RJaCGH 1.5.5: A package for the analysis of CGH arrays through Reversible Jump MCMC.

Oscar M. Rueda[1] and Ramón Díaz-Uriarte[1]

August 22, 2008

1. Statistical Computing Team. Structural Computational Biology Group. Spanish National Cancer Center (CNIO), Madrid (SPAIN). omrueda@cnio.es, rdiaz@ligarto.org

Contents

1 Overview:

RJaCGH is an R package designed for the analysis of microarray CGH data. In this type of problems we have a collection of log-ratios that measure the ratio between the copy number of sequences of nucleotides between a test sample and a control sample for a number of probes. The main goal of the analysis is to detect which of those probes have a normal copy number, a loss copy number or a gained copy number.
This package basically fits a Non Homogeneous Hidden Markov Model through Reversible Jump Markov Chain Montecarlo. That is, we assume that there are k different groups (hidden states; different copy number ratios) within the data. Each of those groups follows a normal distribution with parameters μ_k and σ_k^2. The movements between those hidden states follow a Markov process whose transition probabilities depend on the distance between probes. The estimation of the parameters is made through a Markov Chain Monte Carlo (MCMC) algorithm. These techniques are based on the exploration of the parameter space through sampling. Instead of fitting several models and selecting just one, RJaCGH uses reversible jump [3] to jump between models and get the posterior probability for each of them. We can make birth/death moves (create

or delete a hidden state) and split/combine moves (separate or merge existing states). The inferences are then based on all models visited through Bayesian Model Averaging.

The package estimates the probability for every probe to have a normal copy number, gained or lost and computes probabilistic common regions. This vignette shows some of the package features with small examples. The references give full details about the statistical model and the parameterization it uses, plus further details of the algorithm.

Please note that our methods are computer intensive, so they may take a long time on a slow machine.

2 Data:

We use for the examples the public data set of Snijders et al. [5] with 15 human cells with known karyotypes, as found in the objects from package GLAD 1.6.0. [6].

3 Examples:

3.1 Same model for the whole genome

We will analyze data cell gm13330 from [5]. First, we take out the missing values, because RJaCGH does not handle NA's. We are going to use the log-2 ratios, the positions and the chromosome number:

```
> set.seed(1)
> library(RJaCGH)
> data(snijders)
> y <- gm13330$LogRatio[!is.na(gm13330$LogRatio)]
> Pos <- gm13330$PosBase[!is.na(gm13330$LogRatio)]
> Chrom <- gm13330$Chromosome[!is.na(gm13330$LogRatio)]
```

Now, we are going to fit the model through the function `RJaCGH()`. But first we must decide if we want to fit a model with equal variances for all the hidden states or with different variances. This can be set with the argument `var.equal=TRUE` (default) or `var.equal=FALSE` in the call to `RJaCGH()`. Besides, we can fit the same model to the whole genome or a different one for each chromosome. We can set this option with `model="genome"` or `model="Chrom"` in the call to `RJaCGH()`. In this section we will fit the same model for the whole genome.

We can also set the maximum number of hidden states that we want to fit. For example, we will fit HMMs with a maximum of four hidden states, so we'll set the parameter `k.max=4`.
Besides, we can set, if we wish to, the jumping parameters of the MCMC. They control the exploration of the probability distribution of the model via setting the jumps we make from a particular value of the parameters to a new one. There are two types of them:

- The standard deviation of the candidates of the jumps of the chain within a given model: `sigma.tau.mu`, `sigma.tau.sigma.2` and `sigma.tau.beta`.

They are vectors of length `k.max`. They are related to the dispersion within models.

- The standard deviation of the jumps between models in split/combine moves: `tau.split.mu`. it is a scalar and is related to the dispersion between models.

We must remember that these are not parameters of the model, in the sense that different values produce different models. They are parameters of the algorithm that speed up or assure convergence.

We have to enclose them in a list. By some inspection of the data and/or trial/error we set them to the following values:

```
> jump.parameters <- list(sigma.tau.mu = rep(0.01, 4), sigma.tau.sigma.2 = rep(0.05,
+     4), sigma.tau.beta = rep(0.1, 4), tau.split.mu = 0.1)
```

The arguments `burnin` and `TOT` control the number of iterations of the algorithm (the burn-in and the after burn-in).

`NC` and `deltaT` are arguments related to the number of coupled parallel chains; they will be explained in the last section.

We can also pass other arguments, such as the starting base and end base of the probes (`Start`, `End`), the distance between probes (`Dist`), the names of the probes (`probe.names`), the maximal distance between probes beyond which we consider them independent (`max.dist`)... See the help file for `RJaCGH()` for full reference.

```
> fit <- RJaCGH(y = y, Pos = Pos, Chrom = Chrom, model = "genome",
+     var.equal = TRUE, k.max = 4, burnin = 50000, TOT = 10000,
+     jump.parameters = jump.parameters, NC = 2, deltaT = 0.5)

    Starting Reversible Jump
       Start burn-in
       End burn-in
```

After the fit (it may take a little while), `RJaCGH()` returns an object with several interesting components. Its structure is a list with several lists nested inside of it; one for each model fitted. For example,

```
> fit[[4]]
```

is another list with the results of the fit of a model with 4 hidden states. There are several elements inside; for example, we can get the means and variances of the hidden states fitted:

```
> fit[[4]]$mu
> fit[[4]]$sigma.2
```

They are matrices with as many rows as samples have been drawn from a model with 4 hidden states and as many columns as hidden states (that is, four).

```
> apply(fit[[4]]$mu, 2, mean)

[1] -0.83915385 -0.07770743  0.03548133  0.52557360
```

This would be the mean of the posterior distribution of the means of the 4 hidden states.

In the case of the functions of transition probabilities:

```
> fit[[4]]$beta
```

is an array with the first and second dimensions the number of hidden states and the third the number of MCMC iterations in that model. So

```
> apply(fit[[4]]$beta, c(1, 2), mean)
```

```
         [,1]     [,2]     [,3]     [,4]
[1,] 0.000000 3.013907 2.942078 2.904455
[2,] 6.609204 0.000000 2.594473 7.364862
[3,] 8.114805 3.114729 0.000000 5.848761
[4,] 4.261500 4.292253 2.755572 0.000000
```

would give the mean of the posterior distribution of `beta` (these are parameters of the transition matrix that depends itself on the distance between genes; see references for details on the model).

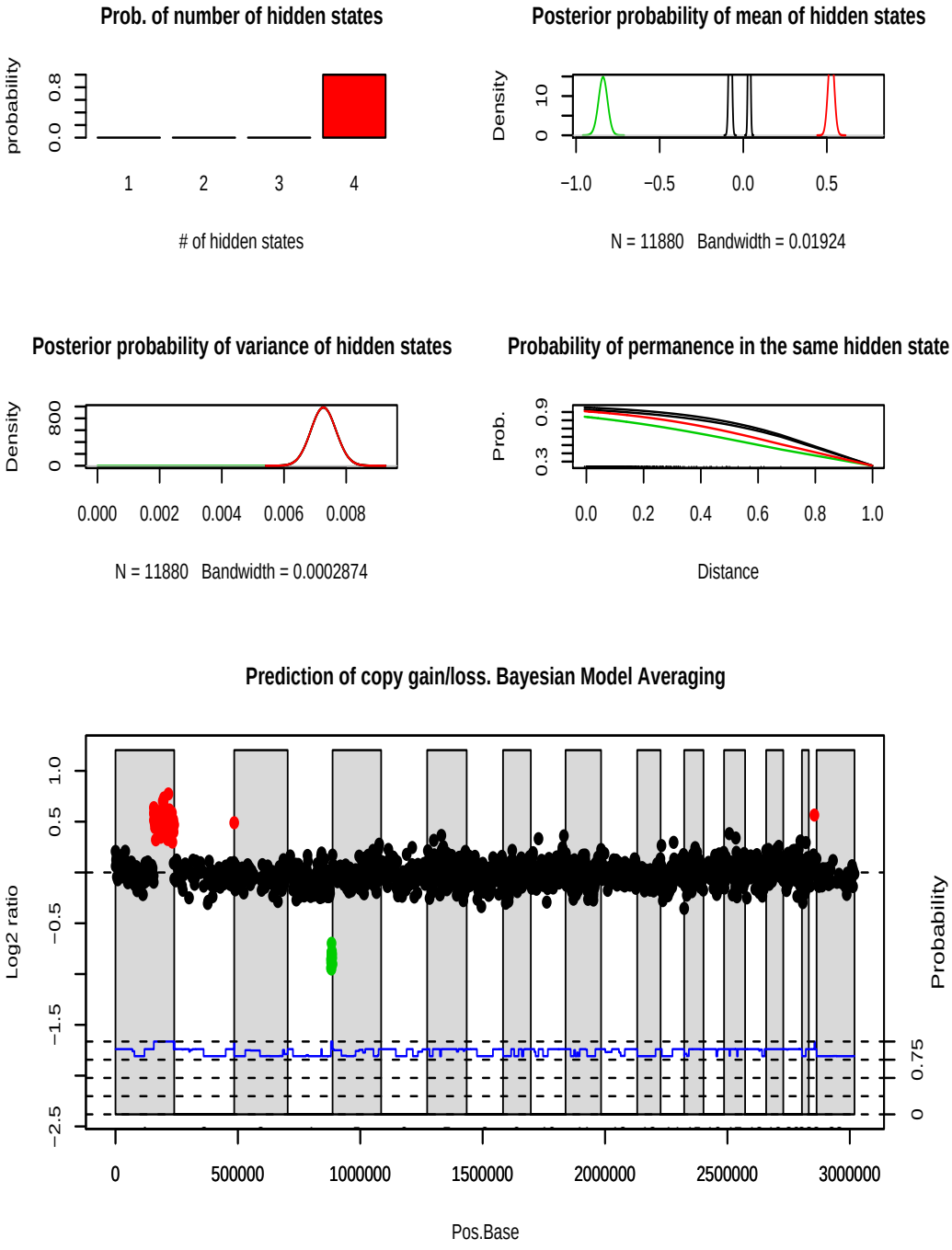
We can also summarize the fit and inspect these results. By default, `summary` returns the quantiles of the posterior distributions for the means and variances and the median of the parameters for the transition probabilities:

```
> summary.HMM <- summary(fit)
> summary.HMM

Distribution of the number of hidden states:

1 2 3 4
0 0 0 1

Model with  4  states:

Distribution of the posterior means of hidden states:
            10%    25%    50%    75%    90%
Loss     -0.863 -0.851 -0.839 -0.827 -0.815
Normal-1 -0.085 -0.081 -0.077 -0.074 -0.071
Normal-2  0.030  0.033  0.036  0.038  0.041
Gain      0.509  0.517  0.526  0.534  0.541

Distribution of the posterior variances of hidden states:
           10%   25%   50%   75%   90%
Loss     0.007 0.007 0.007 0.007 0.008
Normal-1 0.007 0.007 0.007 0.007 0.008
Normal-2 0.007 0.007 0.007 0.007 0.008
Gain     0.007 0.007 0.007 0.007 0.008

Parameters of the transition functions:
          Loss Normal-1 Normal-2  Gain
Loss     0.000    2.879    2.613 2.812
```

```
Normal-1 6.508    0.000    2.586 7.076
Normal-2 7.890    3.109    0.000 5.797
Gain     4.081    4.084    2.700 0.000
```

We can also plot the model with higher posterior probability and the classification of genes using information from all models visited: that is, through Bayesian Model Averaging:

```
> plot(fit, cex = 1.1)
```



The color 'green' is assigned to states of loss, the 'black' to normal states and the 'red' to gains. Note that two states have been labeled as 'Normal'. As statistical states do not always correspond to biological states (for example, a mixture of two normal distributions -two hidden states- might be needed to

fit the distribution of the normal copy numbers), RJaCGH does an automatic labeling giving to each hidden state a probability of beign a state of gain, normal or loss. It is based on the posterior means and variances of the hidden states and the arguments `normal.reference` (the reference value for the mean of the normal state -no change-) and `window` (a multiplier of the standard deviation of the data that sets how much can the distribution of a state of normal copy number separate from the `normal.reference` (see help for further details). We can see the default relabelling:

```
> round(fit[[4]]$state.labels, 3)
```

```
          Loss Normal  Gain
Loss     1.000  0.000 0.000
Normal-1 0.198  0.799 0.004
Normal-2 0.015  0.895 0.090
Gain     0.000  0.000 1.000
```

The user can explore different thresholds with (not shown):

```
> plot(relabelStates(fit, window = 0.25))
> plot(relabelStates(fit, window = 2))
```

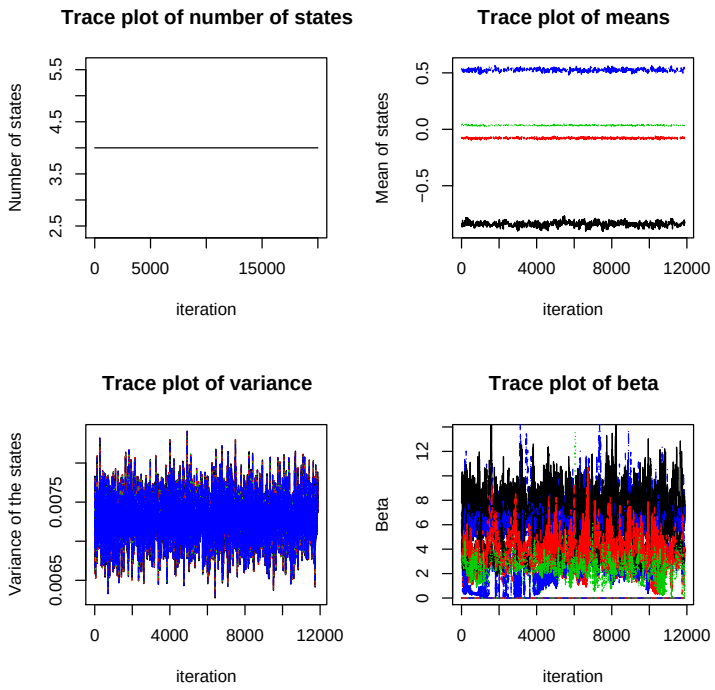
When a good labeling is found, we can update the fit:

```
> fit <- relabelStates(fit, window = 1.25)
```

There are other methods to extract more information, as `states()`, `model.averaging()` or `smoothMeans()`. They will be introduced in the next section.

We can also inspect the exploration of the parameter space in the most visited model:

```
> trace.plot(fit)
```



If we don't see good mixing we can re-adjust the jumping parameters:

- If the lines are too straight for some parameters, we must reduce its corresponding jumping parameters and refit.
- If the lines oscilate too much, we should refit with greater jumping parameters.
- The parameter that rule the movements amongst states is `tau.split.mu`, and the parameters that rule the means, the variances and `beta` are `sigma.tau.mu`, `sigma.tau.sigma.2` and `sigma.tau.beta`.

We can also check the good mixing of the algorithm looking at the proportion of the different values sampled for μ, σ^2 and β: it should not be very low nor very high; some authors say that it should roughly be around 0.23. We'll do it for the model with highest posterior probability:

```
> maxK <- as.numeric(names(which.max(table(fit$k))))
> fit[[maxK]]$prob.mu

[1] 0.1954545

> fit[[maxK]]$prob.sigma.2

[1] 0.6104377

> fit[[maxK]]$prob.beta

[1] 0.2569024
```

And finally, we can check that the algorithm has made some jumps between models (birth, death, split and combine movements):

```
> fit$prob.b

[1] 3 2

> fit$prob.d

[1] 4 2

> fit$prob.s

[1] 1

> fit$prob.c

[1] 2
```

Birth and Death are performed with delayed rejection, so for each iteration they are tried two times and we have two values for them (moves accepted in first and second attempt). (Note that these numbers include the burn-in iterations, but the `trace.plot()` not.)

3.2 A different model for every chromosome

We can also fit a different model for every chromosome with the function `RJaCGH()` changing the parameter `model` to 'Chrom'. We'll fit a model to other cell line: 01524. If there is lot of difference in variance between chromosomes every chromosome should have its own set of jumping parameters, so we shouldn't specify them and let `RJaCGH` do a simple search to find 'good' ones. In this example there is no such different variances per chromosomes, but we'll let the program choose them as a demonstration:

```
> y2 <- gm01524$LogRatio[!is.na(gm01524$LogRatio)]
> Pos2 <- gm01524$PosBase[!is.na(gm01524$LogRatio)]
> Chrom2 <- gm01524$Chromosome[!is.na(gm01524$LogRatio)]
> fit.chrom <- RJaCGH(y = y2, Pos = Pos2, Chrom = Chrom2, model = "Chrom",
+     k.max = 4, burnin = 20000, TOT = 10000, NC = 2, deltaT = 0.5)
```

Again, the result of the fit are nested lists. We can access every chromosome in a simple way, because there is a list for every chromosome, and every chromosome is an object of the same class as explained in the former section. For example, to inspect the chromosome 6 we would do the following:

```
> summary(fit.chrom[[6]])

Distribution of the number of hidden states:

    1     2     3     4
0.000 0.996 0.004 0.000
```

```
Model with  2  states:

Distribution of the posterior means of hidden states:
          10%    25%   50%   75%   90%
Normal -0.009 -0.002 0.007 0.015 0.024
Gain    0.515  0.526 0.542 0.555 0.569

Distribution of the posterior variances of hidden states:
         10%   25%   50%  75%   90%
Normal 0.007 0.008 0.009 0.01 0.011
Gain   0.007 0.008 0.009 0.01 0.011

Parameters of the transition functions:
       Normal  Gain
Normal   0.00 4.259
Gain     2.89 0.000
```

We can also see the sequence of hidden states, that is the copy number status for every probe. We can compute it conditionally to a particular model, (with the method `states`) or averaging through every model fit weighted by the posterior probability of that model (method `model.averaging`):

```
> sequence <- states(fit.chrom)
> sequence.averaged <- model.averaging(fit.chrom)
```

We can see the copy number of chromosome 6:

```
> head(sequence[[6]]$states)

[1] Normal Normal Normal Normal Normal Normal
Levels: Normal Gain

> head(sequence.averaged[[6]]$states)

  <NA>   <NA>   <NA>   <NA>   <NA>   <NA>
Normal Normal Normal Normal Normal Normal
Levels: Loss < Normal < Gain
```

And the probability of every state in that chromosome:

```
> head(sequence[[6]]$prob.states)

     Normal Gain
[1,]      1    0
[2,]      1    0
[3,]      1    0
[4,]      1    0
[5,]      1    0
[6,]      1    0

> head(sequence.averaged[[6]]$prob.states)
```

```
             Loss    Normal        Gain
[1,] 0.002704368 0.9930064 0.004289224
[2,] 0.002703428 0.9930044 0.004292184
[3,] 0.002702632 0.9930027 0.004294689
[4,] 0.002702994 0.9930035 0.004293550
[5,] 0.002702632 0.9930027 0.004294689
[6,] 0.002702632 0.9930027 0.004294689
```

We can also see the smoothed values for every probe (this method returns a vector, not a list with as many vectors as chromosomes):

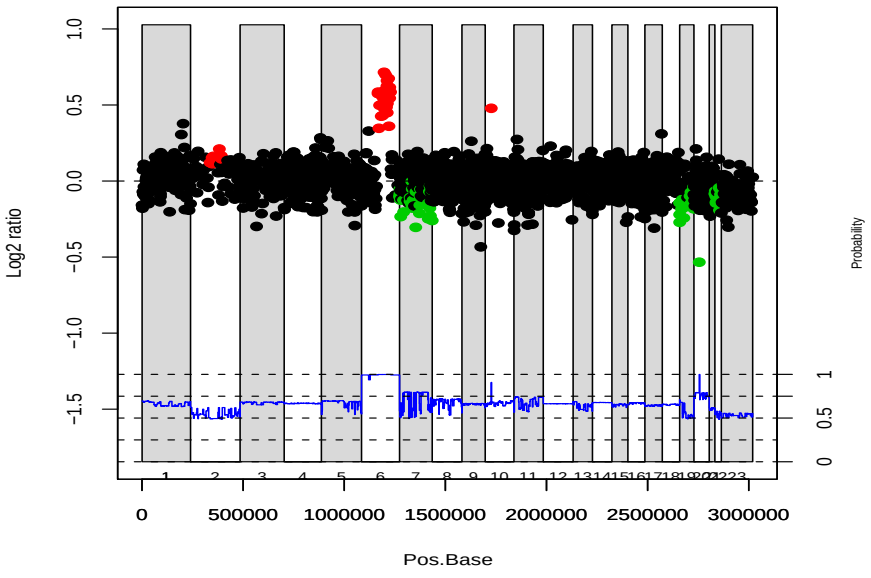
```
> s.means <- smoothMeans(fit.chrom)
> head(s.means)
```

```
           11            12            13            14            15
-0.0123837648 -0.0123837648 -0.0109772175  0.0005394418 -0.0031891922
           16
 0.0049163564
```

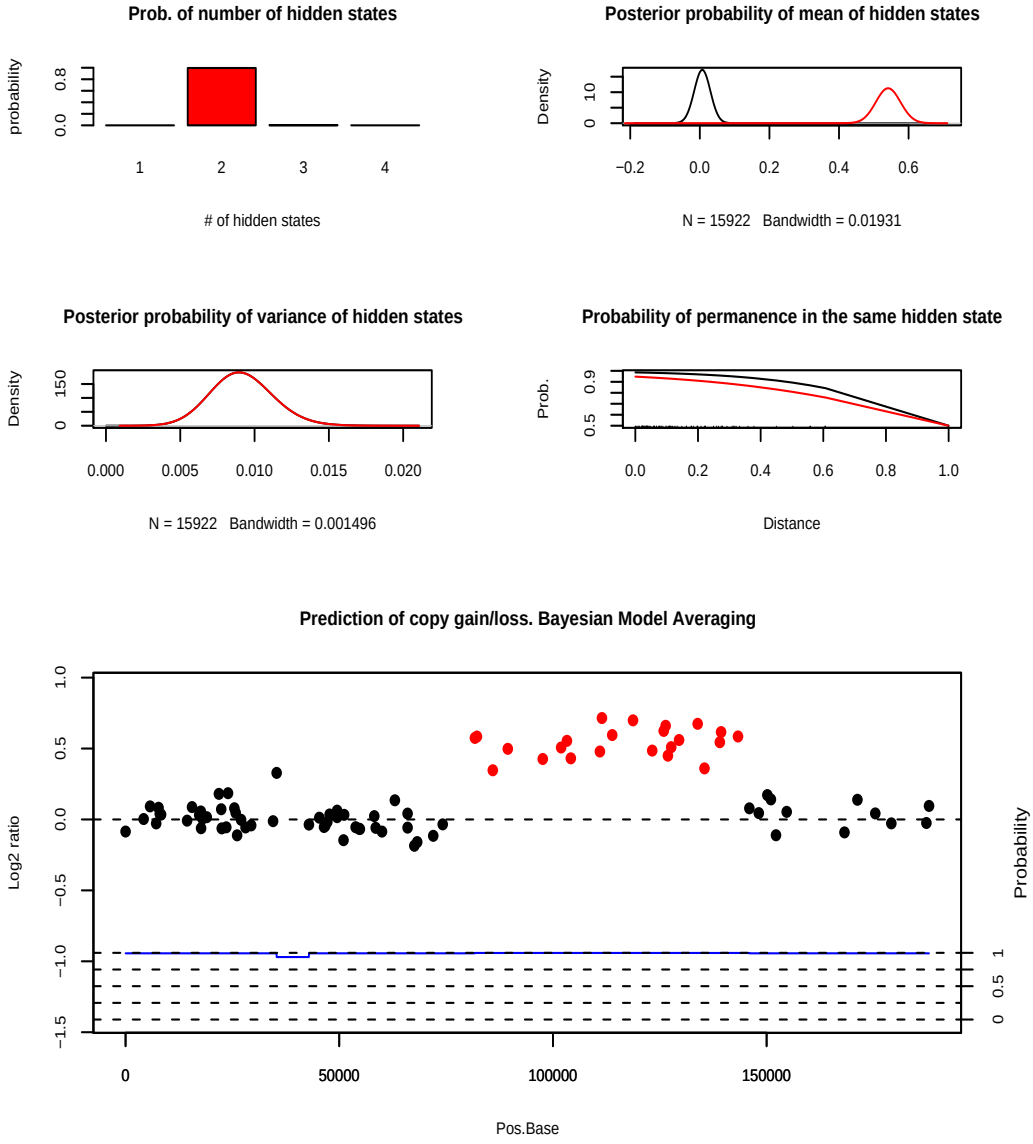
These methods can be also used on a fit with the same model on the whole genome, as the one we fit in the last section.

And we can plot the whole genome or just a chromosome:

```
> plot(fit.chrom)
```

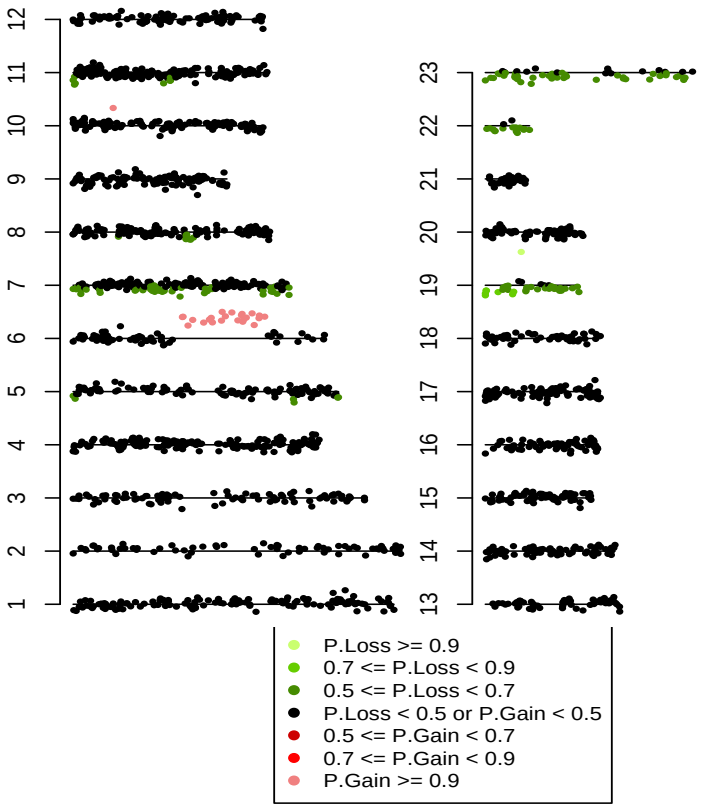



```
> plot(fit.chrom, Chrom = 6)
```



Finally, we can also see the probabilities of alteration in a graph chromosome by chromosome:

```
> genome.plot(fit.chrom)
```



3.3 Fitting several arrays

We can also fit at the same time several arrays (if they have the same probes spotted in the same positions). RJaCGH fits a different model to each of them:

```
> gm07081LR <- gm07081$LogRatio
> gm10315LR <- gm10315$LogRatio
> gm07408LR <- gm07408$LogRatio
> not.NA <- !is.na(gm07081LR) & !is.na(gm10315LR) & !is.na(gm07408LR)
> gm07081LR <- gm07081LR[not.NA]
> gm10315LR <- gm10315LR[not.NA]
> gm07408LR <- gm07408LR[not.NA]
> Pos3 <- gm07081$PosBase[not.NA]
> Chrom3 <- gm07081$Chromosome[not.NA]
> fit.arrays <- RJaCGH(y = cbind(gm07081LR, gm10315LR, gm07408LR),
+     Pos = Pos3, Chrom = Chrom3, model = "genome", k.max = 4,
+     burnin = 20000, TOT = 10000, NC = 2, deltaT = 0.5)
```

The returned object follows the same structure (nested lists); now every object is a list with the result of the fit to each array:

```
> summary(fit.arrays)

Summary for array  gm07081LR :

Distribution of the number of hidden states:

1 2 3 4
0 0 0 1

Model with  4  states:

Distribution of the posterior means of hidden states:
            10%    25%    50%    75%    90%
Normal-1 -0.103 -0.095 -0.080 -0.072 -0.064
Normal-2 -0.002  0.000  0.001  0.003  0.004
Gain-1    0.159  0.184  0.203  0.231  0.256
Gain-2    0.482  0.487  0.492  0.498  0.504

Distribution of the posterior variances of hidden states:
           10%   25%   50%   75%   90%
Normal-1 0.004 0.004 0.004 0.005 0.005
Normal-2 0.004 0.004 0.004 0.005 0.005
Gain-1   0.004 0.004 0.004 0.005 0.005
Gain-2   0.004 0.004 0.004 0.005 0.005

Parameters of the transition functions:
         Normal-1 Normal-2 Gain-1 Gain-2
Normal-1    0.000    1.005  3.326  4.579
Normal-2    4.400    0.000  5.259  6.420
Gain-1      0.568    0.177  0.000  0.383
Gain-2      3.097    2.893  1.882  0.000

================================================

Summary for array  gm10315LR :

Distribution of the number of hidden states:

    1     2     3     4
0.000 0.000 0.482 0.518

Model with  4  states:

Distribution of the posterior means of hidden states:
            10%    25%    50%    75%    90%
Normal-1 -0.136 -0.124 -0.114 -0.101 -0.093
Normal-2 -0.030 -0.027 -0.021 -0.017 -0.014
Normal-3  0.039  0.041  0.045  0.049  0.052
Gain      0.581  0.589  0.595  0.604  0.607
```

```
Distribution of the posterior variances of hidden states:
           10%   25%   50%   75%   90%
Normal-1 0.005 0.006 0.006 0.006 0.006
Normal-2 0.005 0.006 0.006 0.006 0.006
Normal-3 0.005 0.006 0.006 0.006 0.006
Gain     0.005 0.006 0.006 0.006 0.006

Parameters of the transition functions:
         Normal-1 Normal-2 Normal-3  Gain
Normal-1    0.000    1.313    2.839 4.247
Normal-2    4.137    0.000    3.074 7.430
Normal-3    4.378    3.009    0.000 6.316
Gain        4.308    4.886    4.820 0.000

================================================

Summary for array  gm07408LR :

Distribution of the number of hidden states:

    1     2     3     4
0.000 0.000 0.002 0.998

Model with  4  states:

Distribution of the posterior means of hidden states:
          10%    25%    50%    75%    90%
Normal -0.007 -0.006 -0.005 -0.004 -0.003
Gain-1  0.435  0.443  0.451  0.455  0.460
Gain-2  0.576  0.590  0.614  0.626  0.644
Gain-3  0.851  0.890  0.919  0.941  0.959

Distribution of the posterior variances of hidden states:
         10%   25%   50%   75%   90%
Normal 0.004 0.004 0.004 0.004 0.004
Gain-1 0.004 0.004 0.004 0.004 0.004
Gain-2 0.004 0.004 0.004 0.004 0.004
Gain-3 0.004 0.004 0.004 0.004 0.004

Parameters of the transition functions:
       Normal Gain-1 Gain-2 Gain-3
Normal  0.000  6.112  8.380  8.535
Gain-1  2.724  0.000  2.235  3.448
Gain-2  2.795  0.786  0.000  1.499
Gain-3  1.328  0.499  0.294  0.000

================================================

> summary(fit.arrays[["gm07081LR"]])$mu
```

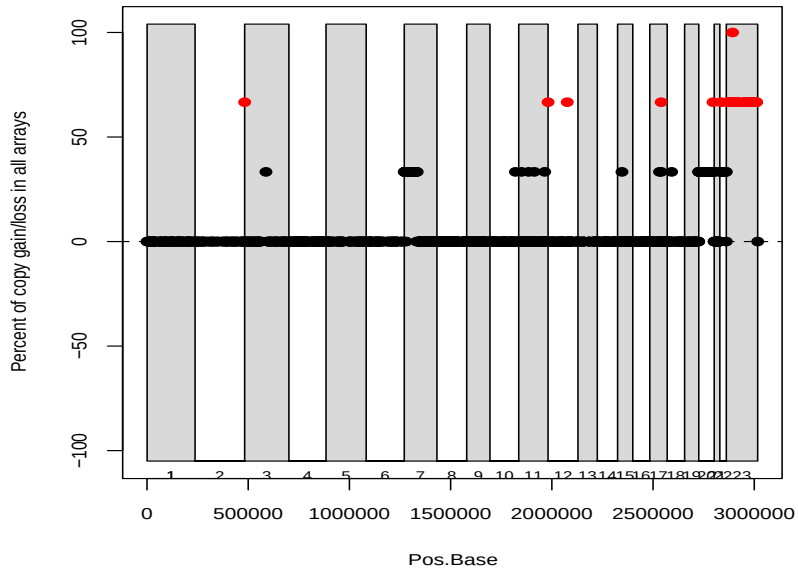
```
                 10%           25%           50%           75%           90%
Normal-1 -0.102640720 -0.0946065789 -0.0797850402 -0.072008429 -0.063703965
Normal-2 -0.002016820 -0.0004569727  0.0006342769  0.002512267  0.004085414
Gain-1    0.159414787  0.1844946003  0.2033508083  0.230989605  0.256185895
Gain-2    0.481700655  0.4872134431  0.4917876248  0.498494909  0.503918122
```

So we can apply the same methods used in previous sections to the whole set of arrays, to a given array (or to a given chromosome of a given array if we fit a different model to every chromosome for each array).

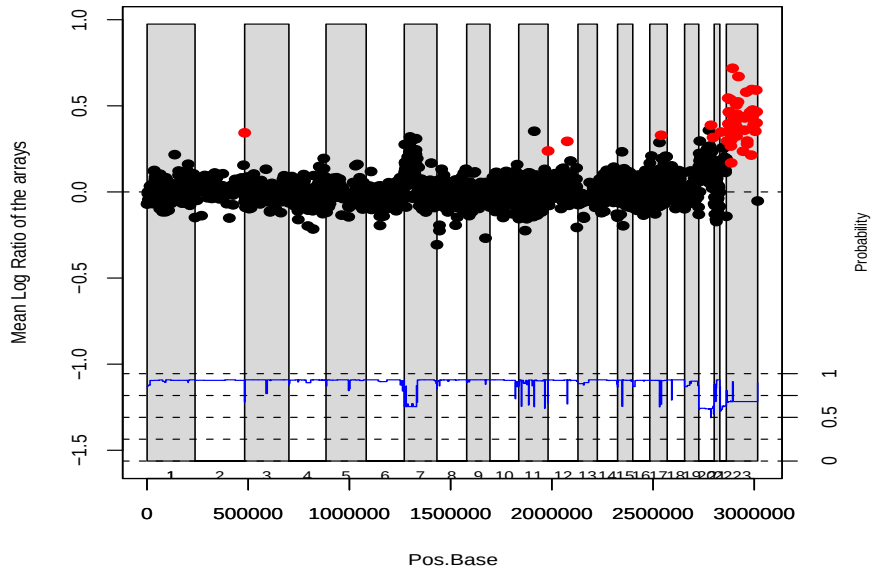
We may want to plot the fit for all the arrays. We can do it in two different ways:

- Plot for every probe the percentage of arrays which have that probe marginally altered.
- Plot for every probe the average probability on all arrays.

```
> plot(fit.arrays, show = "frequency")
```




```
> plot(fit.arrays, show = "average")
```



We can also compare the classification of genes with the true states of Snijders:

```
> seq.states <- model.averaging(fit.arrays[["gm07081LR"]])$states
> table(seq.states, gm07081$Statut[not.NA])
```

```
seq.states Normal Trisomy
    Loss        0       0
    Normal   1871       1
    Gain       18      67
```

3.4 Probabilistic Common Regions

RJaCGH can also compute probabilistic common regions. Note that these regions are different to other approaches, because they take into account the precision or variability inherent to the estimation of the true copy number for every probe on every array considered. There are two different methods:

- `pREC_A` returns regions common to the whole set of arrays with a joint probability of alteration as high as a given threshold.
- `pREC_S` returns regions shared by a subset of arrays (of size as high as a given threshold) with a joint probability within each array as high as a given threshold.

`pREC_A` detects regions common for most of the arrays. It has three arguments, `p` for the minimum probability to call a region altered, `alteration` for the type of alteration ('Gain' or 'Loss') and `array.weights` for the weight that we want to give to each array (by default, it is the same for all of them).

We can find common regions for the three arrays from the last section:

```
> Regions.Gain <- pREC_A(fit.arrays, p = 0.33, alteration = "Gain")
> Regions.Loss <- pREC_A(fit.arrays, p = 0.33, alteration = "Loss")

> Regions.Gain

   Chromosome  Start    End Probes        Prob. Gain
1           2 245000 245000      1 0.675622143119325
2           7      0      0      1 0.340818879933361
3           7   2687   2687      1 0.342828314421495
4           7   3276   6868      8 0.342684285589557
5           7   9696  17181      3 0.342828314421495
6           7  18019  37000     23 0.342786499599319
7           7  38319  38319      1 0.343075552134966
8           7  40773  40773      1 0.343075552134966
9           7  42488  57971     22 0.338629492356668
10         11  13646  13646      1 0.366965686553924
11         11  76848  76848      1 0.363206983408297
12         11 128440 128440      1 0.370463753510269
13         12      0      0      1  0.66034526514375
14         12  94805  94805      1 0.649787845326353
15         15  25615  25615      1 0.342296129958037
16         17  48088  48088      1 0.373681365378281
17         17  56276  56313      2 0.342746314634577
18         20      0  73000     85 0.342967642129042
19         22   3258  33000     14 0.339611933229994
20         23   4000 149342     45 0.672088295778973

> Regions.Loss

[1] "No common regions found"
```

If we want to make this results into a data.frame we would do:

```
> RG <- as.data.frame(print(Regions.Gain))
```

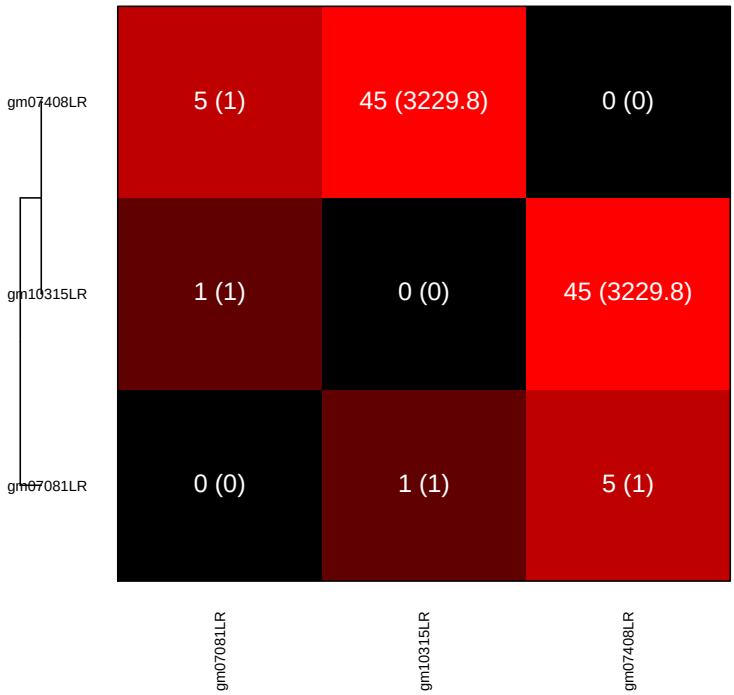
pREC_S is useful to detect subset of arrays that share common alterations. It has the arguments p and alteration but also a freq.array that sets the minimum number of arrays that can form a region.

```
> Regions <- pREC_S(fit.arrays, p = 0.75, alteration = "Gain",
+     freq.array = 2)

> Regions

Common regions of Gain of at least 0.75 probability:
  Chromosome  Start    End Probes              Arrays
1          2 245000 245000      1 gm07081LR;gm07408LR
2         12      0      0      1 gm07081LR;gm07408LR
3         12  94805  94805      1 gm07081LR;gm07408LR
4         17  56276  56276      1 gm07081LR;gm07408LR
5         20  70647  70647      1 gm07081LR;gm07408LR
6         22   7172   7172      1 gm07081LR;gm10315LR
7         23   4000 149342     45 gm10315LR;gm07408LR
```

The result of plotting this object is an image plot that shows for each pair of arrays the number of alterations shared and their mean lengths. Besides, a hierarchical clustering based in that measure is performed and the arrays reordered:

```
> plot(Regions, cex.axis = 0.6)
```

```
$probes
           Var2
Var1        gm07081LR gm10315LR gm07408LR
  gm07081LR         0         1         5
  gm10315LR         1         0        45
  gm07408LR         5        45         0

$length
           Var2
Var1        gm07081LR gm10315LR gm07408LR
  gm07081LR         0     1.000     1.000
  gm10315LR         1     0.000  3229.844
  gm07408LR         1  3229.844     0.000
```

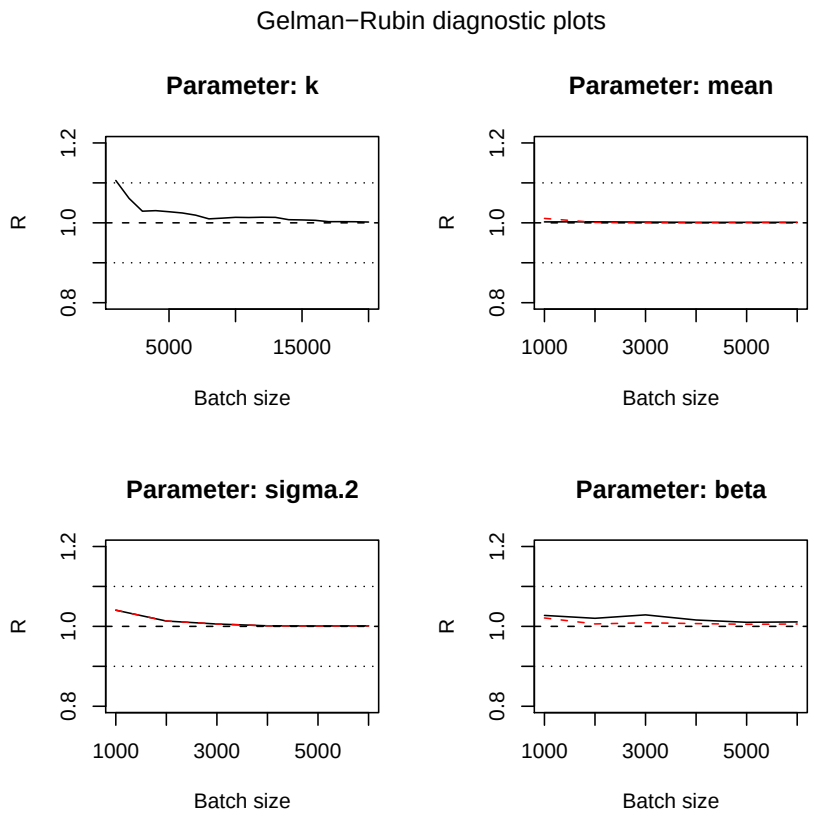


3.5 Checking convergence

There are two alternatives to check the convergence of our fit. One of them is to run one instance of `RJaCGH` with several coupled parallel chains with the argument `NC`. Each of them is heated with the parameter `deltaT` and points from these chains can be exchanged in a swap move (see help file for details). The number of moves peformed is stored in the object `prob.e`.

The other alternative is to run several parallel chains with just one coupled chain and draw the Gelman-Rubin convergence plot. In this example, we use data from cell line gm01524, but only from chromosome 1:

```
> fit <- list()
> for (i in 1:4) {
+     fit[[i]] <- RJaCGH(y = y2[Chrom2 == 1], Pos = Pos2[Chrom2 ==
+         1], k.max = 4, burnin = 50000, TOT = 10000, jump.parameters = jump.parameters,
+         NC = 1)
+ }
> gelman.rubin.plot(fit)
```



We should check that the lines converge to zero, or at least that remain under 1.1. The values that return the function should be under 1.1, too. The results are satisfactory, so we can join the four chains into one:

```
> fit <- collapseChain(fit)
```

And use the former methods to the object `fit`, for example

```
> summary(fit)

Distribution of the number of hidden states:

    1     2     3     4
0.465 0.490 0.039 0.005

Model with  2  states:

Distribution of the posterior means of hidden states:
            10%    25%    50%    75%   90%
Normal-1 -0.055 -0.042 -0.029 -0.014 0.003
Normal-2  0.039  0.053  0.065  0.077 0.087

Distribution of the posterior variances of hidden states:
           10%   25%   50%   75%   90%
Normal-1 0.006 0.006 0.007 0.008 0.009
Normal-2 0.006 0.006 0.007 0.008 0.009

Parameters of the transition functions:
         Normal-1 Normal-2
Normal-1    0.000    1.693
Normal-2    2.024    0.000
```

Other useful functions are `chainsSelect`, which deletes 'outliers' chains (see help file for details) and `trace.plot` to check if the sampler has moved thoroughly through the sample space of a given model.

References

[1] Gelman, A. and Rubin, D. (1992). *"Inference from iterative simulations using multiple sequences"*. Statistical Science, 7. 457–511.

[2] Cappé, Moulines and Rydén. (2005) *"Inference in Hidden Markov Models"*. Springer.

[3] Green, P.J. (1995) *"Reversible Jump Markov Chain Monte Carlo computation and Bayesian model determination"*. Biometrika, 82, 711-732.

[4] Rueda OM, Diaz-Uriarte R. (2007) *"Flexible and Accurate Detection of Genomic Copy-Number Changes from aCGH"*. PLoS Comput Biol, 3(6):e122

[5] Snijders, M. J. et al. (2001) *"Assembly of microarrays for genome-wide measurement of DNA copy number"*. Nature Genetics 29, pp 263 - 264.

[6] Huppé, P. (2005). *"GLAD: Gain and Loss Analysis of DNA"*. R package version 1.6.0. http://bioinfo.curie.fr