq x \leq t_{\hat{w}_i} \\ x - t_{\hat{w}_i} & \text{for } x > t_{\hat{w}_i} \end{cases}$$

where again $t_{\hat{w}_i}$ is the threshold obtained from the posterior median. This rule (dotted line in Figure 1) is continuous, but is biased even for large values of x.

By default, the **treethresh** package uses the posterior median.

2. Algorithmic details

The partitioning algorithm aims to find a partitioning of the index set $\mathcal{I} = P_1 \cup \ldots \cup P_p$, $P_k \cap P_l = \emptyset$, such that $\{w_i,\ i \in P_k\}$ is (almost) constant. An exhaustive search over all possible rectangular partitions is prohibitive, thus the method uses a greedy "one step look-ahead" strategy of recursively partitioning the signal: The canonical step of the algorithm is to split one rectangular region P into two rectangular regions L and R. As there are only a small number of these "splits", an exhaustive search can be performed. An optimal cutoff

should split the current region P into two new regions hopefully corresponding to changes in the sparsity and its heterogeneity. This can be measured by looking at a test of the null hypothesis that the signal is equally sparse in both regions, i.e., $H_0 : w^{(L)} = w^{(R)}$. By default, the software uses the score statistic, as this does not require computing $w^{(L)}$ and $w^{(R)}$ for all pairs of candidate regions L and R, see Evers and Heaton (2009) for the mathematical details.

This canonical step of splitting one rectangular region into two rectangular regions is carried out recursively. This (first) step of the algorithm is implemented in the functions `treethresh` and `wtthresh` dependent upon whether you wish to simply estimate a possibly sparse vector μ as described in Section 1.1 or denoise an image by taking advantage of its sparsity in the wavelet domain (see Section 3 for more on the differences between these two functions).

In order to avoid overfitting, it is important not to estimate too fine a partition. One possibility could be to use stopping rules based on the test statistic of the score test (or a likelihood ratio test). However these suffer from two drawbacks. First, it is difficult to find the correct critical value, as we are testing data-driven hypotheses. Second, using a naïve stopping rule would lead to a short-sighted strategy for choosing the optimal partition: A seemingly worthless split might turn out to be an important boundary in a more complex partition. Thus we propose, in complete analogy with the CART algorithm, to initially estimate too fine a partition and then reduce its complexity by finding a coarser super-partition such that

$$l_{\mathsf{P}} - \alpha \cdot |\mathsf{P}|$$

is maximal, where l_{P} is the log-likelihood obtained by partition P and $|\mathsf{P}|$ is the number of regions in P.

Just as in the case of CARTs, one can show (see e.g., Ripley 1996, Section 7.2) that there exists a nested sequence of partitions which maximize the penalized log-likelihood over different ranges of α. Figure 2 illustrates this idea. The "optimal" value of α can be found using cross-validation. As the parameter α is on a scale which is difficult to interpret, the software works with the parameter $C = \frac{\alpha}{\alpha_0}$, where α_0 is the value that would yield a partition consisting of a single region. This parameter C can thus take values between 0 (no pruning) to 1 (partition reduced to a single region).

It might be expected that one would choose the value of C that yields the largest predictive log-likelihood. However, it turns out to be often better in practice to use a simpler model (corresponding to a larger value of C) if the corresponding predictive log-likelihood is not much worse than that of the best model. Thus the package uses by default the largest C for which the difference to the best predictive log-likelihood is less than half the standard error of the best predictive log-likelihood.

This second step of the algorithm can be carried out by calling the function `prune`. Sections 4.1 and 4.2 contain two examples illustrating both steps of the algorithm and explaining the output generated. For a more detailed description together with some asymptotic properties see Evers and Heaton (2009).

3. Application to wavelet coefficients

Perhaps the most common application of thresholding is for denoising an observed, possibly multidimensional, signal (or image) using wavelets. Here we do not expect the original image

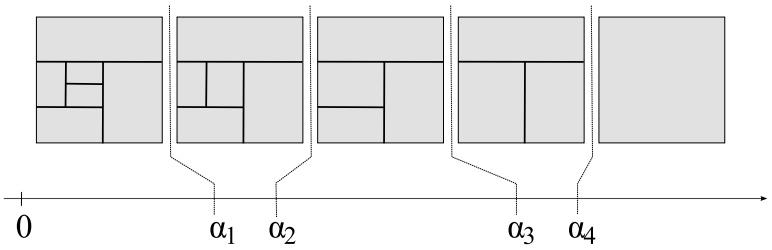


Figure 2: Example of a nested sequence of partitions corresponding to different values of α. As α increases, the optimal penalized likelihood partition becomes coarser and is nested within the optimal partition for smaller values of α.

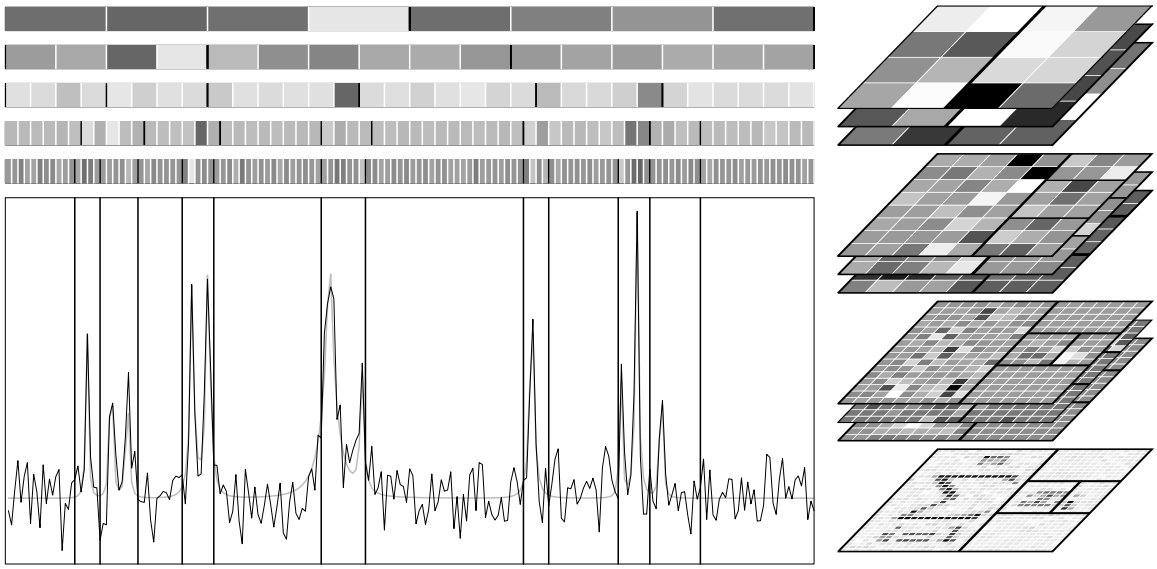
to itself be sparse but obtain this sparsity by first transforming the noisy signal to the wavelet domain where it is expected that the underlying signal has a parsimonious representation. The observed wavelet coefficients are thus thresholded before being transformed back to the original domain to provide a hopefully noise-free version of the original signal.

Denoising of signals/images in this way provokes an additional question of whether we would wish to partition our image in the original untransformed domain or rather within each individual level of the wavelet coefficient space. The former approach is appealing in that it permits the interpretation of the untransformed image as containing distinct regions with differing characteristics and allows partitioning information to be shared across differing levels of the wavelet transform which may improve estimation. Identification of such regions in the original domain may also be of independent interest to the user. Figure 3 illustrates the idea of partitioning the original untransformed domain and shows how the partition of the original domain is transferred to the wavelet coefficients.

Our code provides the possibility to apply both types of partitioning algorithms. *Levelwise TreeThresh* simply applies the partitioning algorithm explained in Section 2 to each level of the wavelet coefficients independently. On the other hand, *Wavelet TreeThresh* combines the information across different levels of the wavelet transform to partition in the original space domain. As well as providing an estimate of the noise-free image/signal, the output of *Wavelet TreeThresh* provides the partition of the space domain selected for the user to see. For an example of how to apply both the *Levelwise* and *Wavelet TreeThresh* algorithms see Section 4.2.

4. Using the software

To explain the implementation of the package we provide a set of examples increasing in complexity. We hope these will guide the reader through the various steps of the algorithm before ending with an application of the methodology to a real life example of image denoising via wavelets. We commence with a highly simplified example of the thresholding of a single artificial sequence with varying sparsity. This example aims to show the reader how *TreeThresh* is able to partition such a sequence, correctly identify the regions of differing sparsity and choose an appropriate threshold in each. We then show how this fundamental idea can be incorporated into the context of wavelet denoising. We present a small simulation study showing how, once in the wavelet domain, the wavelet coefficients of functions and images often fall in regions with varying sparsity. In this section we also demonstrate how



(a) Illustration for a one-dimensional signal.

(b) Illustration for a two-dimensional signal.

Figure 3: Underlying signal in the original domain (bottom) and corresponding wavelet coefficients at fine levels. The thick solid lines indicating the partitions illustrate how the partition of the original index domain is transferred to each level of the wavelet coefficients.

our method is robust to deviations from the Gaussian noise assumption. In real world images it is not uncommon to instead have "salt and pepper" noise where there are a few extremely noisy measurements polluting the underlying image. We show how our method is still able to perform denoising in such instances. Finally we conclude with denoising of a real life image of 3T3 cells illuminated by laser.

4.1. Single sequence estimation of a potentially sparse signal via thresholding

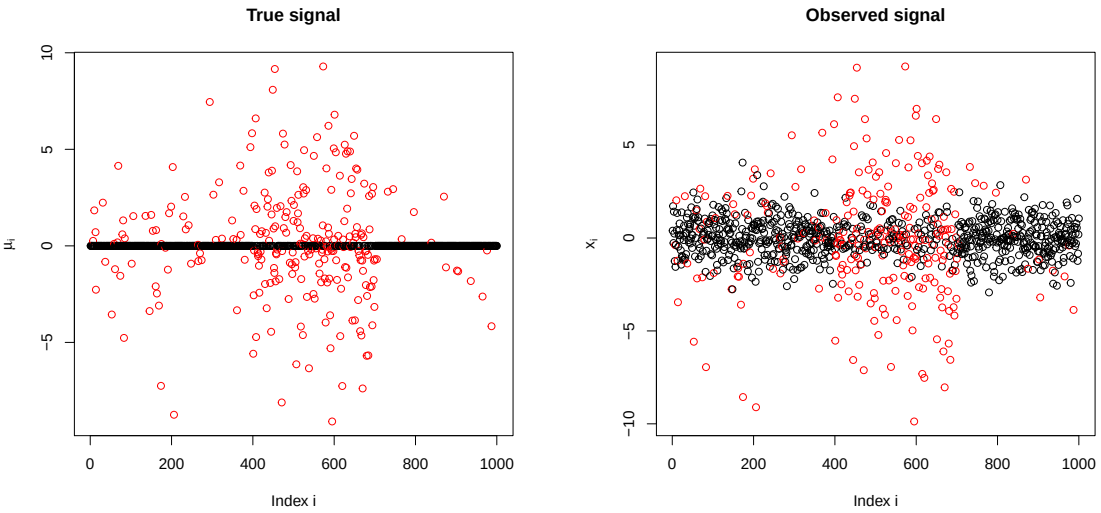
We begin with a short example (very similar to that given in the help file of `treethresh`) to illustrate how the **treethresh** package can be used to threshold a simple sequence and identify the estimated partition. While the example itself is somewhat artificial we spend some time explaining the features of the model and output since this single sequence estimation provides the underlying building block for our more advanced wavelet techniques.

Creating an artificial signal

First let us start with creating a sparse underlying signal which is rather dense towards the middle and sparse at the beginning and at the end. This is done by choosing a vector containing the probabilities w_i that $\mu_i \neq 0$.

```
R> w.true <- c(rep(0.15, 400), rep(0.6, 300), rep(0.05, 300))
```

The true signal $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_{1000})$ is then created by drawing the non-zero μ_i from a Laplace



(a) Underlying true signal $\boldsymbol{\mu}$.

(b) Observed signal $\mathbf{x}$.

Figure 4: The underlying true signal and observed noisy signal $\mathbf{x}$ (entries corresponding to non-zero μ_i are shown in red).

distribution.

```
R> mu <- numeric(length(w.true))
R> non.zero.entry <- runif(length(mu)) < w.true
R> num.non.zero.entries <- sum(non.zero.entry)
R> mu[non.zero.entry] <- rexp(num.non.zero.entries, rate = 0.5) *
+    sample(c(-1, 1), num.non.zero.entries, replace = TRUE)
R> mu[1:12]

 [1] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
 [7] 0.2581282 0.0000000 0.0000000 1.8380074 0.0000000 0.0000000
```

The specific signal generated is displayed in Figure 4(a). Our hope is not only to recover this signal after observation subject to noise but also to identify the three distinct regions of differing sparsity corresponding to changes in the selected $\mathbf{w}$.

We finish by creating the observed noisy signal $\mathbf{x} = (x_1, \ldots, x_{1000})$ with the addition of white noise to the $\boldsymbol{\mu}$. Figure 4(b) displays the simulated "observed" signal.

```
R> x <- mu + rnorm(length(mu))
```

Estimating the noise variance and rescaling

In the above example we know that the noise has unit variance. Ideally a user would know the error in their measurements, for example by running some sort of blank where the underlying signal is known to be empty. However, in many practical settings this may not be the case

True proportion of signal w	0	0.01	0.05	0.1
Correction factor	1.482602	1.473273	1.435957	1.389315
True proportion of signal w (ctd.)	0.2	0.3	0.5	1
Correction factor (ctd.)	1.296104	1.203145	1.019313	0.6176064

Table 1: Correction factors that would give an unbiased estimate of the standard deviation of the noise.

and the standard error will need to be estimated from the signal/image itself. Such a priori estimation is a difficult problem. In the case of sparse signals (and hence by extension wavelet demonizing) the medium absolute deviation as implemented in the function `mad` can be used to get a rough idea of the standard error of the noise. However, the correction factor of 1.4826 used by `mad` is only unbiased if no signal is present, i.e., $\boldsymbol{\mu} = \mathbf{0}$. If a signal is present, it overestimates the standard deviation of the noise. For a homogeneous signal with $w_i \equiv 0.5$, `mad` overestimates the standard deviation by about 50%. To illustrate this bias, Table 1 gives the correction factors one could use (instead of 1.4826) for a homogeneous signal if the w_i were constant and known (although this would of course defeat the purpose of the *EbayesThresh* or *TreeThresh* algorithms).

When using `mad` to estimate the standard error of the noise in our example signal, we use a correction factor of 1.3 to account for the fact that our signal is fairly dense:

```
R> sdev <- mad(x, constant = 1.3)
R> sdev

[1] 0.9973816
```

As indicated in Section 1.1, our algorithm assumes that the signal that is fed into the `treethresh` function has unit variance and so it is necessary to rescale appropriately using our estimate `sdev`:

```
R> x <- x / sdev
```

Signal strength and partition estimation

We are now ready to apply the `treethresh` function, which estimates the partitioning and the corresponding w_i.

```
R> library("treethresh")
R> x.tt <- treethresh(x)
```

The element `splits` contains detailed information about the partition. We provide a guide to help interpret this output in the next section. In short however, each row corresponds to a region or a split, respectively. The columns are as follows:

`id`: Integer uniquely identifying the region / split.

`parent.id`: The modulus of `parent.id` is the `id` of the parent region. If the current region is to the left of the split, `parent.id` is negative, otherwise it is positive.

`dim`: The dimension (indexed starting at 0) used to define the split.

`pos`: The position of the split.

`left.child.id` / `right.child.id`: If the region has been split further, these two columns contain the `id` of the newly created "children", otherwise `NA`.

`crit`: The value of the criterion (i.e., by default the score test) for carrying out this split.

`w`: The value of $\hat{w}^{(P)}$ used in this region (before splitting further).

`t`: The corresponding threshold $t_{\hat{w}^{(P)}}$ in this region (before splitting further).

`loglikelihood`: Contribution of the observations in this region to the log-likelihood (before splitting further).

`alpha` / `C`: If the value of C (or α) in the pruning step is chosen larger than the number given, this region (*not* split) would be removed in the pruning, and only its "parent" or another "ancestor" would be retained.

The output of `treethresh` can be inspected as follows:

```
R> nrow(x.tt$splits)

[1] 63

R> head(x.tt$splits, n = 10)

      id parent.id dim pos left.child.id right.child.id      crit
 [1,]  1        NA   0 745             2             63 51.794514
 [2,]  2        -1   0 393             3             32 52.354525
 [3,]  3        -2   0 369             4             31 19.843343
 [4,]  4        -3   0   9             5              6  4.821266
 [5,]  5        -4  NA  NA            NA             NA        NA
 [6,]  6         4   0 145             7             14  2.741212
 [7,]  7        -6   0  83             8             13 21.499363
 [8,]  8        -7   0  51             9             12  4.202844
 [9,]  9        -8   0  14            10             11 18.933715
[10,] 10        -9  NA  NA            NA             NA        NA
                 w         t loglikelihood     alpha          C
 [1,] 0.291419714 2.1945073  448.22531923        NA         NA
 [2,] 0.369894137 1.9968887  463.49453253 19.957311 1.00000000
 [3,] 0.139722902 2.6506361   66.09874994 19.957311 1.00000000
 [4,] 0.154346257 2.5975401   67.70449641  1.981756 0.09929975
 [5,] 0.008961814 3.7169222   -0.02479908  1.981756 0.09929975
 [6,] 0.158658582 2.5824785   68.15741062  1.981756 0.09929975
 [7,] 0.086433684 2.8834517   17.24646557  1.981756 0.09929975
 [8,] 0.176956791 2.5211251   19.60181045  1.981756 0.09929975
 [9,] 0.050922031 3.1079077    0.56249679  1.981756 0.09929975
[10,] 0.918786110 0.3074389    3.38213713  1.981756 0.09929975
```

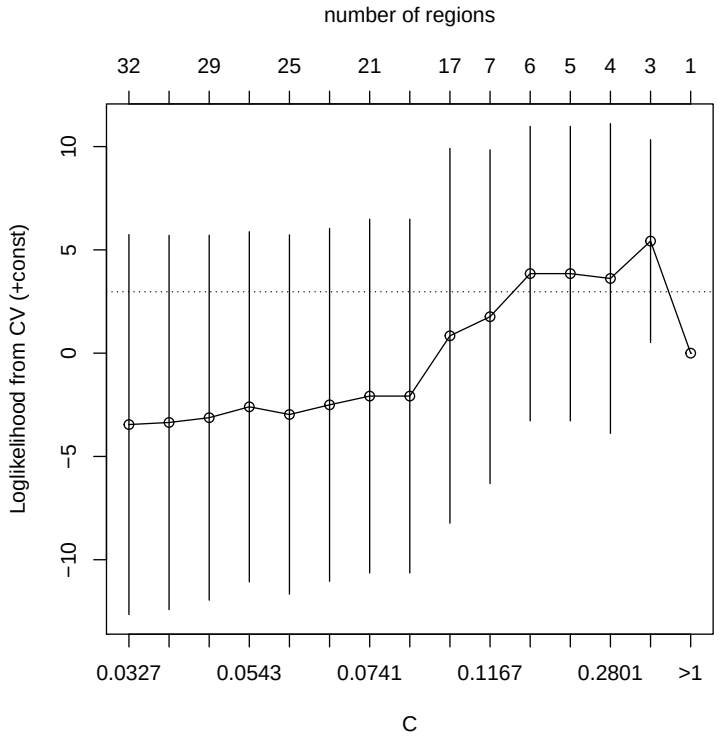


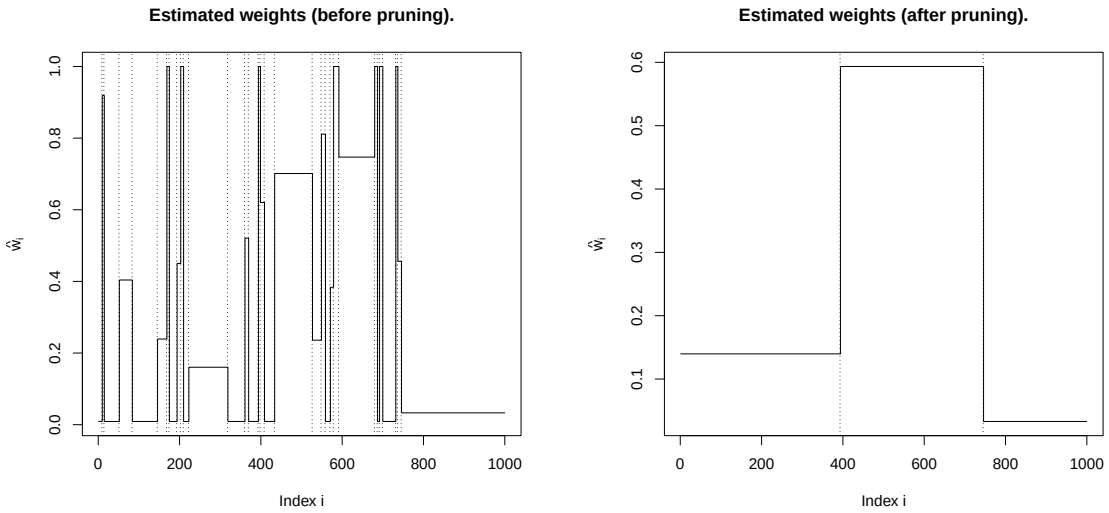
Figure 5: Predictive log-likelihood as a function of the complexity parameter C.

On the left hand side of Figure 6 we show the estimated partition and the estimated weights w_i created by `treethresh` before pruning. As described in Section 2 and as can be clearly seen from the plot, the partition estimated in this first step constitutes an overfit to the data. There are far too many partitions and thus we need to carry out a second, pruning step that reduces the complexity of the estimated partition.

```
R> x.ttp <- prune(x.tt)
R> x.ttp$splits
```

```
     id parent.id dim pos left.child.id right.child.id     crit
[1,]  1        NA   0 745             2            63 51.79451
[2,]  2        -1   0 393             3            32 52.35453
[3,]  3        -2  NA  NA            NA            NA       NA
[4,] 32         2  NA  NA            NA            NA       NA
[5,] 63         1  NA  NA            NA            NA       NA
              w        t loglikelihood    alpha  C
[1,] 0.29141971 2.194507    448.225319       NA NA
[2,] 0.36989414 1.996889    463.494533 19.95731  1
[3,] 0.13972290 2.650636     66.098750 19.95731  1
[4,] 0.59349144 1.427620    420.521354 19.95731  1
[5,] 0.03300857 3.274768      1.519837 19.95731  1
```

In Figure 6(b) we show the estimated partition after pruning has taken place. Most of the splits have now been removed and we are left with just three distinct sections. As can be seen



(a) Before pruning.

(b) After pruning.

Figure 6: Estimated partition and weights before and after pruning.

these correspond well to the changes in the weight vector we used to create the simulated signal both in their location and the estimated values $\hat{w}_i$.

Within `prune` a cross-validation step is used to select the "optimal" complexity parameter C. This is done automatically but we illustrate the approach in Figure 5. By default `prune` uses five-fold cross validation (which can be changed using the argument `v`) to estimate the predictive log-likelihood. The predictive log-likelihood is highest for partitions with three regions, and the simpler partition having only one region is more than half a standard error worse (being below the dotted line), thus we retain the partition with three regions.

Thresholding

Having found the optimal partition, all that is then required is to use the estimated weights in each region to threshold the sequence. Figure 7 shows the corresponding threshold. The thresholding is done using the function `thresh`, which uses by default the posterior median.

```
R> mu.hat <- thresh(x.ttp)
```

Finally, we need to scale the reconstructed signal $\hat{\boldsymbol{\mu}}$ back to the original domain.

```
R> mu.hat <- mu.hat * sdev
```

Figure 8 shows the reconstructed sequence.

Interpreting the partitioning output

The detailed partitioning output provided by the package may initially seem a little complex. Here we give a short explanation of how to interpret that information for the case of the pruned `x.ttp$splits`.

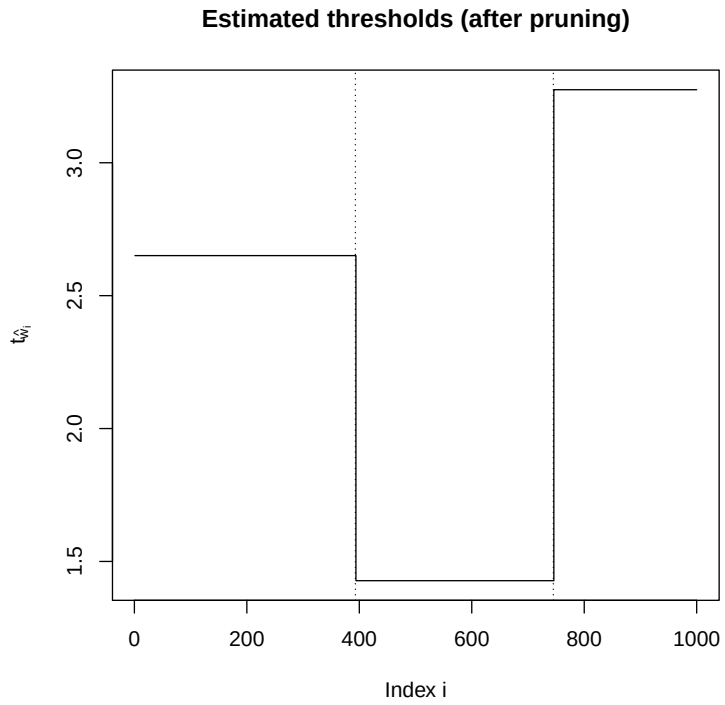


Figure 7: Estimated thresholds $t_{\hat{w}_i}$ of the partition after pruning.

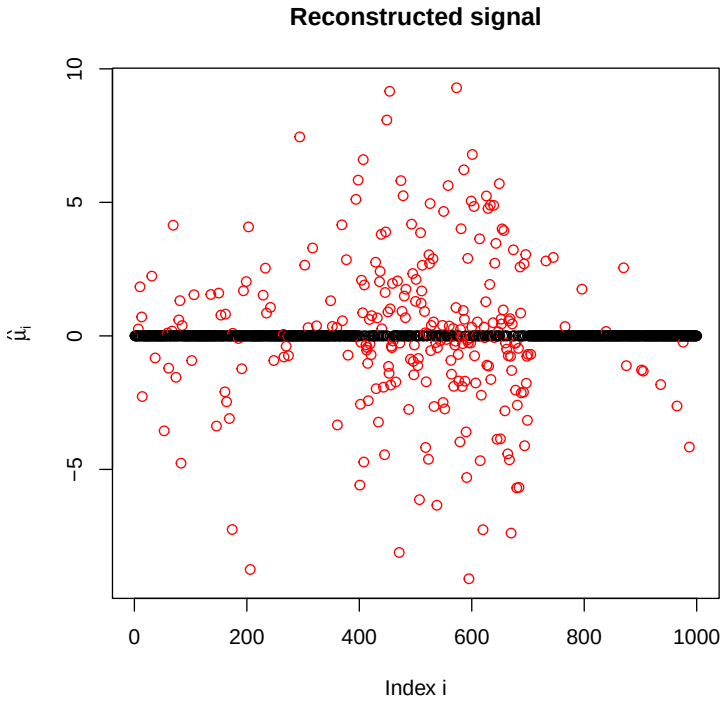


Figure 8: Reconstructed signal $\hat{\mu}_i$ in the original scale (entries corresponding to non-zero true signal μ_i in red).

The user should commence by looking at the first row of output which corresponds to the case of no partitioning. This tells us that the global signal (with no proposed splitting) is estimated to have a sparsity mixing weight of $\hat{w}_i = 0.29$ for all i with a corresponding threshold to apply of $t_{\hat{w}_i} = 2.19$. This "no split" situation is not however optimal and the algorithm suggests an initial split at position 745 into the subregions with identifiers 2 and 63 (to the left and right of this split respectively).

Information on the first of these regions is given in the row with `id = 2`. This corresponds to those elements μ_i for $i = 1, \ldots, 745$. In this region the sparsity weight is estimated to be $\hat{w}_i = 0.37$. Again however the algorithm suggests another split, this time at position 393 into the subpartition with identifiers 3 and 32. This however is as far as we proceed in our splitting since the rows corresponding to these identifiers have `NA` values for both `left.child.id` and `right.child.id`.

Similarly, for more detail on the right hand side of the initial split (and potential further splitting) we look at the row with identifier 63. Here we estimate the signal to be extremely sparse with $\hat{w}_i = 0.03$. It is not split further.

In summary, the method suggests an optimal partition of the signal $\boldsymbol{\mu}$ into three regions. The first being for those $i = 1, \ldots, 393$ with an estimated value $\hat{w}_i = 0.14$; the second being for $i = 394, \ldots, 745$ with $\hat{w}_i = 0.59$; and the third for $i = 746, \ldots, 1000$ with $\hat{w}_i = 0.03$. This is shown on the right hand side of Figure 6. If we refer back to the original vector of w_i selected we can see that the method has identified fairly accurately both the partition locations and the corresponding sparsity in each region.

The column `dim` indicates which dimension is used for the split. In the one-dimensional example presented in this section the column `dim` is always zero. If a two-dimensional signal is thresholded, then `dim` is either 0 or 1, depending on whether splits are horizontal or vertical.

4.2. Denoising an image by thresholding wavelet coefficients

In this section we show how the ideas of sparse sequence partitioning and estimation via thresholding presented in the previous section can be applied to wavelet coefficients permitting improved image denoising. We provide explanations of the function `wavelet.treethresh` which is a high level function automating the process as well as describe how estimation can be performed manually using lower level functions.

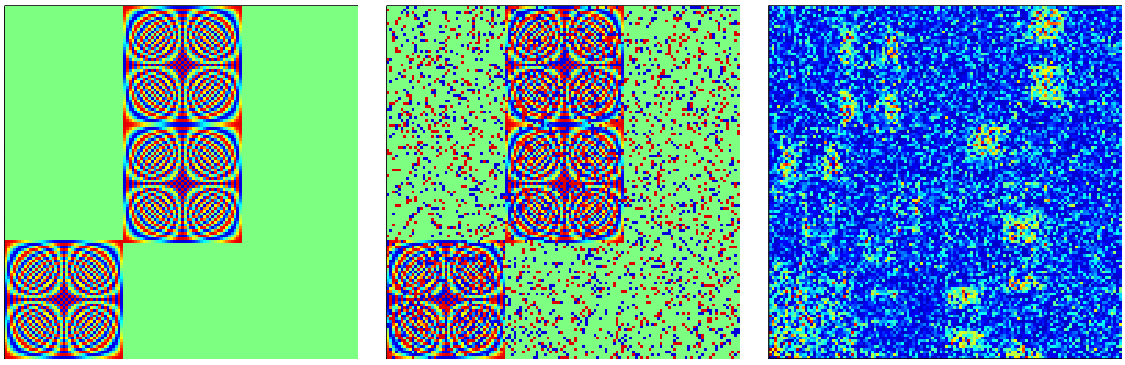
We also show how our image denoising is still able to perform for images with deviations from the Gaussian noise model assumption. We use an artificial example to which we add "salt-and-pepper" noise before attempting to recover the original image. A small simulation study is presented where we compare our *TreeThresh* method to the current state-of-the-art *EbayesThresh* method illustrating the significant improvement that can be achieved by partitioning the image and thresholding each region adaptively.

Preparing the example

This example uses the image `tiles` available with the **treethresh** package and shown in Figure 9(a).

```
R> data("tiles", package = "treethresh")
```

To see whether the *TreeThresh* algorithm is able to recover this image observed subject to



(a) Original image. (b) Noisy image. (c) Wavelet decomposition.

Figure 9: Image `tiles` (Panel (a)) and image with "salt-and-pepper" noise added (Panel (b)). In Panel (c) we show the 2D wavelet decomposition of the noisy image illustrating the localized levels of sparsity. The color palette used was obtained using the function `matlab.like2` from the package **colorRamps** (Keitt 2012).

noise we add "salt-and-pepper" noise, rather than the Gaussian noise assumed by the model. "Salt-and-pepper" noise sets a fixed proportion of the pixels, 20% in our case, to either the largest value ("white") or the smallest value ("black"), chosen randomly for each pixel.

```
R> tiles.noisy <- tiles
R> subset <- sample(length(tiles), round(length(tiles) * 0.2))
R> tiles.noisy[subset] <- range(tiles)[sample(1:2, length(subset),
+    replace = TRUE)]
```

Figure 9(b) shows the noisy image generated. The corresponding signal to noise ratio is about 1:1.

In order to be able to use the *TreeThresh* algorithm, we need to compute the wavelet transform of the image. We do this using the function `imwd` from the package **wavethresh** (Nason 2010).

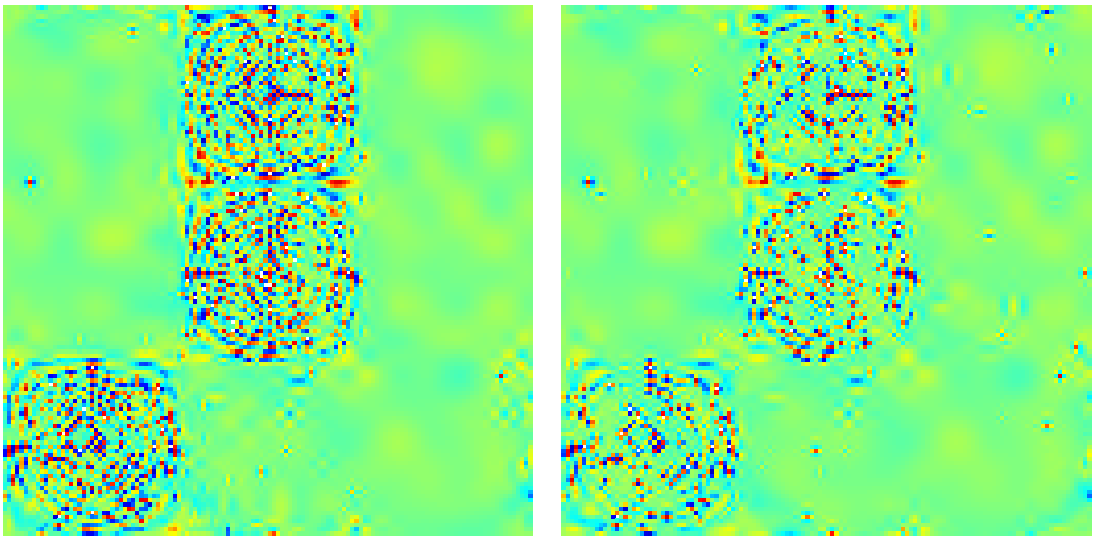
```
R> tiles.noisy.imwd <- imwd(tiles.noisy)
```

Using the high-level function `wavelet.treethresh`

The function `wavelet.treethresh` allows for thresholding in a more user-friendly way by calling the relevant functions `extract.coefficients`, `estimate.sdev`, `treethresh` / `wttresh`, `prune`, and `thresh` internally as well as rescaling the coefficients so that the noise has approximately unit variance. This section explains how to use this more user-friendly interface, see the following section for the commands required to carry out the thresholding step-by-step or if information on the estimated partition is required.

```
R> tiles.noisy.imwd.threshed <- wavelet.treethresh(tiles.noisy.imwd)
```

The default approach is to create the partition in the original untransformed domain. If the *Levelwise TreeThresh* algorithm is desired instead then the user can simply add an additional argument `levelwise = TRUE`.



(a) *TreeThresh*. (b) *EbayesThresh*.

Figure 10: Image reconstructed by the *TreeThresh* algorithm (Panel (a)) compared to the one reconstructed by the *EbayesThresh* algorithm.

	TreeThresh	*EbayesThresh*
Mean l_2 loss	5488.534	7255.330
Standard deviation of l_2 loss	350.9619	223.1894

Table 2: Comparison of the l_2 loss incurred by *TreeThresh* and *EbayesThresh* for the tiles data (1,000 replications).

Having thresholded the wavelet coefficients, we transform them back to the original domain using the function `imwr` from the package **wavethresh** to recover our original signal with the noise hopefully removed. The **wavethresh** package is set as a required dependency for **treethresh** and so does not require independent installation.

```
R> tiles.denoised <- imwr(tiles.noisy.imwd.threshed)
```

Figure 10(a) shows the reconstructed image and compares it to the result obtained by the *EbayesThresh* algorithm (Panel (b)). The result of the *TreeThresh* method appears sharper and the regions without the three square designs are notably clearer than the *EbayesThresh* reconstruction. The corresponding l_2 loss is 5963.337 for the *TreeThresh* algorithm and 7682.055 for the *EbayesThresh* algorithm. Table 2 gives the l_2 loss for 1,000 replications. *TreeThresh* outperforms *EbayesThresh* for every single replication and leads on average to a reduction of the l_2 loss by almost a quarter. Neither method appears to be hampered by the presence of "salt-and-pepper", rather than Gaussian, noise. This suggests that both methods are robust with respect to violations of the assumption of white noise.

A step-by-step guide to carrying out the thresholding manually

This section explains how the reconstruction of the image can be done manually using the functions `extract.coefficients`, `estimate.sdev`, `treethresh` / `wttresh`, `prune`, and `thresh`.

Starting with the wavelet transform we have computed at the beginning of this section we first estimate the standard error of the noise. This is easier for wavelets than it is for general sequences, as one can base the estimate on the coefficients at the finest level, which typically do not contain much of the underlying signal. This can be done using the function `estimate.sdev` which can be applied to objects of the classes '`wd`' or '`imwd`'.

```
R> sdev <- estimate.sdev(tiles.noisy.imwd)
```

Our estimate of the standard deviation is 0.9691384, which is not too far from the actual standard deviation of the "salt-and-pepper" noise (0.9752828).

Next, we need to extract the coefficient matrices (or vectors in the case of '`wd`' objects) from the object, so that we can threshold them. Typically one would not threshold the coarser coefficients, by default `extract.coefficients` does not extract the coefficients at the four coarsest levels (i.e., these will not be thresholded).

```
R> tiles.noisy.coefs <- extract.coefficients(tiles.noisy.imwd)
```

Then we rescale these coefficients, so that the noise has (approximately) unit variance.

```
R> for (nm in names(tiles.noisy.coefs)) {
+    tiles.noisy.coefs[[nm]] <- tiles.noisy.coefs[[nm]] / sdev
+  }
```

We are now ready to threshold the coefficients. We will use the *Wavelet TreeThresh* algorithm.[2]

```
R> tiles.noisy.wtt <- wtthresh(tiles.noisy.coefs)
```

Figure 12 (a) shows the estimated partitioning (before having carried out the pruning) together with the corresponding thresholded image. Again it is a little too fine. Panel (b) shows the partitioning after the pruning, which removes two splits towards the middle of the image and one towards the bottom left. To determine the correct amount of pruning we again use cross-validation to estimate the optimal complexity parameter C. This is shown in Figure 11 where we plot the predictive log-likelihood estimated by cross-validation as a function of the complexity parameter C. The predictive log-likelihood is highest for $C = 0.2719$ (corresponding to 16 regions). However, choosing the slightly larger $C = 0.4210$ (corresponding to 15 regions) does not give results that are more than half a standard error worse than the best choice (being above the dotted line). Thus, as explained in Section 1, a partition with 15 regions is retained.

```
R> tiles.noisy.wttp <- prune(tiles.noisy.wtt)
```

Once we have determined the partitioning, we only need to carry out the actual thresholding, rescale the coefficients to their original domain, insert them into the '`imwd`' (or '`wd`' object) and transform the coefficients back to the original domain.

[2]If we wanted to use the *Levelwise TreeThresh* algorithm we would simply threshold each coefficient matrix (or vector) separately as described in Section 4.1 (with the only exception that we would do the rescaling again).

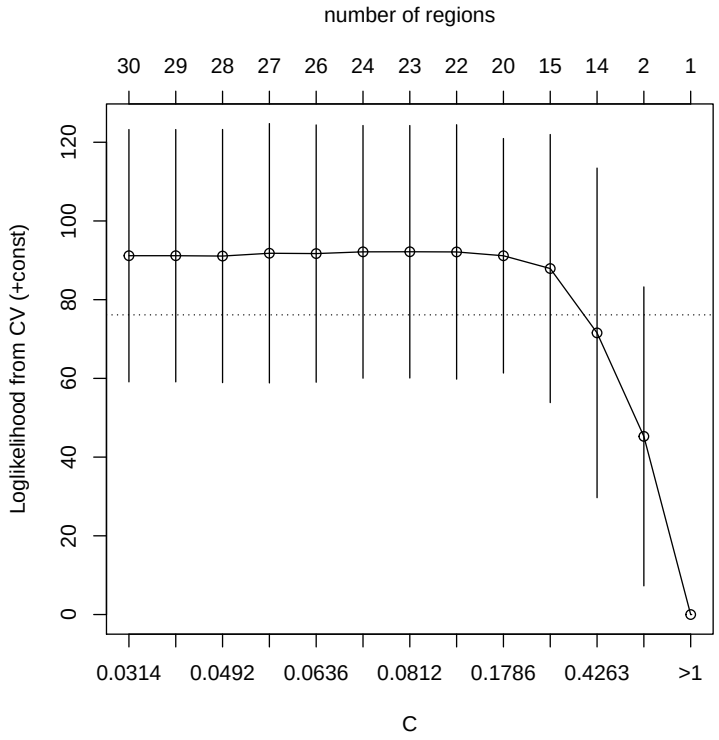


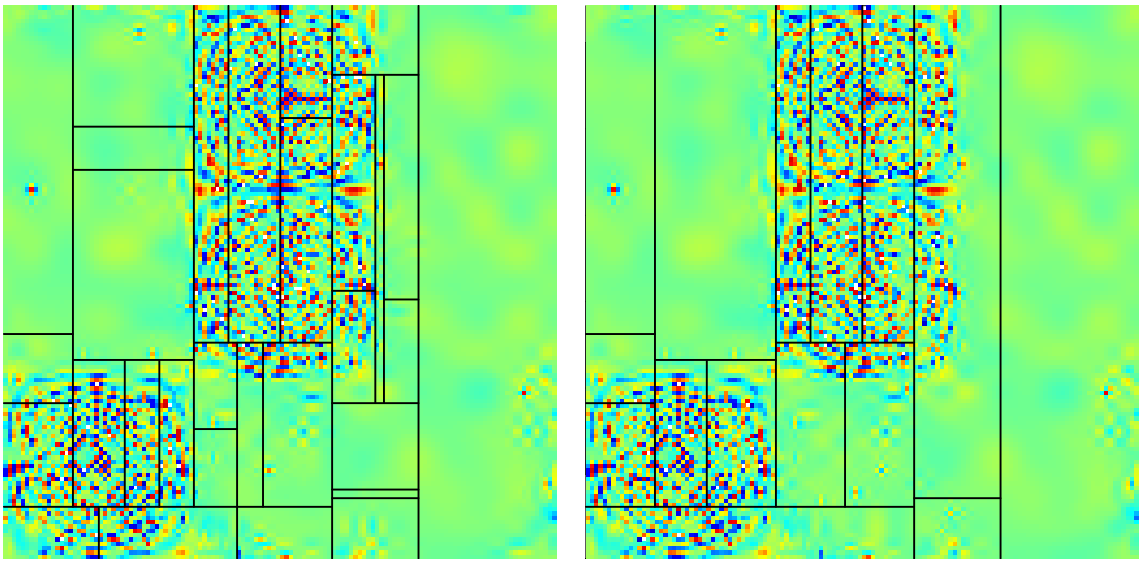
Figure 11: Predictive log-likelihood as a function of the complexity parameter C.

```
R> tiles.noisy.coefs.threshed <- thresh(tiles.noisy.wttp)
R> for (nm in names(tiles.noisy.coefs)) {
+    tiles.noisy.coefs.threshed[[nm]] <-
+      tiles.noisy.coefs.threshed[[nm]] * sdev
+  }
R> tiles.noisy.imwd.threshed <- insert.coefficients(tiles.noisy.imwd,
+    tiles.noisy.coefs.threshed)
R> tiles.noisy.threshed <- imwr(tiles.noisy.imwd.threshed)
```

4.3. A real word example

We conclude our illustration of the software implementation with a final example showing how the method can be applied to denoise a real life image of cells illuminated by laser provided by Dr. Ashley Cadby (Dept. of Physics, University of Sheffield). The noisy image can be seen in Figure 13 and show 3T3 cells where the Actin cytoskeleton of the cells has been labeled with AlexaFluor 647. The image is a single frame from a stochastic optical reconstruction microscopy (STORM) experiment. The sample was illuminated with a high power 640nm laser causing stochastic emission from single molecules. The emission from the single molecules is diffraction limited, with the magnification of the optical system such that a single diffraction limited spot appears over 3×3 pixels.

The raw photograph is provided as a grayscale image file in PNG format and read in using the package **png** (Urbanek 2013).



(a) Before pruning. (b) After pruning.

Figure 12: Estimated partitioning and corresponding reconstructed image (before and after the pruning).

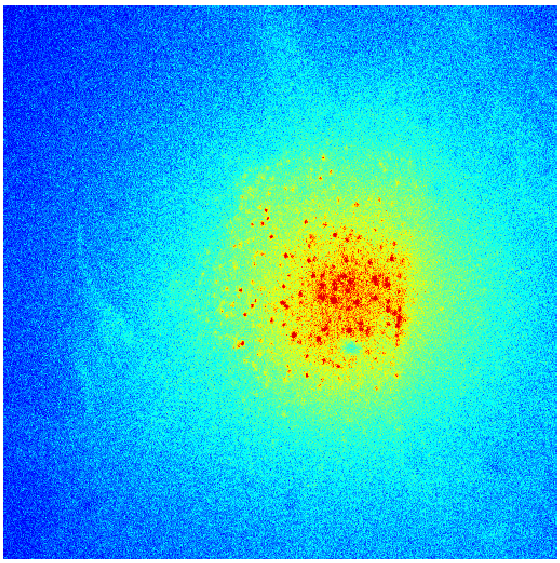
```
R> library("png")
R> img <- readPNG("3T3cells.png")
R> img.imwd <- imwd(img)
R> img.imwd.threshed <- wavelet.treethresh(img.imwd)
R> img.denoised <- imwr(img.imwd.threshed)
R> writePNG(img.denoised, "3T3_denoised.png")
```

Figure 13 presents the original image and the denoised image using both *TreeThresh* and *EbayesThresh*. As can be seen, *TreeThresh* provides a much cleaner reconstruction than *EbayesThresh*. In the center, *TreeThresh* retains significantly more contrast when compared to *EbayesThresh*. Furthermore, in the outer regions where no signal is present, *TreeThresh* provides an estimate with fewer visually distracting artefacts. This illustrates that, due to the capacity of *TreeThresh* to partition images and denoise adaptively, improvement in the estimation of one part of an image need not come at the expense of a poorer reconstruction in other areas.

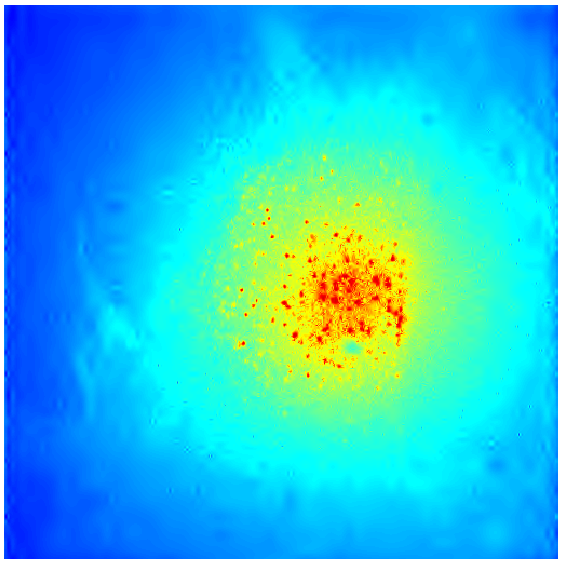
5. Conclusion

In this manuscript we have demonstrated how the **treethresh** package can be used for the thresholding of sequences observed with error and for the denoising of images via the use of a wavelet transformation. The package has been developed in R (R Core Team 2017) and is freely available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=treethresh`.

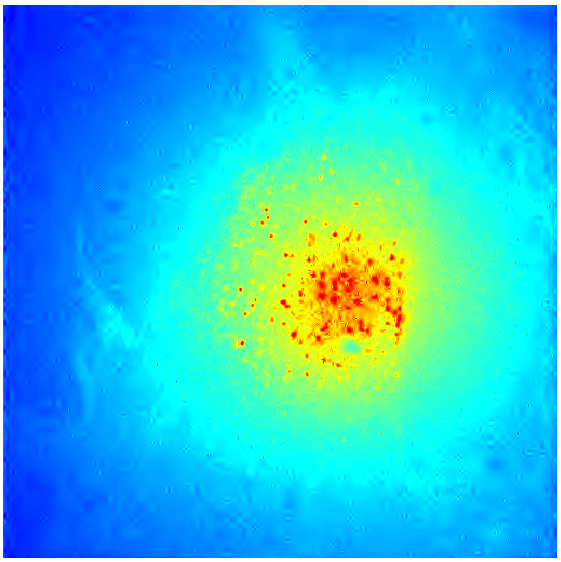
The *TreeThresh* method and the associated software we have introduced often yield lower reconstruction errors than the default *EbayesThresh* method. Furthermore, they create po-



(a) Original image.



(b) Denoised image (*TreeThresh*).



(c) Denoised image (*EbayesThresh*).

Figure 13: STORM image (Panel (a)) and denoised image obtained using *TreeThresh* (Panel (b)) and *EbayesThresh* (Panel (c)).

tential partitions of the sequence or image into regions with differing characteristics according to signal strength. This allows for independent interpretation and may give additional insight into the data at hand.

It should be noted that since only splits parallel to the axes are considered, the method is not invariant under rotations of the input signal. This is of course not an issue for one-dimensional data and in most applications two-dimensional and higher-dimensional signals have a natural orientation, so that the rotational invariance does not seem to be an important desired property of the method. In principle, this could be addressed by modifying the method

to use partitions based on Voronoi tessellations instead of perpendicular splits although these will add computational expense if dimensions extend considerably higher than the two or three we envisage our method being most commonly applied in.

References

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984). *Classification and Regression Trees*. Wadsworth and Brooks/Cole, Monterey.

Evers L, Heaton TJ (2009). "Locally-Adaptive Tree-Based Thresholding." *Journal of Computational and Graphical Statistics*, **18**(4), 961–977. doi:10.1198/jcgs.2009.07109.

Evers L, Heaton TJ (2017). ***treethresh**: Methods for Tree-Based Local Adaptive Thresholding*. R package version 0.1-11, URL https://CRAN.R-project.org/package=treethresh.

Heaton TJ (2009). "Adaptive Thresholding of Sequences with Locally Variable Strength." *Statistics and Computing*, **19**(1), 57–71. doi:10.1007/s11222-008-9071-1.

Johnstone IM, Silverman BW (2004). "Needles and Straw in Haystacks: Empirical Bayes Estimates of Possible Sparse Sequences." *The Annals of Statistics*, **32**(4), 1594–1649. doi:10.1214/009053604000000030.

Johnstone IM, Silverman BW (2005a). "Empirical Bayes Selection of Wavelet Thresholds." *The Annals of Statistics*, **33**(4), 1700–1752. doi:10.1214/009053605000000345.

Johnstone IM, Silverman BW (2005b). "**EbayesThresh**: R Programs for Empirical Bayes Thresholding." *Journal of Statistical Software*, **12**(8), 1–38. doi:10.18637/jss.v012.i08.

Keitt T (2012). ***colorRamps**: Builds Color Tables*. R package version 2.3, URL https://CRAN.R-project.org/package=colorRamps.

Nason G (2010). ***wavethresh**: Wavelets Statistics and Transforms*. R package version 4.5, URL https://CRAN.R-project.org/package=wavethresh.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. Vienna, Austria. R Foundation for Statistical Computing, URL https://www.R-project.org/.

Ripley BD (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge.

Silverman BW (2010). ***EbayesThresh**: Empirical Bayes Thresholding and Related Methods*. R package version 1.3-2, URL https://CRAN.R-project.org/package=EbayesThresh.

Urbanek S (2013). ***png**: Read and Write PNG Images*. R package version 0.1-7, URL https://CRAN.R-project.org/package=png.

Affiliation:

Ludger Evers
School of Mathematics and Statistics
University of Glasgow
Glasgow, G12 8QQ, United Kingdom
E-mail: Ludger.Evers@glasgow.ac.uk
Tim Heaton
School of Mathematics and Statistics
University of Sheffield
Sheffield, S3 7RH, United Kingdom
E-mail: T.Heaton@shef.ac.uk

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
June 2017, Volume 78, Code Snippet 2 *Submitted:* 2012-01-25
doi:10.18637/jss.v078.c02 *Accepted:* 2016-05-27