An Introduction to the "genoset" Package

Peter M. Haverty

October 13, 2014

Contents

1 Introduction **2**
- 1.1 Creating Objects . . . 2
- 1.2 Accessing Genome Information . . . 3
- 1.3 Genome Order . . . 5
- 1.4 Using the Subset Features . . . 5

2 Processing Data **6**
- 2.1 Correction of Copy Number for local GC content . . . 6
- 2.2 Segmentation . . . 6
- 2.3 Segments as tables or runs . . . 7
- 2.4 Gene Level Summaries . . . 8

3 Plots **8**

4 *CNSet* and *BAFSet* Objects **10**
- 4.1 Processing B-Allele Frequency Data . . . 10
- 4.2 Plots . . . 11

5 Cross-sample summaries **11**

6 Big Data and *bigmemoryExtras* **12**

1 Introduction

The *genoset* package offers an extension of the familiar Bioconductor *eSet* object for genome assays: the *GenoSet* class. The *GenoSet* class adds location meta-data to the existing feature and phenotype meta-data. This 'locData' allows for various queries, summaries, plots and subsetting operations by genome position. The genoset package also provides a number of convenient functions for working with data associated with with genome locations.

1.1 Creating Objects

In typical Bioconductor style, *GenoSet* objects, and derivatives, can be created using the functions with the same name. Let's load up some fake data to experiment with. Don't worry too much about how the fake data gets made. Notice how assayData elements can be matrices or *DataFrames* with *Rle* columns (from *IRanges*). They can also be *BigMatrix* or *BigMatrixFactor* objects (from *bigmemoryExtras*).

```
> library(genoset)
> data(genoset)
> gs = GenoSet( locData=locData.gr, cn=fake.cn, pData=fake.pData, annotation="SNP6" )
> gs
```

```
GenoSet (storageMode: lockedEnvironment)
assayData: 1000 features, 3 samples
  element names: cn
protocolData: none
phenoData
  sampleNames: K L M
  varLabels: a b ... e (5 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: SNP6
Feature Locations:
GRanges object with 1000 ranges and 0 metadata columns:
        seqnames                ranges strand
           <Rle>             <IRanges>  <Rle>
     p1     chr8 [125000000, 125000000]      *
     p2     chr8 [125030000, 125030000]      *
     p3     chr8 [125060000, 125060000]      *
     p4     chr8 [125090000, 125090000]      *
     p5     chr8 [125120000, 125120000]      *
    ...      ...                     ...    ...
   p996    chr17   [35850000, 35850000]      *
   p997    chr17   [35880000, 35880000]      *
   p998    chr17   [35910000, 35910000]      *
   p999    chr17   [35940000, 35940000]      *
  p1000    chr17   [35970000, 35970000]      *
  -------
  seqinfo: 3 sequences from hg19 genome; no seqlengths
```

```
> rle.ds = GenoSet( locData=locData.gr,
+   cn = fake.cn,
+   cn.segments=DataFrame(
```

```
+     K=Rle(c(rep(1.5,300),rep(2.3,700))),L=Rle( c(rep(3.2,700),rep(2.1,300)) ),
+     M=Rle(rep(1.1,1000)),row.names=rownames(fake.cn)),
+   pData=fake.pData,
+   annotation="SNP6"
+   )
```

Let's have look at what's inside these objects.

```
> names(rle.ds)

[1] "cn"          "cn.segments"

> head( rle.ds[,,"cn"] )

           K            L          M
p1 0.3447339  0.054100830 0.08183075
p2 0.3514447  0.026945155 0.09475004
p3 0.4006017 -0.006892585 0.08953482
p4 0.3920756  0.036985680 0.19294156
p5 0.4085072  0.003319633 0.07288038
p6 0.3279136 -0.014489804 0.15425615

> head( rle.ds[,,"cn.segments"] )

DataFrame with 6 rows and 3 columns
       K     L     M
   <Rle> <Rle> <Rle>
p1   1.5   3.2   1.1
p2   1.5   3.2   1.1
p3   1.5   3.2   1.1
p4   1.5   3.2   1.1
p5   1.5   3.2   1.1
p6   1.5   3.2   1.1
```

Note that `names` lists the data matrices.

1.2 Accessing Genome Information

Now lets look at some special functions for accessing genome information from a genoset object. These functions are all defined for *GenoSet*, *RangedData* and *GRanges* objects. We can access per-feature information as well as summaries of chromosome boundaries in base-pair or row-index units.

There are a number of functions for getting portions of the locData data. `chr` and `pos` return the chromosome and position information for each feature. `genoPos` is like `pos`, but it returns the base positions counting from the first base in the genome, with the chromosomes in order by number and then alphabetically for the letter chromosomes. `chrInfo` returns the genoPos of the first and last feature on each chromosome in addition to the offset of the first feature from the start of the genome. `chrInfo` results are used for drawing chromosome boundaries on genome-scale plots. `pos` and `genoPos` are defined as the floor of the average of each features start and end positions.

```
> head( locData(gs) )

GRanges object with 6 ranges and 0 metadata columns:
     seqnames                 ranges strand
```

```
       <Rle>              <IRanges>  <Rle>
  p1     chr8 [125000000, 125000000]      *
  p2     chr8 [125030000, 125030000]      *
  p3     chr8 [125060000, 125060000]      *
  p4     chr8 [125090000, 125090000]      *
  p5     chr8 [125120000, 125120000]      *
  p6     chr8 [125150000, 125150000]      *
  -------
  seqinfo: 3 sequences from hg19 genome; no seqlengths
```

```
> chrNames(gs)
```

```
[1] "chr8"  "chr12" "chr17"
```

```
> chrOrder(c("chr12","chr12","chrX","chr8","chr7","chrY"))
```

```
[1] "chr7"  "chr8"  "chr12" "chr12" "chrX"  "chrY"
```

```
> chrInfo(gs)
```

```
          start      stop    offset
chr8          1 136970000         0
chr12 136970001 149937501 136970000
chr17 149937502 185907501 149937501
```

```
> chrIndices(gs)
```

```
      first last offset
chr8      1  400      0
chr12   401  800    400
chr17   801 1000    800
```

```
> elementLengths(gs)
```

```
 chr8 chr12 chr17
  400   400   200
```

```
> head(chr(gs))
```

```
[1] "chr8" "chr8" "chr8" "chr8" "chr8" "chr8"
```

```
> head(start(gs))
```

```
[1] 125000000 125030000 125060000 125090000 125120000 125150000
```

```
> head(end(gs))
```

```
[1] 125000000 125030000 125060000 125090000 125120000 125150000
```

```
> head(pos(gs))
```

```
[1] 125000000 125030000 125060000 125090000 125120000 125150000
```

```
> head(genoPos(gs))
```

```
     chr8      chr8      chr8      chr8      chr8      chr8
125000000 125030000 125060000 125090000 125120000 125150000
```

1.3 Genome Order

GenoSet, *GRanges*, and *RangedData* objects can be set to, and checked for, genome order. Weak genome order requires that features be ordered within each chromosome. Strong genome order requires a certain order of chromosomes as well. Features must be ordered so that features from the same chromosome are in contiguous blocks.

Certain methods on `GenoSet` objects expect the rows to be in genome order. Users are free to rearrange rows within chromosome as they please, although if the locData is a `RangedData`, mixing rows from different chromosomes is not possible.

The proper order of chromosomes is desirable for full-genome plots and is specified by the `chrOrder` function. The object creation method `Genoset` creates objects in strict genome order.

```
> chrOrder(chrNames(gs))

[1] "chr8"  "chr12" "chr17"

> gs = toGenomeOrder(gs, strict=TRUE)
> isGenomeOrder(gs, strict=TRUE)

[1] TRUE
```

1.4 Using the Subset Features

GenoSet objects can be subset using array notation. The "features" index can be a set of ranges or the usual logical, numeric or character indices. `chrIndices` with a chromosome argument is a convenient way to get the indices needed to subset by chromosome.

Subset by chromosome

```
> chr12.ds = gs[ chrIndices(gs,"chr12"), ]
> dim(chr12.ds)

featureNames  sampleNames
         400            3

> chrIndices(chr12.ds)

      first last offset
chr12     1  400      0
```

Subset by a collection of gene locations

```
> gene.gr = GRanges(ranges=IRanges(start=c(35e6,127e6),end=c(35.5e6,129e6),
+                         names=c("HER2","CMYC")), seqnames=c("chr17","chr8"))
> gene.ds = gs[ gene.gr, ]
> dim(gene.ds)

featureNames  sampleNames
          84            3

> chrIndices(gene.ds)

      first last offset
chr8      1   67      0
chr17    68   84     67
```

GenoSet objects can also be subset by a group of samples and/or features, just like an ExpressionSet, or a matrix for that matter.

```
> dim(gs[1:4,1:2])
```

```
featureNames  sampleNames
           4            2
```

eSet-derived classes tend to have special functions to get and set specific assayDataElement members (the big data matrices). For example, *ExpressionSet* has the `exprs` function. It is common to put other optional matrices in assayData too (genotypes, quality scores, etc.). These can be get and set with the `assayDataElement` function, but typing that out can get old. *GenoSet* and derived classes use the "k" argument to the matrix subsetting bracket to subset from a specific assayDataElement. In addition to saving some typing, you can directly use a set of ranges to subset the assayDataElement.

```
> all( gs[ 1:4,1:2,"cn"] == assayDataElement(gs,"cn")[1:4,1:2] )
```

```
[1] TRUE
```

2 Processing Data

2.1 Correction of Copy Number for local GC content

Copy number data generally shows a GC content effect that appears as slow "waves" along the genome (Diskin et al., NAR, 2008). The function `gcCorrect` can be used to remove this effect resulting in much clearer data and more accurate segmentation. GC content is best measured as the gc content in windows around each feature, about 2Mb in size.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> gc = rnorm(nrow(gs))
> gs[,,"cn"] = gcCorrect(gs[,,"cn"],gc)
```

2.2 Segmentation

Segmentation is the process of identifying blocks of the genome in each sample that have the same copy number value. It is a smoothing method that attempts to replicate the biological reality where chunks of chromosome have been deleted or amplified.

Genoset contains a convenience function for segmenting data for each sample/chr using the *DNAcopy* package (the CBS algorithm). Genoset adds features to split jobs among processor cores. When the library *parallel* is loaded, the argument n.cores can control the number of processor cores utilized.

Additionally, *GenoSet* stores segment values so that they can be accessed quickly at both the feature and segment level. We use a *DataFrame* object from *IRanges* where each column is a Run-Length-Encoded *Rle* object. This dramatically reduces the amount of memory required to store the segments. Note how the segmented values become just another member of the assayData slot.

Try running CBS directly

```
> library(DNAcopy)
> cbs.cna = CNA(gs[,,"cn"], chr(gs), pos(gs) )
> cbs.smoothed.CNA = smooth.CNA( cbs.cna )
> cbs.segs = segment( cbs.cna )
```

```
Analyzing: Sample.1
Analyzing: Sample.2
Analyzing: Sample.3
```

Or use the convenience function `runCBS`

```
> gs[,,"cn.segs"] = runCBS(gs[,,"cn"],locData(gs))
```

```
Working on segmentation for sample number 1 : K
Working on segmentation for sample number 2 : L
Working on segmentation for sample number 3 : M
```

Try it with *parallel*

```
> library(parallel)
> gs[,,"cn.segs"] = runCBS(gs[, , "cn"],locData(gs), n.cores=3)
> gs[,,"cn.segs"][1:5,1:3]
```

Other segmenting methods can also be used of course.

This function makes use of the parallel package to run things in parallel, so plan ahead when picking "n.cores". Memory usage can be a bit hard to predict.

2.3 Segments as tables or runs

Having segmented the data for each sample, you may want to explore different representations of the segments. *Genoset* describes data in genome segments two ways: 1) as a table of segments, and 2) a Run-Length-Encoded vector. Tables of segments are useful for printing, overlap queries, database storage, or for summarizing changes in a sample. Rle representations can be used like regular vectors, plotted as segments (see `genoPlot`), and stored efficiently. A collection of *Rle* objects, one for each sample, are often stored as one *DataFrame* in a *GenoSet*. *Genoset* provides functions to quickly flip back and forth between table and Rle representations. You can use these functions on single samples, or the whole collection of samples.

```
> head( gs[,,"cn.segs"] )
```

```
$K
numeric-Rle of length 6 with 1 run
  Lengths:      6
  Values : 0.4045

$L
numeric-Rle of length 6 with 1 run
  Lengths:       6
  Values : -0.0013

$M
numeric-Rle of length 6 with 1 run
  Lengths:      6
  Values : 0.1024
```

```
> segs = segTable( gs[,2,"cn.segs"], locData(gs) )
> list.of.segs = segTable( gs[,,"cn.segs"], locData(gs) )
> rbind.list.of.segs = segTable( gs[,,"cn.segs"], locData(gs), stack=TRUE )
> two.kinds.of.segs = segPairTable( gs[,2,"cn.segs"], gs[,3,"cn.segs"], locData(gs) )
> rle = segs2Rle( segs, locData(gs) )
> rle.df = segs2RleDataFrame( list.of.segs, locData(gs) )
> bounds = matrix( c(1,3,4,6,7,10),ncol=2,byrow=TRUE)
> cn = c(1,3,2)
> rle = bounds2Rle( bounds, cn, 10 )
```

`segPairTable` summarizes two Rle objects into segments that have one unique value for each Rle. This is useful for cases where you want genome regions with one copy number state, and one LOH state, for example.

`bounds2Rle` is convenient if you already know the genome feature indices corresponding to the bounds of each segment.

Currently we use *data.frames* for tables of segments. In the near future these will have colnames that will make it easy to coerce these to *GRanges*. Coercion to *GRanges* takes a while, so we don't do that by default.

2.4 Gene Level Summaries

Analyses usually start with SNP or probeset level data. Often it is desirable to get summaries of assayData matrices over an arbitrary set of ranges, like exons, genes or cytobands. The function `rangeSampleMeans` serves this purpose. Given a *RangedData* or *GRanges* of arbitrary genome ranges and a *GenoSet* object, `rangeSampleMeans` will return a matrix of values with a row for each range.

`rangeSampleMeans` uses `boundingIndicesByChr` to select the features bounding each range. The bounding features are the features with locations equal to or within the start and end of the range. If no feature exactly matches an end of the range, the nearest features outside the range will be used. This bounding ensures that the full extent of the range is accounted for, and more importantly, at least two features are included for each gene, even if the range falls between two features.

`rangeColMeans` is used to do a fast average of each of a set of such bounding indices for each sample. These functions are optimized for speed. For example, with 2.5M features and 750 samples, it takes 0.12 seconds to find the features bounded by all Entrez Genes (one RefSeq each). Calculating the mean value for each gene and sample takes 9 seconds for a matrix of array data and 30 seconds for a DataFrame of compressed Rle objects.

Generally, you will want to summarize segmented data and will be working with a *DataFrame* of *Rle*, like that returned by `runCBS`.

As an example, let's say you want to get the copynumber of your two favorite genes from the subsetting example:

Get the gene-level summary:

```
> boundingIndicesByChr( gene.gr, locData(gs) )

     left right
HER2  967   985
CMYC   67   135

> rangeSampleMeans(gene.gr, gs, "cn.segs")

          K       L      M
HER2 1.9981 -0.0481 2.9961
CMYC 1.9981 -0.0481 2.9961
```

3 Plots

Genoset has some handy functions for plotting data along the genome. Segmented data "knows" it should be plotted as lines, rather than points. One often wants to plot just one chromosome, so a convenience argument for chromosome subsetting is provided. Like `plot`, `genoPlot` plots x against y. 'x' can be some form of location data, like a *GenoSet*, *RangedData*, or*GRanges*. 'y' is some form of data along those coordinates, like a numeric vector or *Rle*. `genoPlot` marks chromosome boundaries and labels positions in "bp", "kb", "Mb", or "Gb" units as appropriate.

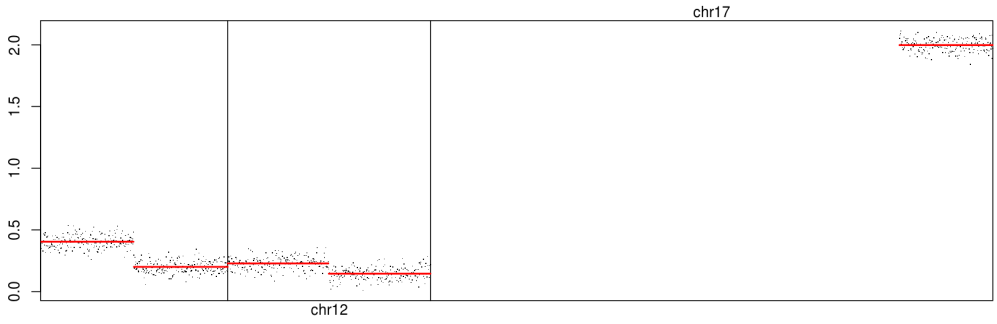

Figure 1: Segmented copy number across the genome for 1st sample

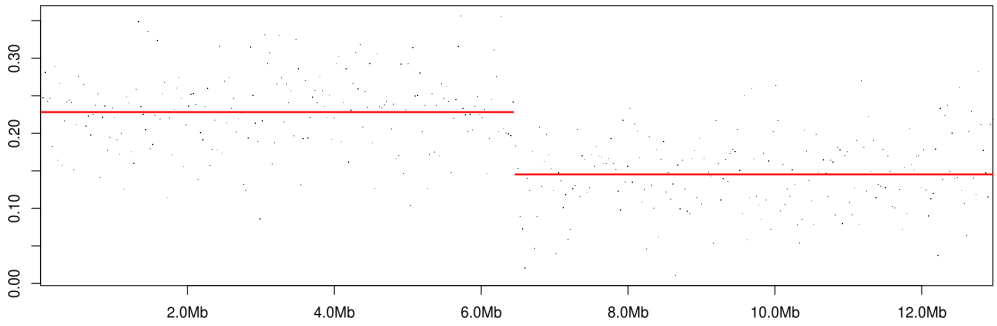

Figure 2: Segmented copy number across chromosome 12 for 1st sample

```
> genoPlot(gs, gs[ , 1, "cn"])
> genoPlot(gs, gs[ , 1, "cn.segs"], add=TRUE, col="red")
```

The result is shown in Fig. 1.

```
> genoPlot(gs,gs[,1,"cn"],chr="chr12")
> genoPlot(gs,gs[,1,"cn.segs"],chr="chr12",add=TRUE, col="red")
```

The result is shown in Fig. 2.
Plot data without a *GenoSet* object using numeric or *Rle* data:

```
> chr12.ds = gs[chr(gs) == "chr12",]
> genoPlot(pos(chr12.ds),chr12.ds[,1,"cn"],locs=locData(chr12.ds))  # Numeric data and location
> genoPlot(pos(chr12.ds),chr12.ds[,1,"cn.segs"],add=TRUE, col="red") # Rle data and numeric position
```

4 *CNSet* and *BAFSet* Objects

Two classes extend *GenoSet*: *CNSet* and *BAFSet*. *CNSet* is the basic copy number object. It requires that one assayDataElement be called "cn" slot, similar to the ExpressionSet uses "exprs". *BAFSet* is intended to store "LRR" or Log-R Ratio and "BAF" or B-Allele Frequency data for SNP arrays. LRR and BAF come from the terms coined by Illumina and are discussed in Peiffer et al., 2008. LRR copy number data, basically log2(tumor/normal). BAF represents the fraction of signal coming from the "B" allele, relative to the "A" allele, where A and B are arbitrarily assigned. BAF has the expected value of 0 or 1 for HOM alleles and 0.5 for HET alelles. Deviation from these expected values can be interpreted as Allelic Imbalance, which is a sign of gain, loss, or copy-neutral LOH. *BAFSet*s require 'lrr' and 'baf' matrices as assayDataElements and have getter/setter methods for these elements.

4.1 Processing B-Allele Frequency Data

B-Allele Frequency (BAF) data can be converted into the "Modified BAF" or mBAF metric, introduced by Staaf, et al., 2008. mBAF folds the values around the 0.5 axis and makes the HOM positions NA. The preferred way to identify HOMs is to use genotype calls from a matched normal (AA, AC, AG, etc.), but NA'ing greater than a certain value works OK. A hom.cutoff of 0.90 is suggested for Affymetrix arrays and 0.95 for Illumina arrays, following Staaf, et al.

Return data as a matrix:

```
> baf.ds = GenoSet( locData=locData.gr, lrr=fake.lrr, baf=fake.baf, pData=fake.pData, annotation="SNP6" )
> baf.ds[, , "mbaf"] = baf2mbaf(baf.ds[, , "baf"], hom.cutoff = 0.90)
```

... or use compress it to a DataFrame of Rle. This uses 1/3 the space on our random test data.

```
> mbaf.data = DataFrame( sapply(colnames( baf.ds),
+   function(x) { Rle( baf.ds[,x, "mbaf"] ) },
+   USE.NAMES=TRUE, simplify=FALSE ) )
> as.numeric(object.size( baf.ds[, ,"mbaf"]))  / as.numeric( object.size(mbaf.data))
```

```
[1] 3.147408
```

Using the HOM SNP calls from the matched normal works much better. A matrix of genotypes can be used to set the HOM SNPs to NA. A list of sample names matches the columns of the genotypes to the columns of your baf matrix. The names of the list should match column names in your baf matrix and the values of the list should match the column names in your genotype matrix. If this method is used and some columns in your baf matrix do not have an entry in this list, then those baf columns are cleaned of HOMs using the hom.cutoff, as above.

Both mBAF and LRR can and should be segmented. Consider storing mBAF as a DataFrame of Rle as only the 1/1000 HET positions are being used and all those NA HOM positions will compress nicely.

```
> baf.ds[,,"baf.segs"] = runCBS( baf.ds[, ,"mbaf"], locData(baf.ds) )
```

```
Working on segmentation for sample number 1 : K
Working on segmentation for sample number 2 : L
Working on segmentation for sample number 3 : M
```

```
> baf.ds[,,"lrr.segs"] = runCBS( baf.ds[, , "lrr"], locData(baf.ds) )
```

```
Working on segmentation for sample number 1 : K
Working on segmentation for sample number 2 : L
Working on segmentation for sample number 3 : M
```

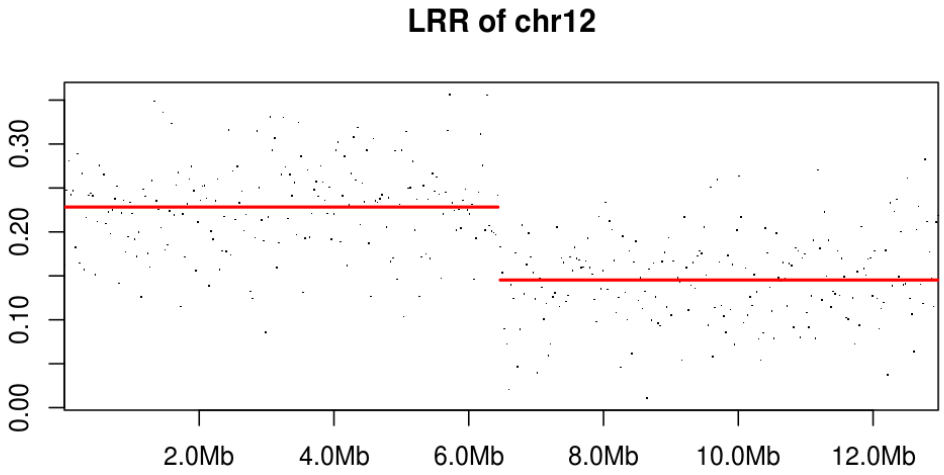

Figure 3: Segmented copy number across the genome for 1st sample

4.2 Plots

```
> genoPlot(baf.ds,baf.ds[,1,"lrr"],chr="chr12",main="LRR of chr12")
> genoPlot(baf.ds,baf.ds[,1,"lrr.segs"],chr="chr12",add=TRUE,col="red")
```

The result is shown in Fig. 3.

```
> par(mfrow=c(2,1))
> genoPlot(baf.ds,baf.ds[,1,"baf"],chr="chr12", main="BAF of chr12")
> genoPlot(baf.ds,baf.ds[,1,"mbaf"],chr="chr12", main="mBAF of chr12")
> genoPlot(baf.ds,baf.ds[,1,"baf.segs"],chr="chr12", add=TRUE,col="red")
```

The result is shown in Fig. 4.

5 Cross-sample summaries

You can quickly calculate summaries across samples to identify regions with frequent alterations. A bit more care is necessary to work one sample at a time if your data "matrix" is a *DataFrame*.

```
> gain.list = lapply(colnames(baf.ds),
+   function(sample.name) {
+     as.logical( baf.ds[, sample.name, "lrr.segs"] > 0.3 )
+ })
> gain.mat = do.call(cbind, gain.list)
> gain.freq = rowMeans(gain.mat,na.rm=TRUE)
```

GISTIC (by Behroukhim and Getz of the Broad Institute) is the standard method for assessing significance of such summaries. You'll find `segTable` convenient for getting your data formatted for input. I find it convenient to load GISTIC output as a RangedData for intersection with gene locations.

6 Big Data and *bigmemoryExtras*

Genome-scale data can be huge and keeping everything in memory can get you into trouble quickly, especially if you like using *parallel*'s `mclapply`.

It is often convenient to use *BigMatrix* objects from the *bigmemoryExtras* package as assayDataElements, rather than base matrices. *BigMatrix* is based on the *bigmemory* package, which provides a matrix API to memory-mapped files of numeric data. This allows for data matrices larger than R's maximum size with just the tiniest footprint in RAM. The *bigmemoryExtras* vignette has more details about using *eSet*-derived classes and *BigMatrix* objects.

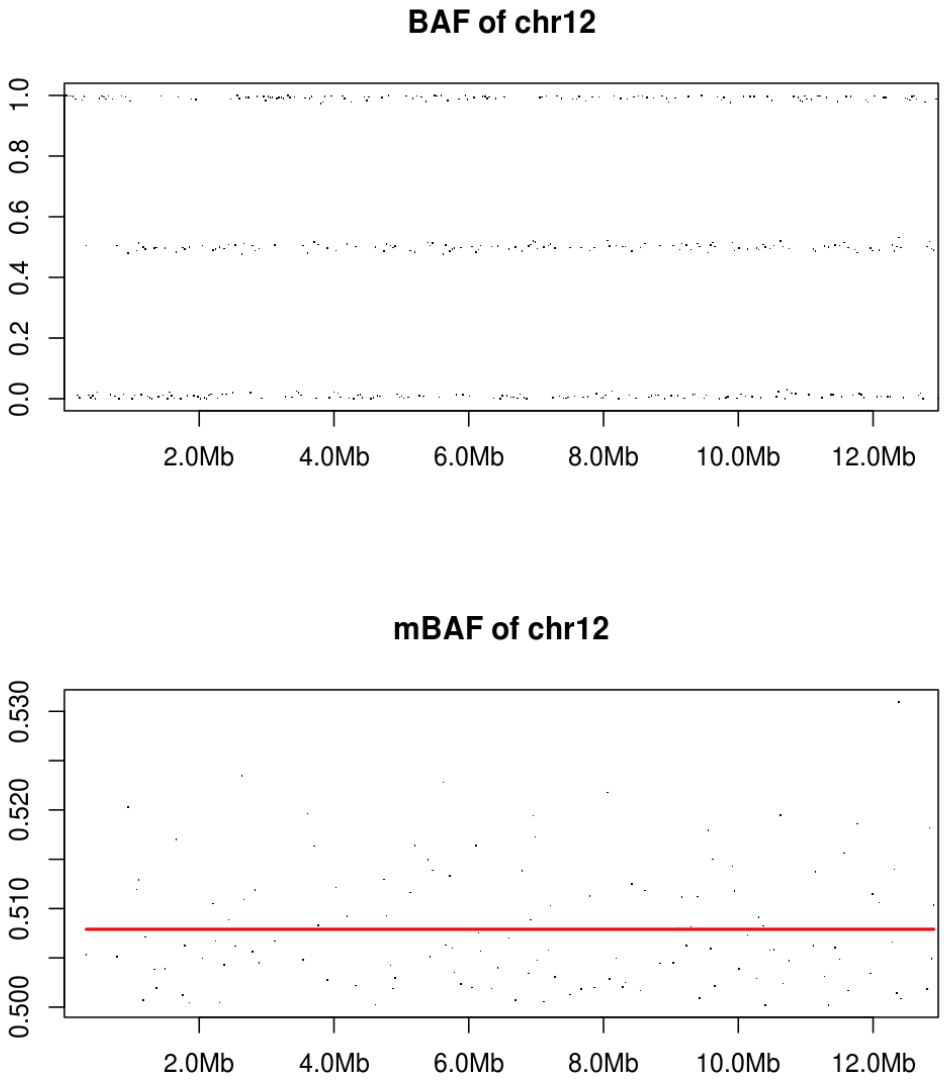

Figure 4: Segmented copy number across the genome for 1st sample