```
WaveThresh: R wavelet software, release 4.5, installed
Copyright Guy Nason and others 1993-2010
```

Locally adaptive tree-based thresholding using the `treethresh` package in R

Ludger Evers Tim Heaton

August 31, 2012

1 Methodology

TreeThresh (Evers and Heaton, 2009) is a modification of the *EbayesThresh* method (Johnstone and Silverman, 2004, 2005a,b) which tries to partition the underlying signal before carrying out the thresholding. This should yield better results for heterogeneous signals.

1.1 Model

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 generalisation of the *EbayesThresh* method, 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 prior distribution for the μ_i:

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

In contrast to the *EbayesThresh* method the mixing weights w_i depends 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 $\mathcal{I} = P_1 \cup \ldots \cup P_p$, $P_k \cap P_l = \emptyset$, such that the weights within each region P are (almost) constant.

The `treethresh` software uses a double exponential distribution[1] with fixed scale parameter a (set to 0.5 by default) as $\gamma(\cdot)$, which is also the default setting used in the `EbayesThresh` package. This yields

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

as the marginal loglikelihood of the observed $\mathbf{x}$.

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

1.2 Estimation

In order to estimate the mixing weights w_i we need to estimate the partition $\mathcal{P}$ and the mixing weights in each region P. Once an "optimal" partition has been identified, one can estimate the mixing weights in each partition by maximising the loglikelihood of the observations in that region. This corresponds to carrying out the *EbayesThresh* algorithm for region separately.

The partitioning is found using an algorithm that resembles the one used in recursive partitioning algorithms like *Classification and Regression Trees* (CARTs, Breiman et al., 1984). First of all, a nested sequence of increasingly fine partitions is estimated. Cross-validation is then used to find the "best" partition of that sequence. Section 2 gives a more detailed description of the algorithm.

1.3 Thresholding

Having estimated the mixing weights w_i, we can use these to estimate the underlying signal μ_i. This can be done in many ways:

Posterior median The posterior median of μ_i given $X_i = x$ is shown in figure 1 as a function of x (solid line). It has a thresholding property: 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, sec. 6.1).

Hard thresholding Alternatively, we could use the $t_{\hat{w}_i}$ obtained from the posterior median to define the hard thresholding rule

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

(dashed line in figure 1). The hard thresholding rule is discontinuous at $-t_{\hat{w}_i}$ and at $t_{\hat{w}_i}$.

Soft thresholding The soft thresholding rule

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

(dotted line in figure 1) is continuous, but is biased even for large values of x.

By default, the `treethresh` software 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

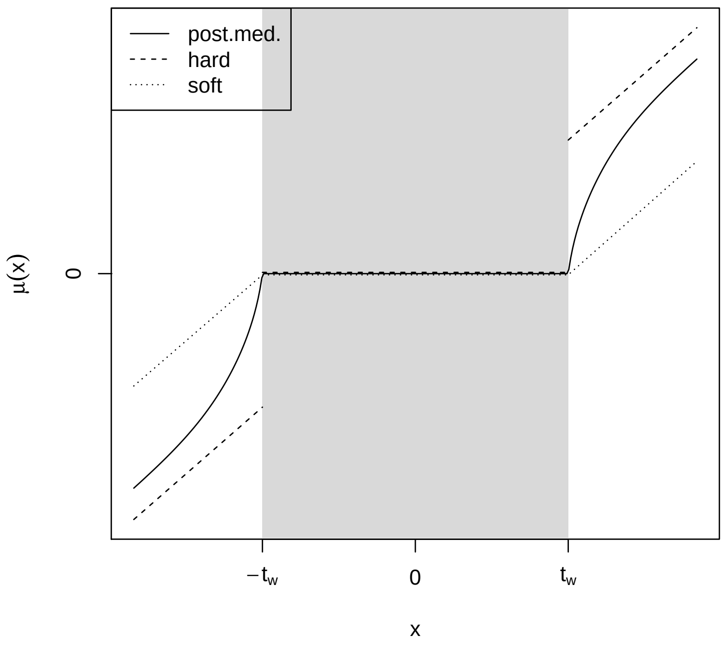


Figure 1: Comparison of the three thresholding rules.

L and R. As there is 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 in which the signal is hopefully more heterogeneous. 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` (see section 3 for 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

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

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

Just as in the case of CARTs, one can show (see e.g. Ripley, 1996, sec. 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 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).

As such, one would choose the value of C that yields the largest predictive loglikelihood. However, it turns out to be often better to use a simpler model (corresponding to a larger value of C) if the corresponding predictive loglikelihood 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 loglikelihood is less than half the standard error of the best predictive loglikelihood.

This second step of the algorithm can be carried out by calling the function `prune`.

For a more detailed description of the algorithm together with its asymptotic properties see Evers and Heaton (2009). Sections 4.1 and 4.2 contain two examples illustrating these two steps of the algorithm.

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 the process consists of transforming the noisy signal to the wavelet domain where it is expected that the underlying signal has a sparse representation. The observed wavelet coefficients are thus thresholded before being

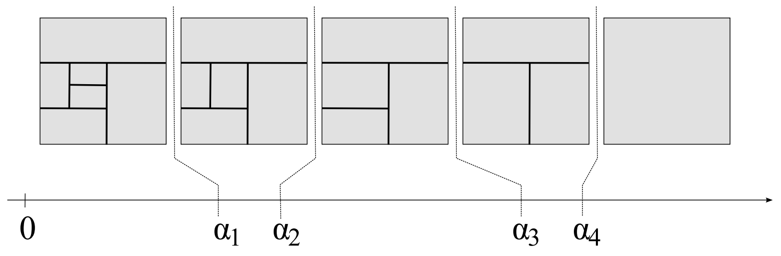


Figure 2: Example of a nested sequence of partitions corresponding to different values of α. As α increases, the optimal penalised likelihood partition becomes coarser and is nested within the optimal partition for smaller values of α.

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 simply 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 illustrates how the partition of the original domain is transferred to the wavelet coefficients.

Our code provides the possibility to apply both types of partitioning algorithm. *Levelwise TreeThresh* simply applies the partitioning algorithm explained in 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 the *TreeThresh* algorithm see section 4.2.

4 Using the software

4.1 Thresholding sequences

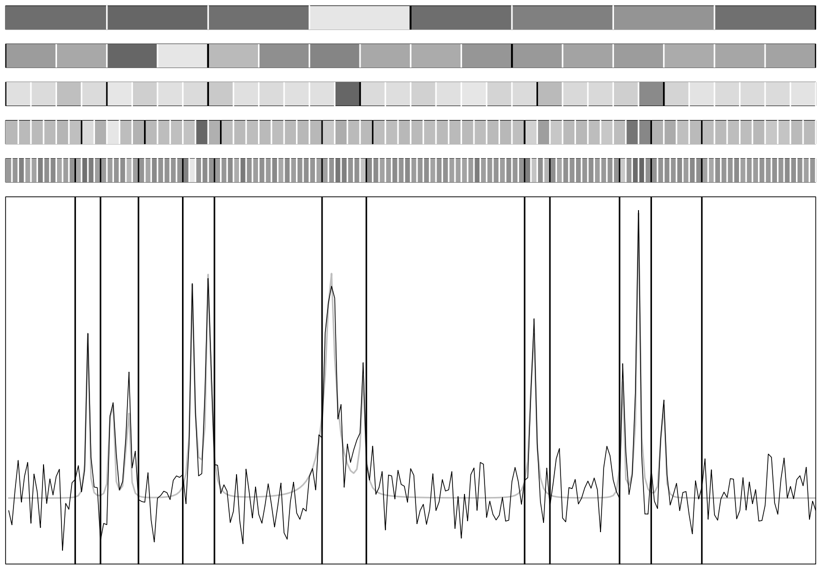
This section uses a simple example (which is very similar to the one given in the help file of `treethresh`) to illustrates how the `treethresh` package can be used to threshold a simple sequence.

First of all we start with creating a sparse signal, which is rather dense towards the middle and sparse at the beginning and at the end.

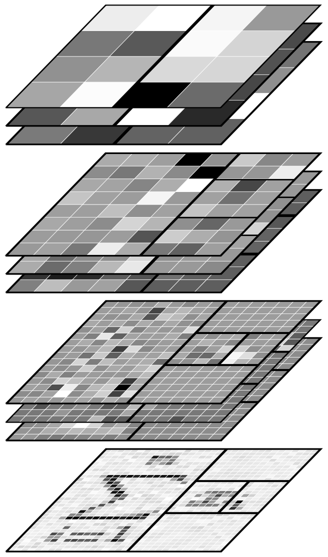
We start with creating a vector that contains the probabilities w_i that $\mu_i \neq 0$.

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

Next we create the signal $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_{1000})$ by drawing the non-zero μ_i from a Laplace distribution. Figure 4(a) displays the simulated true signal.



(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 in transferred to the each level of the wavelet coefficients.

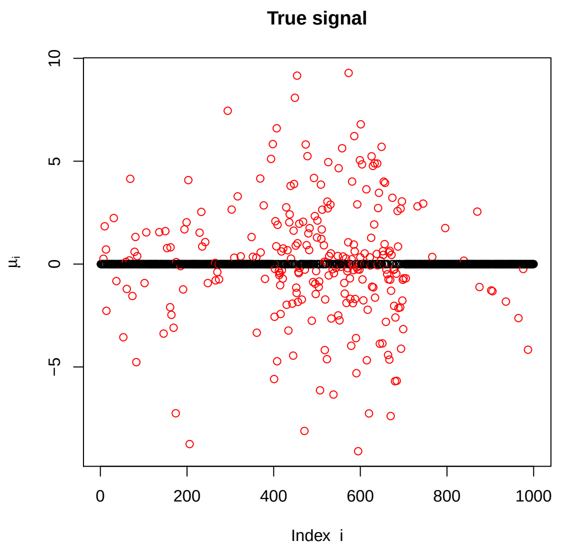
```
> mu <- numeric(length(w.true))
> non.zero.entry <- runif(length(mu))<w.true
> num.non.zero.entries <- sum(non.zero.entry)
> mu[non.zero.entry] <- rexp(num.non.zero.entries,rate=0.5) *
+                          sample(c(-1,1),num.non.zero.entries,replace=TRUE)
> mu[1:14]

 [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
[13]  0.7091753 -2.2708853
```

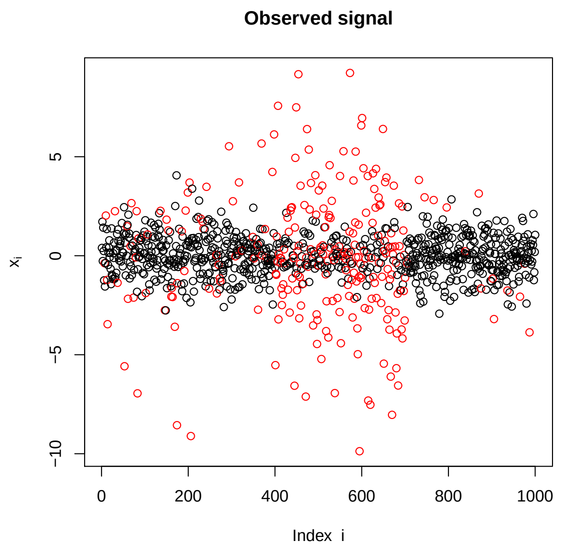
Next we create the observed noisy signal $\mathbf{x} = (x_1, \ldots, x_{1000})$ by adding white noise to $\boldsymbol{\mu}$. Figure 4(b) displays the simulated "observed" signal.

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

In our example we know that the noise has unit variance. However, in most practical settings this would not necessarily be the case. Estimating the standard error a priori is difficult. The medium absolute deviation as used in the function `mad` can be used to get a rough idea of the standard error of the noise. The correction factor of 1.4826 used by `mad` however 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



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



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

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

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

the correction factors one could use (instead of 1.4826) for a homogeneous signal if the w_i were constant and known (which 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:

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

```
[1] 0.9973816
```

Next, we rescale the signal using our estimate `sdev`:

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

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

```
> x.tt <- treethresh(x)
```

The element `splits` contains detailed information about the partition. 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` / `left.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 loglikelihood (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.

```
> x.tt$splits

      id parent.id dim pos left.child.id right.child.id      crit           w
 [1,]  1        NA   0 745             2             63 51.794514 0.291419714
 [2,]  2        -1   0 393             3             32 52.354525 0.369894137
 [3,]  3        -2   0 369             4             31 19.843343 0.139722902
 [4,]  4        -3   0   9             5              6  4.821266 0.154346257
 [5,]  5        -4  NA  NA            NA             NA        NA 0.008961814
 [6,]  6         4   0 145             7             14  2.741212 0.158658582
 [7,]  7        -6   0  83             8             13 21.499363 0.086433684
 [8,]  8        -7   0  51             9             12  4.202844 0.176956791
 [9,]  9        -8   0  14            10             11 18.933715 0.050922031
[10,] 10        -9  NA  NA            NA             NA        NA 0.918786110
[11,] 11         9  NA  NA            NA             NA        NA 0.008961814
[12,] 12         8  NA  NA            NA             NA        NA 0.403881292
[13,] 13         7  NA  NA            NA             NA        NA 0.008961814
[14,] 14         6   0 209            15             24  7.217389 0.201446984
[15,] 15       -14   0 202            16             23  2.755131 0.444045959
[16,] 16       -15   0 174            17             20  4.602292 0.333423913
[17,] 17       -16   0 168            18             19  3.978556 0.575379645
[18,] 18       -17  NA  NA            NA             NA        NA 0.239113084
[19,] 19        17  NA  NA            NA             NA        NA 1.000000000
```

```
[20,] 20         16   0 193             21            22 14.132629 0.085758632
[21,] 21        -20  NA  NA             NA            NA        NA 0.008961814
[22,] 22         20  NA  NA             NA            NA        NA 0.450094843
[23,] 23         15  NA  NA             NA            NA        NA 1.000000000
[24,] 24         14   0 222             25            26  6.292213 0.103932956
[25,] 25        -24  NA  NA             NA            NA        NA 0.008961814
[26,] 26         24   0 360             27            30  1.403153 0.115532264
[27,] 27        -26   0 318             28            29 21.033108 0.086737152
[28,] 28        -27  NA  NA             NA            NA        NA 0.160374716
[29,] 29         27  NA  NA             NA            NA        NA 0.008961814
[30,] 30         26  NA  NA             NA            NA        NA 0.520767184
[31,] 31          3  NA  NA             NA            NA        NA 0.008961814
[32,] 32          2   0 699             33            58 13.165712 0.593491441
[33,] 33        -32   0 578             34            49  6.490249 0.656840656
[34,] 34        -33   0 558             35            46  6.313493 0.534220715
[35,] 35        -34   0 398             36            37  1.477147 0.583540235
[36,] 36        -35  NA  NA             NA            NA        NA 1.000000000
[37,] 37         35   0 433             38            41  3.805415 0.562061951
[38,] 38        -37   0 408             39            40  4.487796 0.283618576
[39,] 39        -38  NA  NA             NA            NA        NA 0.620281112
[40,] 40         38  NA  NA             NA            NA        NA 0.008961814
[41,] 41         37   0 526             42            43  1.689291 0.632060627
[42,] 42        -41  NA  NA             NA            NA        NA 0.701163056
[43,] 43         41   0 548             44            45  2.529591 0.444711221
[44,] 44        -43  NA  NA             NA            NA        NA 0.235895281
[45,] 45         43  NA  NA             NA            NA        NA 0.811099320
[46,] 46         34   0 570             47            48 10.629355 0.132429216
[47,] 47        -46  NA  NA             NA            NA        NA 0.008961814
[48,] 48         46  NA  NA             NA            NA        NA 0.383637108
[49,] 49         33   0 691             50            57  1.769788 0.856418890
[50,] 50        -49   0 686             51            56  1.924944 0.821869635
[51,] 51        -50   0 679             52            55  1.924676 0.846940694
[52,] 52        -51   0 591             53            54  2.009814 0.805784166
[53,] 53        -52  NA  NA             NA            NA        NA 1.000000000
[54,] 54         52  NA  NA             NA            NA        NA 0.747009644
[55,] 55         51  NA  NA             NA            NA        NA 1.000000000
[56,] 56         50  NA  NA             NA            NA        NA 0.008961814
[57,] 57         49  NA  NA             NA            NA        NA 1.000000000
[58,] 58         32   0 731             59            60 16.093096 0.135461573
[59,] 59        -58  NA  NA             NA            NA        NA 0.008961814
[60,] 60         58   0 736             61            62  1.889118 0.783767974
[61,] 61        -60  NA  NA             NA            NA        NA 1.000000000
[62,] 62         60  NA  NA             NA            NA        NA 0.456043602
[63,] 63          1  NA  NA             NA            NA        NA 0.033008570
                 t loglikelihood      alpha          C
 [1,] 2.194507e+00 448.225319227         NA         NA
 [2,] 1.996889e+00 463.494532535 19.9573109 1.00000000
 [3,] 2.650636e+00  66.098749944 19.9573109 1.00000000
 [4,] 2.597540e+00  67.704496409  1.9817559 0.09929975
 [5,] 3.716922e+00  -0.024799078  1.9817559 0.09929975
 [6,] 2.582479e+00  68.157410623  1.9817559 0.09929975
 [7,] 2.883452e+00  17.246465573  1.9817559 0.09929975
 [8,] 2.521125e+00  19.601810451  1.9817559 0.09929975
 [9,] 3.107908e+00   0.562496789  1.9817559 0.09929975
[10,] 3.074389e-01   3.382137134  1.9817559 0.09929975
[11,] 3.716922e+00  -0.121345896  1.9817559 0.09929975
[12,] 1.913253e+00  20.864824894  1.9817559 0.09929975
[13,] 3.716922e+00  -0.162716706  1.9817559 0.09929975
[14,] 2.444340e+00  52.060227762  1.9817559 0.09929975
[15,] 1.814353e+00  36.212726074  1.9817559 0.09929975
[16,] 2.087522e+00  19.043980745  1.9817559 0.09929975
[17,] 1.477179e+00  20.552128104  1.9817559 0.09929975
[18,] 2.334947e+00   0.906651079  1.9817559 0.09929975
[19,] 4.656613e-09  21.800441126  1.9817559 0.09929975
[20,] 2.886988e+00   0.465277215  1.9817559 0.09929975
[21,] 3.716922e+00  -0.064435955  1.4789267 0.07410451
[22,] 1.799384e+00   2.008639895  1.4789267 0.07410451
[23,] 4.656613e-09  19.089805391  1.9817559 0.09929975
[24,] 2.798074e+00  19.810361342  1.9817559 0.09929975
[25,] 3.716922e+00  -0.038290657  1.2531460 0.06279132
[26,] 2.746972e+00  20.296258081  1.2531460 0.06279132
[27,] 2.881868e+00  10.820791673  1.2531460 0.06279132
[28,] 2.576553e+00  12.913908687  1.2531460 0.06279132
[29,] 3.716922e+00  -0.151214788  1.2531460 0.06279132
[30,] 1.621294e+00  10.845396079  1.2531460 0.06279132
[31,] 3.716922e+00  -0.094327569  1.9817559 0.09929975
[32,] 1.427620e+00 420.521353677 19.9573109 1.00000000
[33,] 1.245454e+00 423.344045805  5.5900262 0.28009917
[34,] 1.586444e+00 206.568628409  3.3452366 0.16761960
[35,] 1.454972e+00 200.361348969  2.3299279 0.11674558
[36,] 4.656613e-09  18.321250908  1.5539855 0.07786548
[37,] 1.513012e+00 182.942570321  1.5539855 0.07786548
[38,] 2.214841e+00  20.311702162  1.5539855 0.07786548
[39,] 1.352379e+00  22.414088081  1.5539855 0.07786548
[40,] 3.716922e+00  -0.020264070  1.5539855 0.07786548
[41,] 1.318493e+00 164.308230660  1.5539855 0.07786548
[42,] 1.108258e+00 134.137444158  1.0828199 0.05425681
[43,] 1.812708e+00  30.973234531  1.0828199 0.05425681
[44,] 2.343965e+00   9.707221576  1.0828199 0.05425681
[45,] 7.266783e-01  22.629204792  1.0828199 0.05425681
[46,] 2.678441e+00   8.537207297  2.3299279 0.11674558
[47,] 3.716922e+00  -0.046508986  1.1502850 0.05763728
```

```
[48,] 1.963023e+00   9.734001310  1.1502850 0.05763728
[49,] 5.532468e-01 220.120653951  3.3452366 0.16761960
[50,] 6.861616e-01 205.410488114  0.8402070 0.04210021
[51,] 5.901004e-01 206.648162934  0.8402070 0.04210021
[52,] 7.464881e-01 177.582960433  0.6522152 0.03268052
[53,] 4.656613e-09  32.280795173  0.6522152 0.03268052
[54,] 9.564945e-01 146.085835615  0.6522152 0.03268052
[55,] 4.656613e-09  29.585962588  0.6522152 0.03268052
[56,] 3.716922e+00  -0.008273762  0.8402070 0.04210021
[57,] 4.656613e-09  15.161178727  0.8402070 0.04210021
[58,] 2.666764e+00   2.767334024  5.5900262 0.28009917
[59,] 3.716922e+00  -0.105419332  4.5238544 0.22667655
[60,] 8.271500e-01   7.396607787  4.5238544 0.22667655
[61,] 4.656613e-09   6.643391846  0.7137060 0.03576163
[62,] 1.784635e+00   1.466921907  0.7137060 0.03576163
[63,] 3.274768e+00   1.519837484 19.9573109 1.00000000
```

Figure 6 shows the estimated partion and the estimated weights w_i both before and after the pruning.

As mentioned in section 2 and as one can see from figure 6(a), the partition estimated in this first step constitutes an overfit to the data. Thus we need to carry out a second pruning step that reduces the complexity of the estimated partition.

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

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

Figure 5 shows how the “optimal” value of the complexity parameter C was determined. By default `prune` uses five-fold cross validation (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.

Now that we have found the optimal partition, we can start using the estimated weights 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.

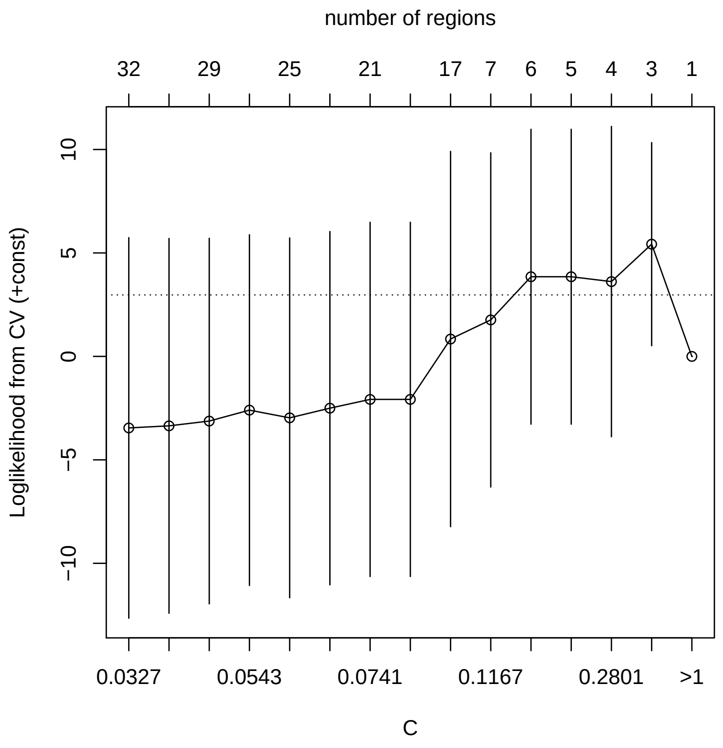
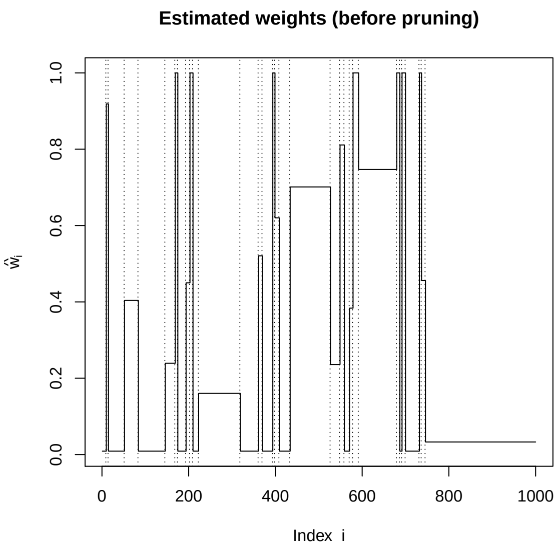
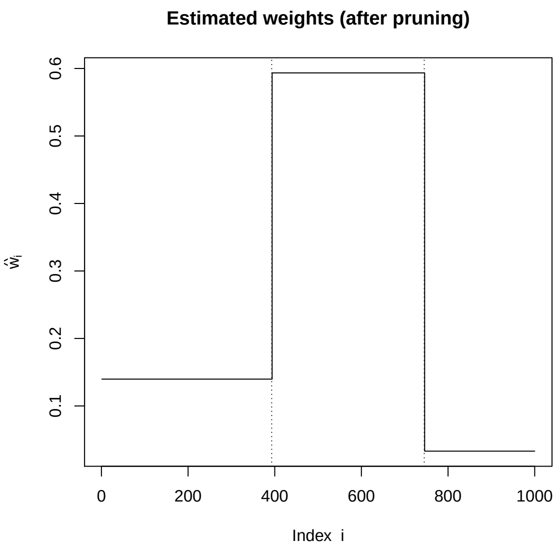


Figure 5: Predictive loglikelihood as a function of the complexity parameter C.



(a) before pruning



(b) after pruning

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

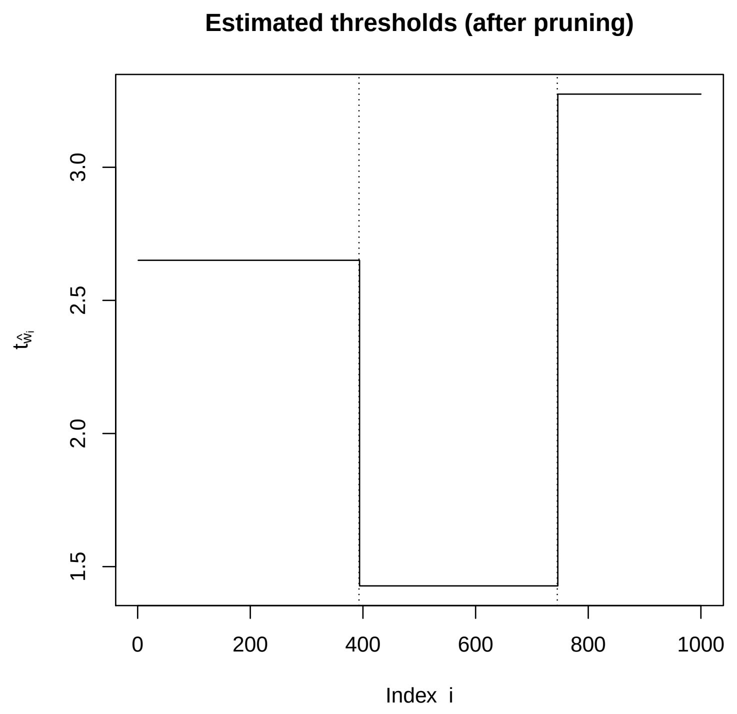


Figure 7: Estimated thresholds $t_{\hat{w}_i}$ of the partition after pruning.

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

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

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

Figure 8 shows the reconstructed sequence.

4.2 Thresholding wavelet coefficients

4.2.1 Preparing the example

This example uses the image `tiles`, shown in figure 9(a)

```
> data(tiles)
```

In the next step we will add noise to the image to see whether we can remove this noise using the *TreeThresh* algorithm.

```
> tiles.noisy <- tiles + 0.8 * rnorm(length(tiles))
```

Figure 9(b) shows the noisy image. 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, 1998).

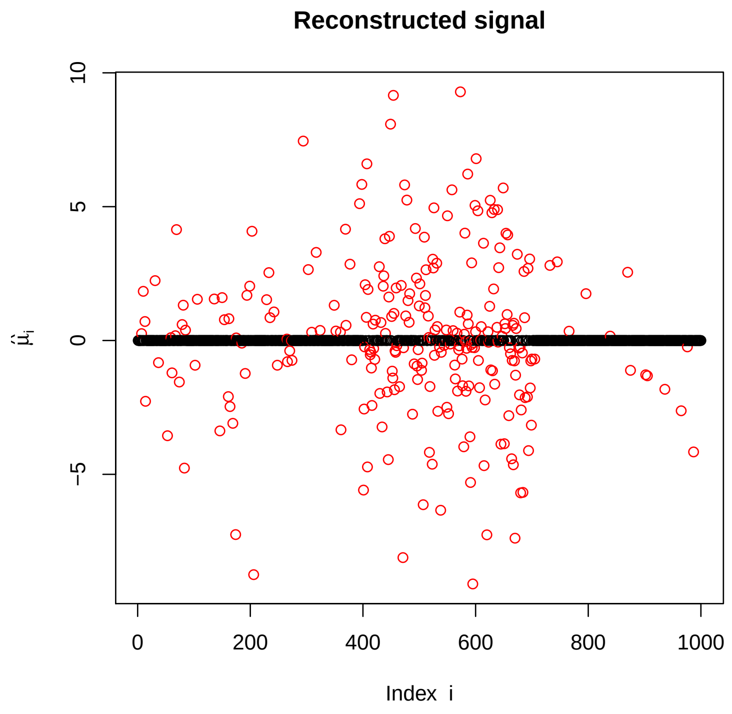


Figure 8: Reconstructed signal $\hat{\mu}_i$ in the original scale (entries corresponding to non-zero true signal μ_i in red).

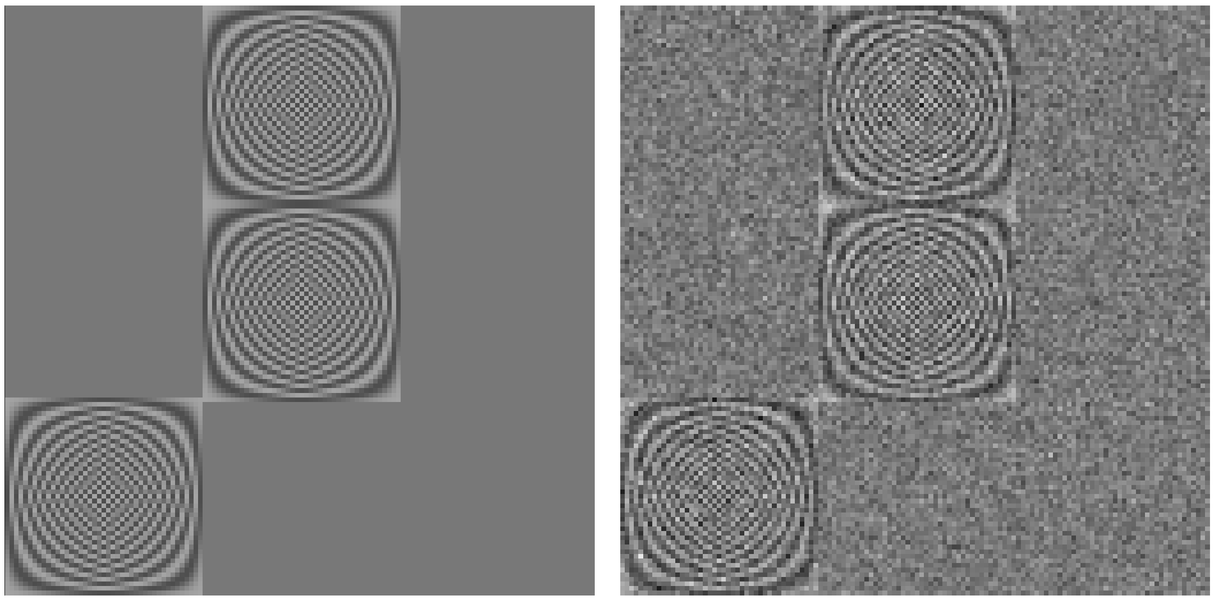


Figure 9: Image `tiles` (panel (a)) and image with white noise added (panel (b)).

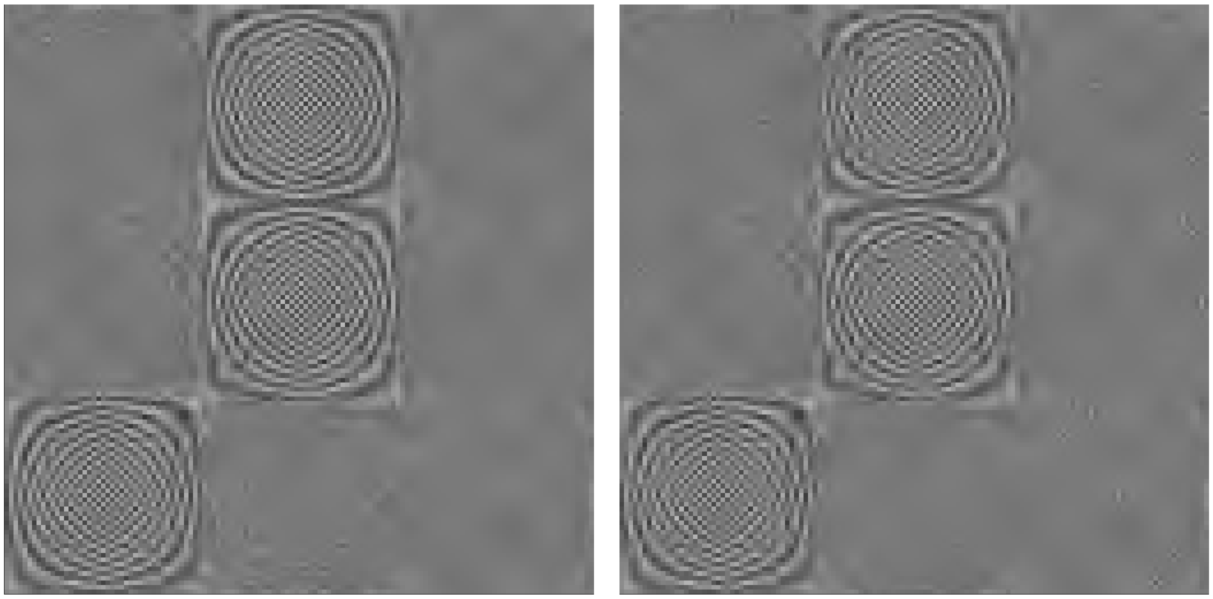

(a) *TreeThresh* (b) *EbayesThresh*

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

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

4.2.2 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` as well as rescaling the coefficients so that the noise has approximately unit variance. This subsection explains how to use this more user-friendly interface, see subsection 4.2.3 for the commands required to carry out the thresholding step by step.

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

To use the *Levelwise TreeThresh* algorithm simply add an additional argument `levelwise=TRUE`.

After having thresholded the wavelet coefficients, we transform them back to the original domain using the function `imwr` from the package `wavethresh`.

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

Figure 10(a) shows the reconstructed image and compares it to the result obtained by *EbayesThresh* (panel (b)). The corresponding l_2 loss is 2829.028 for the *TreeThresh* algorithm and 4282.252 for the *EbayesThresh* algorithm.

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

This subsection 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 in section 4.2.1 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`.

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

Our estimate of the standard error is 0.903319, which is not too far from the true value of 0.8 which we used when we added the noise.

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).

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

Next we need to rescale the coefficients, so that the noise has (approximately) unit variance.

```
> 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]

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

Figure 12 (a) shows the estimated partitioning together with the corresponding thresholded image (before having carried out the pruning). Panel (b) shows the partitioning after the pruning, which removes two splits towards the middle of the image and one towards the bottom left. Figure 11 shows how the optimal complexity parameter C was estimated: it shows the predictive loglikelihood estimated by cross-validation as a function of the complexity parameter C. The predictive log-likelihood is highest for $C = 0.0130$ (corresponding to 14 regions). However, choosing the slightly larger $C = 0.2049$ (corresponding to 13 regions) does not give results that are more than half a standard error worse than the best choice (being above the dotted line). Thus a partition with 13 regions is retained.

```
> 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.

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

[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).

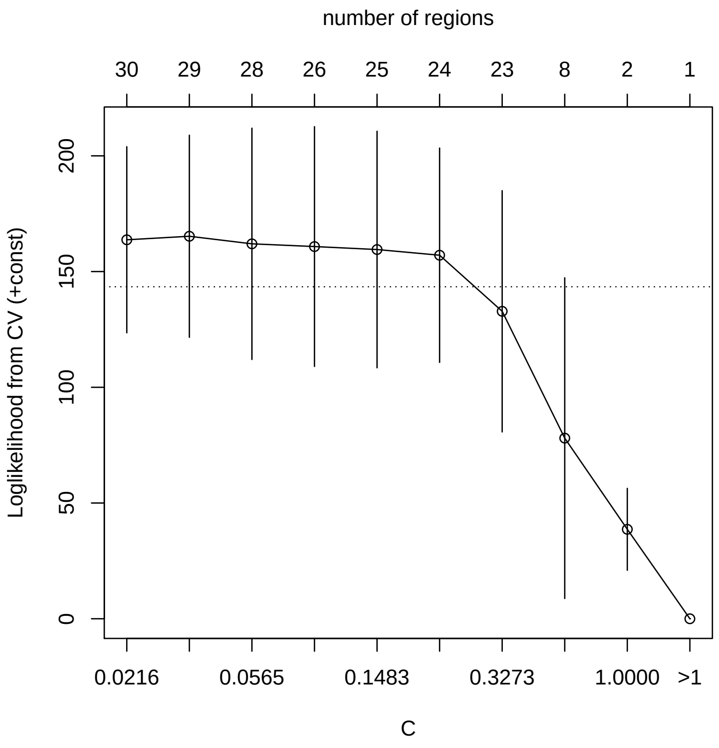


Figure 11: Predictive loglikelihood as a function of the complexity parameter C.

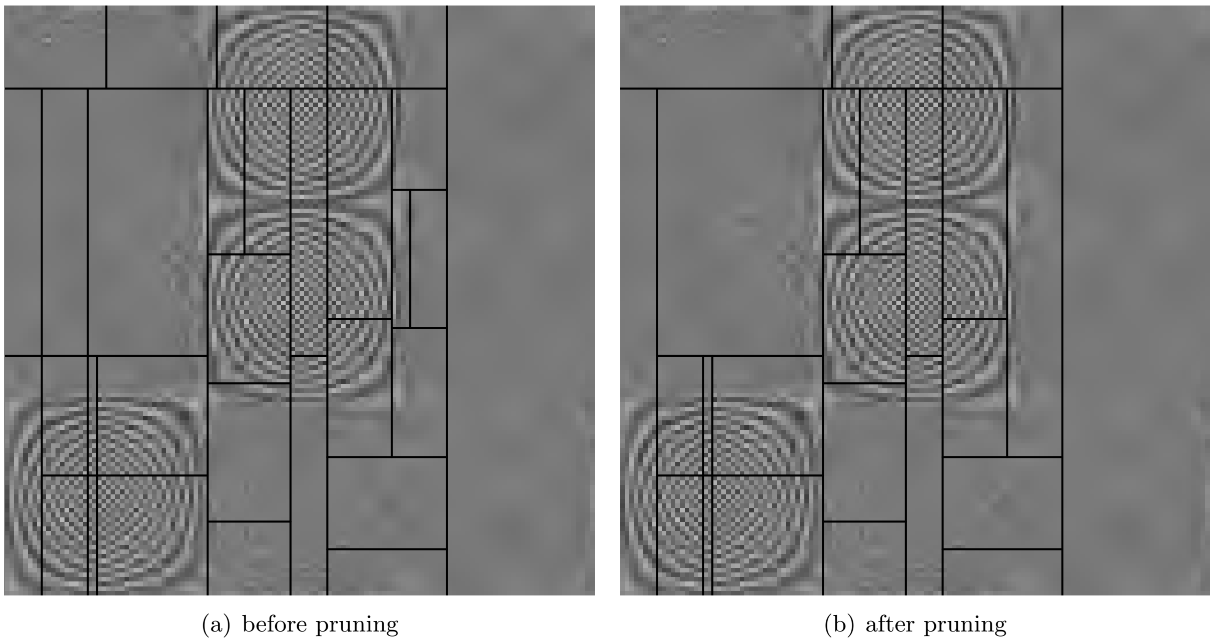


Figure 12: Estimated partitioning (before and after the pruning) and corresponding reconstructed image.

References

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984), *Classification and Regression Trees*, Monterey, CA: Wadsworth and Brooks/Cole.

Evers, L. and Heaton T. J. (2009), "Locally-adaptive tree-based thresholding," *Journal of Compuational and Graphical Statistics*, to appear.

Johnstone, I. M. and Silverman, B. W. (2004), "Needles and straw in haystacks: Empirical Bayes estimates of possible sparse sequences," *Annals of Statistics*, 32, 1594–1649.

— (2005a), "Empirical Bayes selection of wavelet thresholds," *Annals of Statistics*, 33, 1700–1752.

— (2005b), "EbayesThresh: R Programs for Empirical Bayes Thresholding," *Journal of Statistical Software*, 12 (8).

Nason, G. P. (1998), *WaveThresh3 Software*, University of Bristol.

Ripley, B. D. (1996), *Pattern recognition and neural networks*, Cambridge: Cambridge University Press.