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

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

January 22, 2007

1. Statistical Computing Team. 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 CGH data. Basically, it fits a Non Homogeneous Hidden Markov Model through Reversible Jump Markov Chain Montecarlo. The package estimates the probability for every gene to have a normal copy number, gained or lost. The technical report gives full details about the statistical model and the parameterization it uses, plus algorithm details.

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. (10001) with 15 human cells with known karyotypes. We use here the objects in package GLAD 1.6.0. (Huppé and Barillot).

3 Examples:

3.1 Same model for the whole genome

We'll analyze data cell gm13330 from Snijders. First, we take out the missing values, because RJaCGH does not handle NA's:

```
> 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)]
```

We can fit the same Non Homogeneous Hidden Markov Model to the whole genome with the function `RJaCGH`, setting the `model` argument to 'genome'. We will fit HMM's with a maximum of four hidden states, so we'll set the parameter `k.max=4`.
We can also set, if we wish to, the jumping parameters of the MCMC. There are two types:

- 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` and `tau.split.beta`. They are scalars and are 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, tau.split.beta = 0.1)
> fit <- RJaCGH(y = y, Pos = Pos, Chrom = Chrom, model = "genome",
+     k.max = 4, burnin = 50000, TOT = 10000, jump.parameters = jump.parameters,
+     auto.label = 0.75)

Starting Reversible Jump
```

The parameter `auto.label` is optional and represents the expected minimum proportion of normal genes in the sample. It can be set to NULL if it is unknown, but here we will set it to 0.75.

After the fit (it may take a little while), we can inspect the posterior probability of the number of hidden states (different number of copy numbers):

```
> round(prop.table(table(fit$k)), 3)

1 2 3 4
0 0 0 1
```

The fit object is a list with several lists nested; one for each model fitted. For example,

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

is a list with the results of the fit of a model with 4 hidden states. There are several elements inside; for example, we can access to 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 to a model with 4 hidden states and as many columns as hidden states (that is, four).

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

[1] -0.83781894 -0.07789532  0.03590638  0.51815899
```

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

In the case of the function 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

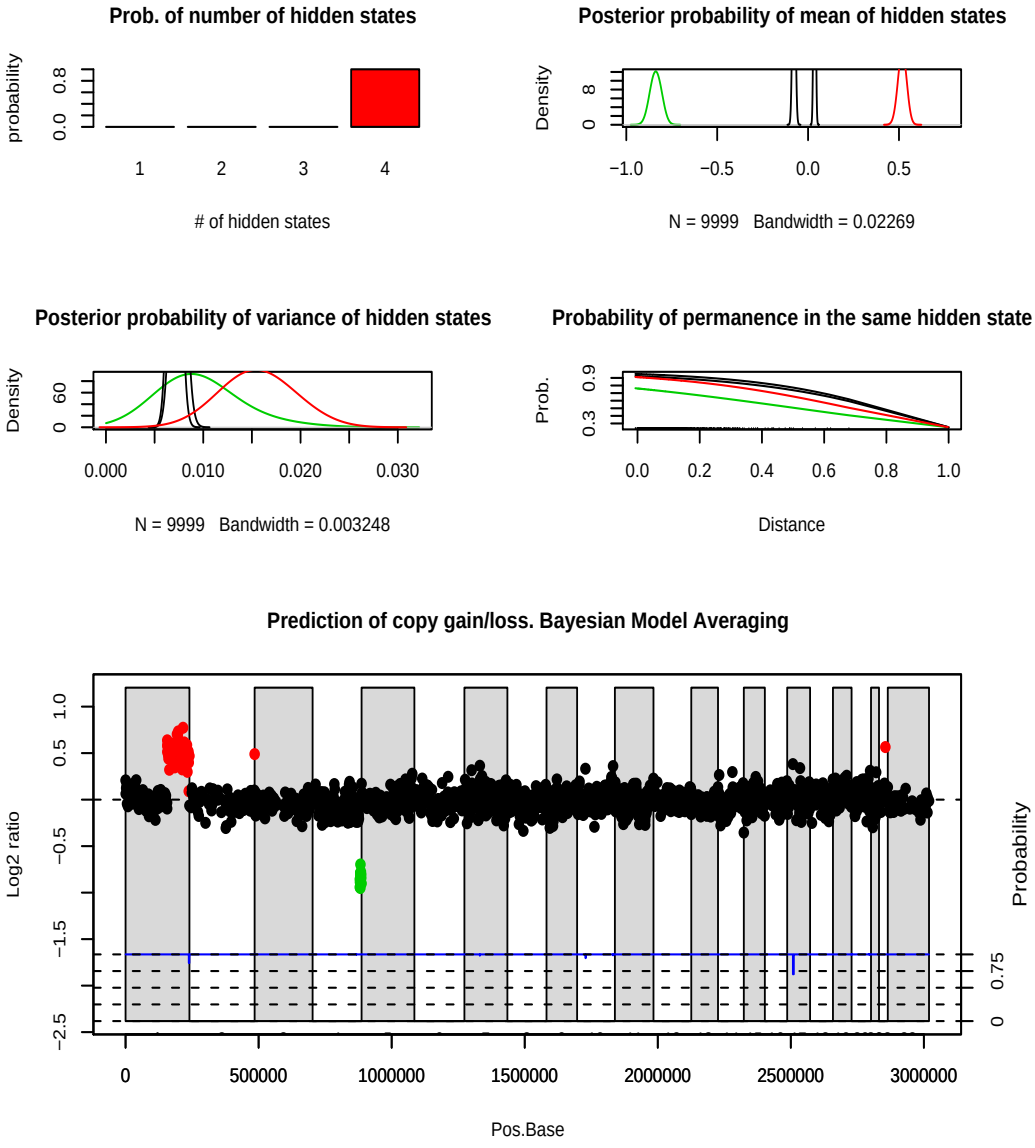
```
> apply(fit[[4]]$beta, c(1, 2), mean)

         [,1]     [,2]     [,3]     [,4]
[1,] 0.000000 2.397859 2.437199 2.314110
[2,] 6.461568 0.000000 2.536138 7.089820
[3,] 7.965401 3.048981 0.000000 5.894768
[4,] 4.096827 4.118661 2.870794 0.000000
```

would give the mean of the posterior distribution of `beta` (the transition matrix depends on the distance between genes; see tech. report for details on the model). We can also summarize the fit and inspect these results. By default, `summary` returns the median of the posterior distributions:

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

     Loss-1      Normal      Normal      Gain-1
-0.83797069 -0.07798217  0.03584619  0.51811133

> summary.HMM$sigma.2

     Loss-1      Normal      Normal      Gain-1
0.008970285 0.007439757 0.007049290 0.015476109
```

We can also plot the model and the classification of genes to the hidden states:

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



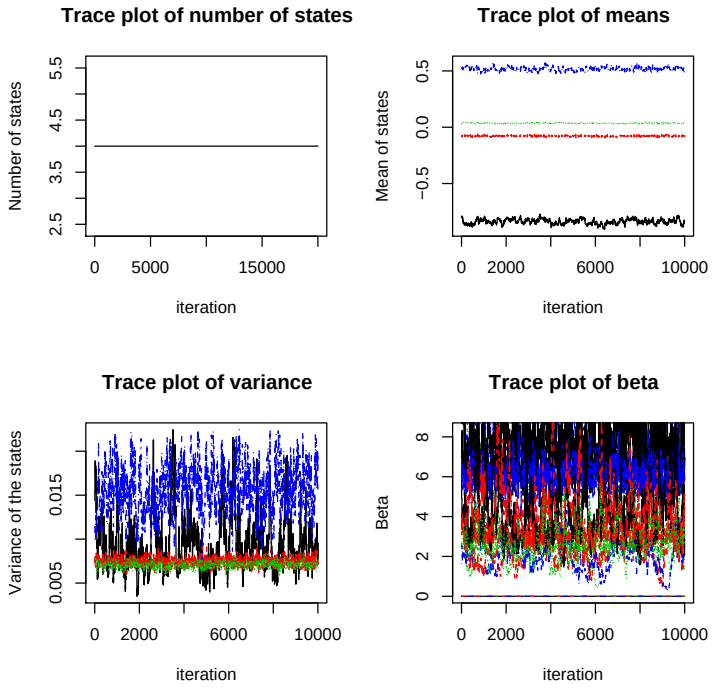
The color 'green' correspond to states of loss, the 'black' to normal states and the 'red' to gains. Note that two states have been labeled as 'Normal'. As statistical states do not always correspond to biological states, RJaCGH does an automatic labeling based on the posterior means and variances of the hidden states and the argument `auto.label` (see the help file for `RJaCGH` for details). If the user wants to make his own relabelling of states, he has to define `fit[[k]]$state.labels`, for the model `k` of interest. It must be a vector of length `k` with elements 'Loss', 'Normal' or 'Gain'. For example, in our case:

```
> fit[[4]]$state.labels <- c("Loss", rep("Normal", 2), "Gain")
```

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

We can also inspect the convergence of 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 parameters that rule the number of states mixing are `tau.split.mu` and `tau.split.beta`, 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 proportion of different values in the chain: it should 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] 0.1651165

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

[1] 0.2836284
```

```
> fit[[maxK]]$prob.beta

[1] 0.2268227
```

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

> fit$prob.d

[1] 3

> fit$prob.s

[1] 3

> fit$prob.c

[1] 2
```

These numbers include the the burn-in iterations.

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 recommend fitting by chromosome only when the data are normalized in every chromosome, because otherwise we couldn't detect a whole chromosome gained/lost. We'll fit a model to other cell line: 01524. Every chromosome should have its own set of jumping parameters, so we won't specify them and let RJaCGH do a simple search to find 'good' ones:

```
> 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 = 50000, TOT = 10000)
```

We can access the results for every chromosome in a simple way, because the objects now contains every list for every chromosome, and every chromosome inlcudes a list of the same kind as explained in the former secton. For example, to inspect the chromosome 6:

```
> summary.chrom.6 <- summary(fit.chrom[[6]])
> summary.chrom.6$mu

     Normal      Gain-1
0.006334982 0.539898058

> summary.chrom.6$sigma.2
```

```
     Normal      Gain-1
0.007904324 0.010125107
```

We can also see the sequence of hidden states, that is the copy number status for every gene. We can compute it conditionally to the most visited 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-1

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

[1] Normal Normal Normal Normal Normal Normal
Levels: Loss < Normal < Gain
```

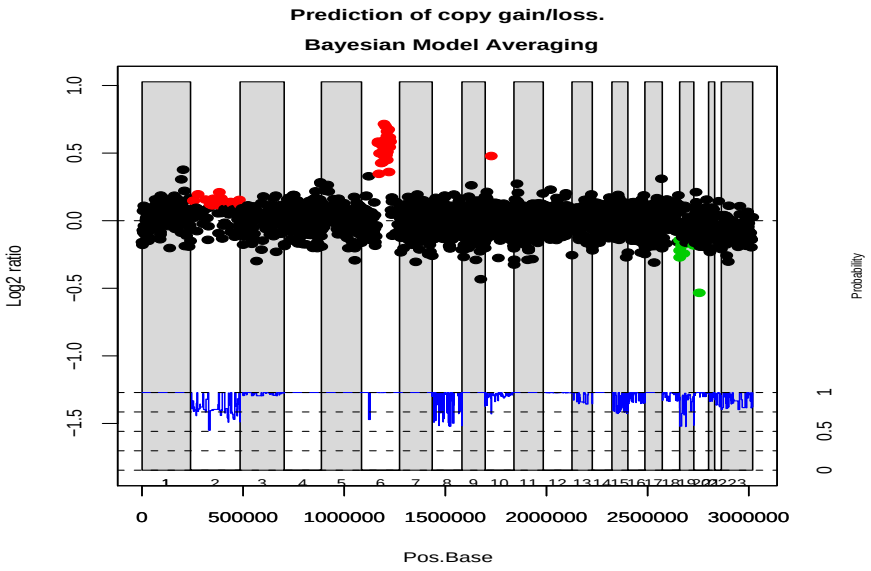
And the probability of every state in that chromosome:

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

     Normal Gain-1
[1,]      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      1  0.000000e+00
[2,]    0      1  0.000000e+00
[3,]    0      1 -5.107537e-18
[4,]    0      1 -5.107537e-18
[5,]    0      1 -5.107537e-18
[6,]    0      1 -5.107537e-18
```

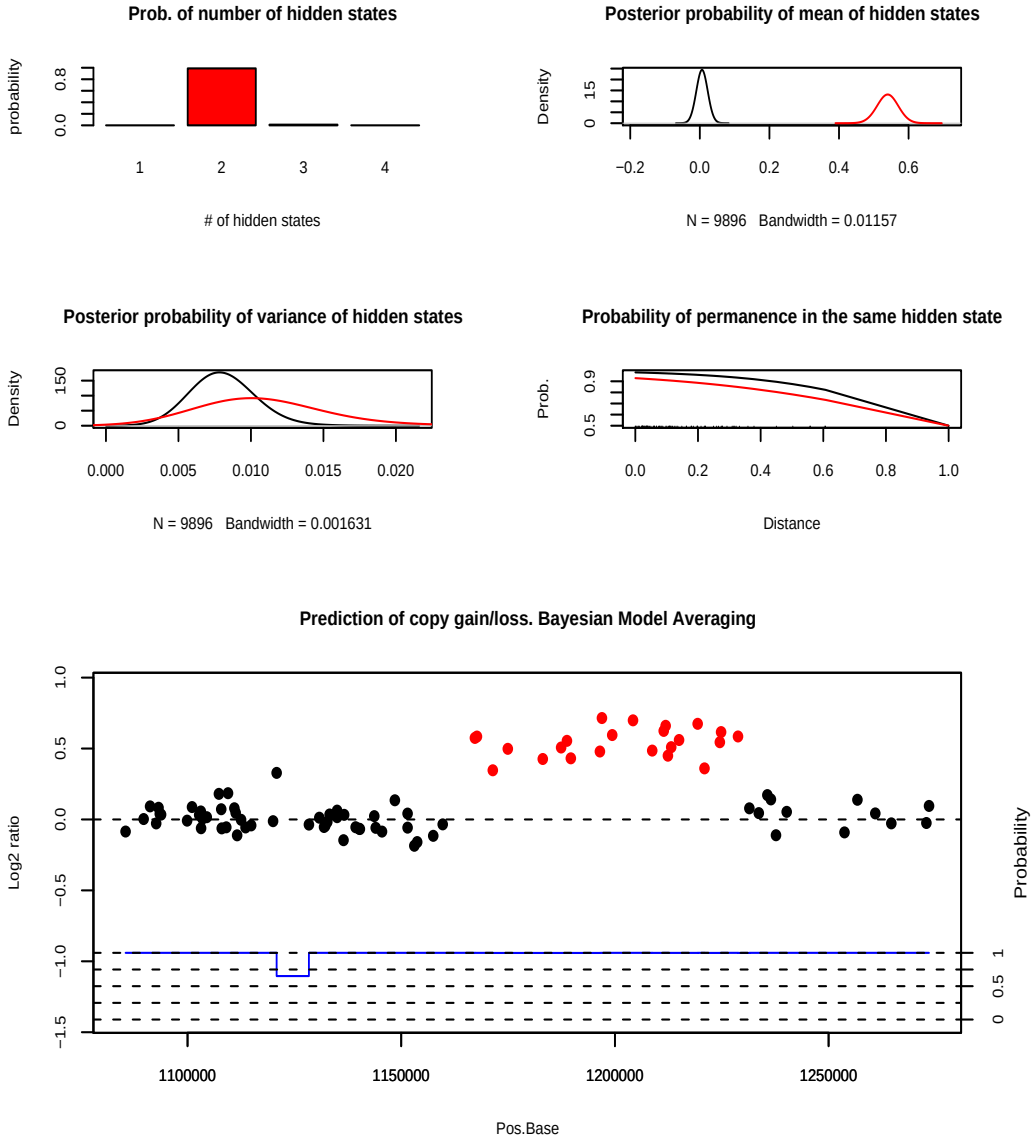
These methods can be also used on a fit with the same model on the whole genome, as the one in the last section.

We can also plot the whole genome or just a chromosome:

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



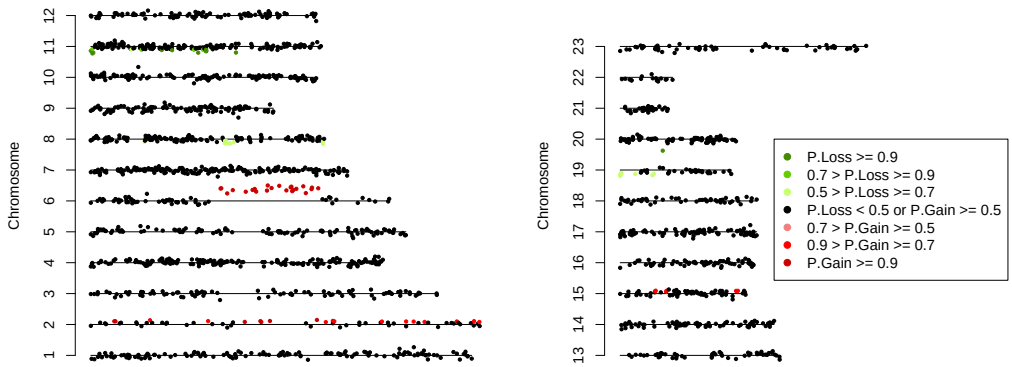

```
> plot(fit.chrom[[6]])
```



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

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

X11 2



3.3 Fitting several arrays

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

```
> gm07081LR <- gm07081$LogRatio
> gm10315LR <- gm10315$LogRatio
> not.NA <- !is.na(gm07081LR) & !is.na(gm10315LR)
> gm07081LR <- gm07081LR[not.NA]
> gm10315LR <- gm10315LR[not.NA]
> Pos3 <- gm07081$PosBase[not.NA]
> Chrom3 <- gm07081$Chromosome[not.NA]
> fit.arrays <- RJaCGH(y = cbind(gm07081LR, gm10315LR), Pos = Pos3,
+     Chrom = Chrom3, model = "genome", k.max = 4, burnin = 50000,
+     TOT = 10000, auto.label = 0.75)

array gm07081LR
Searching jump parameters...
$sigma.tau.mu
[1] 0.01970249 0.01970249 0.01687985 0.00985027

$sigma.tau.sigma.2
[1] 0.04346067 0.04346067 0.03623181 0.02808137

$sigma.tau.beta
[1] 0.1123349 0.1123349 0.1123349 0.1123349

Starting Reversible Jump
array gm10315LR
Searching jump parameters...
$sigma.tau.mu
[1] 0.02306237 0.02306237 0.01625209 0.01218712
```

```
$sigma.tau.sigma.2
[1] 0.05937149 0.05937149 0.03886832 0.03180660

$sigma.tau.beta
[1] 0.13149 0.13149 0.13149 0.13149

Starting Reversible Jump
```

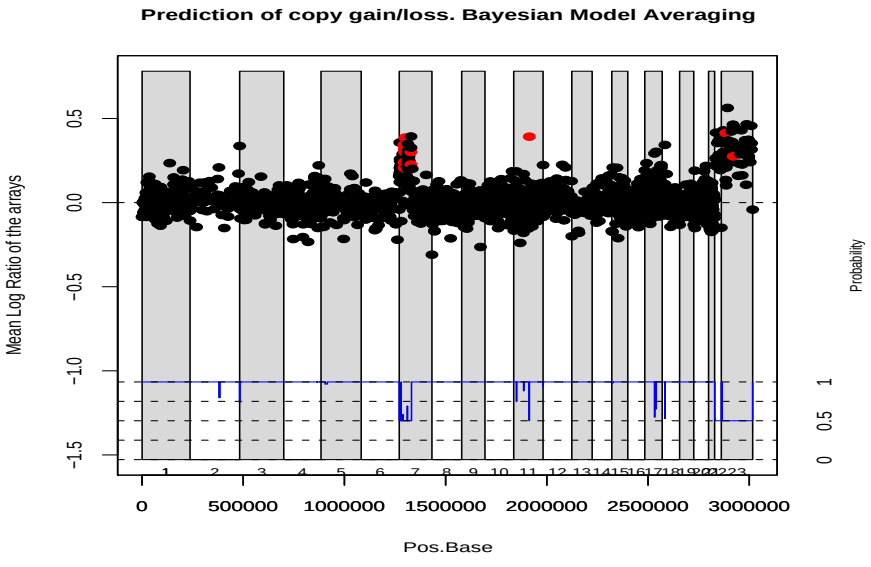
We can examine every one of them the same way, but now the object is a list whose elements are every array fitted:

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

      Normal       Normal       Normal       Gain-1
-0.074073834 -0.004180011  0.029816624  0.464577417

> summary(fit.arrays[["gm10315LR"]])$mu

     Normal      Normal      Gain-1
-0.03949410  0.03750817  0.59004385
```

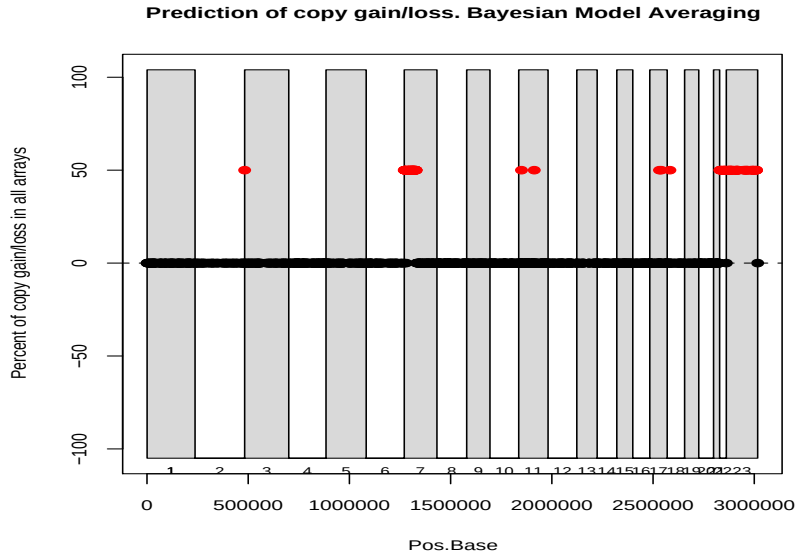
And we can plot the copy number of every gene in several ways:

- Averaging the probability of every gene for every array (by default with the same weight, but there is a `weights` argument to include reliability or importance of each array).
- Plotting the percentage of arrays in which every gene is (marginally) gained or lost.

```
> plot(fit.arrays, method = "averaging")
```



```
> plot(fit.arrays, method = "region")
```



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   1930       1
    Gain        5      68
```

3.4 Probabilistic Minimal Common Regions

RJaCGH can also compute probabilistic minimal common regions. Note that these regions are different to other MCR approaches, because they don't take into account the precision or variability inherent to the estimation of the true copy number for every gene on every array considered. `pMCR` computes all the region of genes with a probability of alteration as high as a given threshold. For a single array, they are interesting because the genes are not independent, so the probability for any sequence is not the product of its marginal probabilities. For several arrays, `pMCR` averages the joint probability of every array as if they were independent and identically distributed sequences. For example,

```
> pMCR(fit, p = 0.9, alteration = "Gain")
```

```
     Chromosome  Start    End #Genes Prob. Gain
[1,]          1 156678 237341     45          1
[2,]          1 240000 240000      1          1
[3,]          2 245000 245000      1          1
[4,]         22  23911  23911      1          1
```

```
> pMCR(fit, p = 0.9, alteration = "Loss")

     Chromosome  Start    End #Genes Prob. Loss
[1,]          4 177282 184000     17          1
```

would give us regions for gains and losses of at least 0.9 probability. Note also that the threshold is for each region, not for all of them; this means that the probability of all regions doesn't have to be over 0.9.

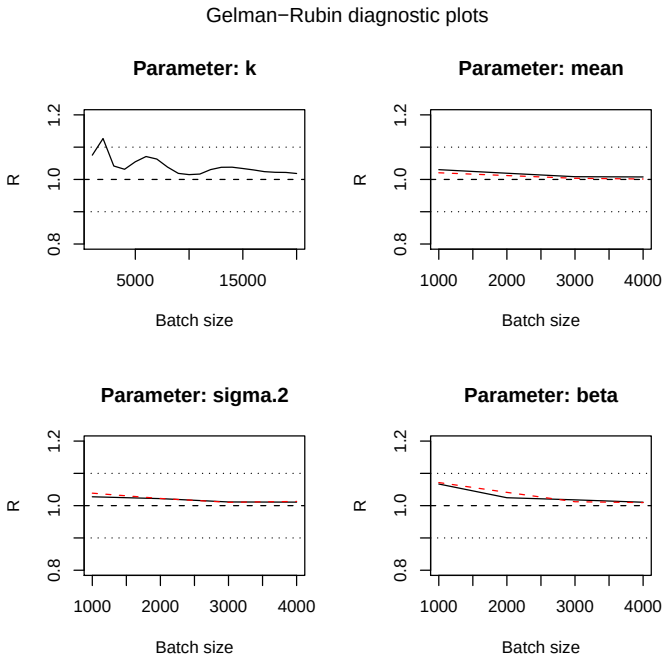
For the two arrays analyzed:

```
> pMCR(fit.arrays, p = 0.5, alteration = "Gain")

     Chromosome Start    End #Genes Prob. Gain
[1,]          7  5765   6868      2        0.5
[2,]          7  9696  17181      3        0.5
[3,]          7 18019  38319     25        0.5
[4,]          7 57472  57971      4        0.5
[5,]         11 76848  76848      1        0.5
[6,]         22  5966  33000     14        0.5
[7,]         23 22055 149342     40        0.5

> pMCR(fit.arrays, p = 0.5, alteration = "Loss")

[1] "No common minimal regions found\n"
```

3.5 Checking convergence

We have seen the function `trace.plot` to check convergence in a given model. But the best way to be sure that RJaCGH has converged is to run several parallel chains and draw Gelman-Brooks convergence plot. In this example, We use data from cell line gm01524, but only from chromosome 1 to save time:

```
> fit <- list()
> for (i in 1:4) {
+     fit[[i]] <- RJaCGH(y = y2[Chrom2 == 1], Pos = Pos2[Chrom2 ==
+         1], k.max = 4, burnin = 30000, TOT = 10000, jump.parameters = jump.parameters,
+         auto.label = 0.75)
+ }
> gelman.brooks.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`.

Other useful function is `selectChains`, which deletes 'outliers' chains. See help file for details.

References

[1] Brooks, S.P. and Gelman, A. (1998). *"General Methods for Monitoring convergence of iterative simulations"*. Journal of Computational and Graphical Statistics. p434-455.

[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, O.M. and Díaz-Uriarte, R. (2006) *"A flexible, accurate and extensible statistical method for detecting genomic copy-number changes*, In prep.

[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