An Introduction to the "genoset" Package

Peter M. Haverty

October 17, 2012

Contents

1 Introduction **2**
1.1 Creating Objects . . . 2
1.2 Accessing Genome Information . . . 5
1.3 Using the Subset Features . . . 6
1.4 Genome Order . . . 9
1.5 Gene Level Summaries . . . 9

2 Processing Data **10**
2.1 Correction of Copy Number for local GC Content . . . 11
2.2 Plots . . . 11

3 BAFSet Objects **12**
3.1 Processing Data . . . 12
3.2 Plots . . . 13
3.3 Cross-sample summaries . . . 14

4 Big Data and bigmemory **14**

1 Introduction

The genoset package offers an extension of the BioConductor eSet object for genome arrays. The package offers three classes. The first class is the "GenoSet" class which can hold an arbitrary number of equal-sized matrices in its assayData slot. The principal addition of the GenoSet class is a "locData" slot that holds a GRanges object from the GenomicRanges package or a RangedData object from the IRanges package. The locData slot allows for quick subsetting by genome position.

Two classes extend GenoSet: CNSet and BAFSet. CNSet is the basic copy number object. It keeps its data in the "cn" slot, similar to the "exprs" slot of the ExpressionSet. 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. LRR is copynumber data processed on a per-snp basis to remove some variability using the expected log-ratio of normal samples with the same genotype. 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.

I will use a CNSet to demonstrate the features that apply to both classes. Some BAFSet-specific features will be handled later in the document.

1.1 Creating Objects

GenoSet, CNSet and BAFSet objects 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 CNSet requires a 'cn' element, BAFSet requires 'lrr' and 'baf', and genoset can take assayData elements with any name. Also notice how assayData elements can be matrices or DataFrames with Rle columns.

```
> library(genoset)
> data(genoset)
> genoset.ds = GenoSet( locData=locData.rd, foo=fake.lrr, pData=fake.pData, annotation="SNP6" )
> baf.ds = BAFSet( locData=locData.rd, lrr=fake.lrr, baf=fake.baf, pData=fake.pData, annotation="SNP6" )
> cn.ds = CNSet( locData=locData.rd, cn=fake.lrr, pData=fake.pData, annotation="SNP6" )
> cn.ds
```

```
CNSet (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:
RangedData with 1000 rows and 0 value columns across 3 spaces
        space             ranges   |
     <factor>          <IRanges>   |
p601     chr8 [1000000, 1000000]   |
p602     chr8 [1030000, 1030000]   |
p603     chr8 [1060000, 1060000]   |
p604     chr8 [1090000, 1090000]   |
```

```
p605      chr8 [1120000, 1120000]   |
p606      chr8 [1150000, 1150000]   |
p607      chr8 [1180000, 1180000]   |
p608      chr8 [1210000, 1210000]   |
p609      chr8 [1240000, 1240000]   |
...        ...                ... ...
p592     chr17 [6730000, 6730000]   |
p593     chr17 [6760000, 6760000]   |
p594     chr17 [6790000, 6790000]   |
p595     chr17 [6820000, 6820000]   |
p596     chr17 [6850000, 6850000]   |
p597     chr17 [6880000, 6880000]   |
p598     chr17 [6910000, 6910000]   |
p599     chr17 [6940000, 6940000]   |
p600     chr17 [6970000, 6970000]   |
```

```
> rle.baf.ds = BAFSet( locData=locData.rd,
+   lrr=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.
+   baf=DataFrame(K=Rle(c(rep(0.05,600),rep(0.5,400))),L=Rle( c(rep(0,700),rep(0.5,300)) ), M=Rle(rep(1,1
+   pData=fake.pData,
+   annotation="SNP6"
+   )
```

Let's have look at what's inside these objects. CNSets extend ExpressionSets and so have all of the same functions and slots, except that the exprs slot and getter and setter functions have been replaced with "cn". The locData slot has been added. This slot holds a GRanges object or a RangedData object that keeps track of the feature positions and allows for quick subsetting. The RangedData is always sorted such that the chromosomes are in contiguous blocks, which provides a constraint on the ordering of the CNSet.

```
> slotNames(genoset.ds)

[1] "locData"           "assayData"         "phenoData"         "featureData"
[5] "experimentData"    "annotation"        "protocolData"      ".__classVersion__"

> phenoData(genoset.ds)

An object of class 'AnnotatedDataFrame'
  sampleNames: K L M
  varLabels: a b ... e (5 total)
  varMetadata: labelDescription

> pData(genoset.ds)

  a b c d e
K A D G J M
L B E H K N
M C F I L O

> annotation(genoset.ds)

[1] "SNP6"

> locData(genoset.ds)
```

```
RangedData with 1000 rows and 0 value columns across 3 spaces
        space             ranges   |
     <factor>          <IRanges>   |
p601     chr8 [1000000, 1000000]   |
p602     chr8 [1030000, 1030000]   |
p603     chr8 [1060000, 1060000]   |
p604     chr8 [1090000, 1090000]   |
p605     chr8 [1120000, 1120000]   |
p606     chr8 [1150000, 1150000]   |
p607     chr8 [1180000, 1180000]   |
p608     chr8 [1210000, 1210000]   |
p609     chr8 [1240000, 1240000]   |
...       ...                ... ...
p592    chr17 [6730000, 6730000]   |
p593    chr17 [6760000, 6760000]   |
p594    chr17 [6790000, 6790000]   |
p595    chr17 [6820000, 6820000]   |
p596    chr17 [6850000, 6850000]   |
p597    chr17 [6880000, 6880000]   |
p598    chr17 [6910000, 6910000]   |
p599    chr17 [6940000, 6940000]   |
p600    chr17 [6970000, 6970000]   |
```

The assayData slot is a collection of equally sized matrix-like objects. Access to required elements can be done by special accessor functions. Others require more typing.

```
> assayDataElementNames(baf.ds)
```

```
[1] "baf" "lrr"
```

```
> head( baf(baf.ds) )
```

```
               K           L           M
p601 0.987558533 0.987558533 0.987558533
p602 0.512913900 0.512913900 0.512913900
p603 0.499877154 0.499877154 0.499877154
p604 0.996214300 0.996214300 0.996214300
p605 0.507650279 0.507650279 0.507650279
p606 0.004420263 0.004420263 0.004420263
```

```
> head( lrr(baf.ds) )
```

```
              K        L         M
p601 0.12769246 3.003308 0.2432253
p602 0.02633469 3.058768 0.2106521
p603 0.06803288 2.996147 0.2044982
p604 0.26454472 3.013705 0.1287740
p605 0.08067449 3.047557 0.1898511
p606 0.01656708 2.969302 0.1278525
```

```
> head( cn(cn.ds) )
```

```
               K        L         M
p601 0.12769246 3.003308 0.2432253
p602 0.02633469 3.058768 0.2106521
p603 0.06803288 2.996147 0.2044982
p604 0.26454472 3.013705 0.1287740
p605 0.08067449 3.047557 0.1898511
p606 0.01656708 2.969302 0.1278525
```

```
> head( assayDataElement(genoset.ds,"foo") )
```

```
               K        L         M
p601 0.12769246 3.003308 0.2432253
p602 0.02633469 3.058768 0.2106521
p603 0.06803288 2.996147 0.2044982
p604 0.26454472 3.013705 0.1287740
p605 0.08067449 3.047557 0.1898511
p606 0.01656708 2.969302 0.1278525
```

1.2 Accessing Genome Information

Now lets look at some special functions for accessing genome information from a genoset object. Methods for RangedData and GRanges have also been added to give them same API. We can access per-probe 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.

```
> chrNames(genoset.ds)
```

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

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

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

```
> chrInfo(genoset.ds)
```

```
         start     stop   offset
chr8         1 12970000        0
chr12 12970001 25937501 12970000
chr17 25937502 32907501 25937501
```

```
> chrIndices(genoset.ds)
```

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

```
> elementLengths(genoset.ds)
```

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

```
> head(chr(genoset.ds))
```

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

```
> head(start(genoset.ds))
```

```
[1] 1000000 1030000 1060000 1090000 1120000 1150000
```

```
> head(end(genoset.ds))
```

```
[1] 1000000 1030000 1060000 1090000 1120000 1150000
```

```
> head(pos(genoset.ds))
```

```
[1] 1000000 1030000 1060000 1090000 1120000 1150000
```

```
> tail(pos(genoset.ds))
```

```
[1] 6820000 6850000 6880000 6910000 6940000 6970000
```

```
> tail(genoPos(genoset.ds))
```

```
   chr17    chr17    chr17    chr17    chr17    chr17
32757501 32787501 32817501 32847501 32877501 32907501
```

1.3 Using the Subset Features

GenoSet objects can be subset using matrix notation. The "features" index can be a set of ranges. chrIndices with a chromosome argument is a convenient way to get the indices needed to subset by chromosome.

Subset by chromosome

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

```
CNSet (storageMode: lockedEnvironment)
assayData: 400 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:
RangedData with 400 rows and 0 value columns across 1 space
        space              ranges   |
     <factor>           <IRanges>   |
p1      chr12     [     1,      1]   |
p2      chr12     [ 32501,  32501]   |
```

```
p3      chr12     [ 65001,  65001]   |
p4      chr12     [ 97501,  97501]   |
p5      chr12     [130001, 130001]   |
p6      chr12     [162501, 162501]   |
p7      chr12     [195001, 195001]   |
p8      chr12     [227501, 227501]   |
p9      chr12     [260001, 260001]   |
...       ...                    ... ...
p392    chr12 [12707501, 12707501]   |
p393    chr12 [12740001, 12740001]   |
p394    chr12 [12772501, 12772501]   |
p395    chr12 [12805001, 12805001]   |
p396    chr12 [12837501, 12837501]   |
p397    chr12 [12870001, 12870001]   |
p398    chr12 [12902501, 12902501]   |
p399    chr12 [12935001, 12935001]   |
p400    chr12 [12967501, 12967501]   |
```

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 = cn.ds[ gene.gr, ]
> gene.ds
```

```
CNSet (storageMode: lockedEnvironment)
assayData: 0 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:
RangedData with 0 rows and 0 value columns across 0 spaces
```

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

```
> cn.ds[1:4,1:2]
```

```
CNSet (storageMode: lockedEnvironment)
assayData: 4 features, 2 samples
  element names: cn
protocolData: none
phenoData
  sampleNames: K L
  varLabels: a b ... e (5 total)
  varMetadata: labelDescription
featureData: none
```

```
experimentData: use 'experimentData(object)'
Annotation: SNP6
Feature Locations:
RangedData with 4 rows and 0 value columns across 1 space
        space            ranges |
     <factor>          <IRanges> |
p601     chr8 [1000000, 1000000] |
p602     chr8 [1030000, 1030000] |
p603     chr8 [1060000, 1060000] |
p604     chr8 [1090000, 1090000] |
```

```
> cn.ds[ c("p1","p2"), cn.ds$b %in% c("E","F")]
```

```
CNSet (storageMode: lockedEnvironment)
assayData: 2 features, 2 samples
  element names: cn
protocolData: none
phenoData
  sampleNames: L M
  varLabels: a b ... e (5 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: SNP6
Feature Locations:
RangedData with 2 rows and 0 value columns across 1 space
      space        ranges |
   <factor>     <IRanges> |
p1    chr12 [    1,     1] |
p2    chr12 [32501, 32501] |
```

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. Likewise, BAFSet has "baf" and "lrr" and CNSet has "cn". It is common to put other optional matrices in assayData too (e.g. 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( cn.ds[ 1:4,1:2,"cn"] == cn(cn.ds)[1:4,1:2] )
```

```
[1] TRUE
```

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

```
[1] TRUE
```

```
> cn.ds[ gene.gr, 1:2, "cn" ]
```

```
     K L
```

1.4 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, this RangedData will prohibit mixing rows from different chromosomes. The function "isGenomeOrder" can check if a RangedData or GRanges object, like the locData slot, is in genome order. Here "genome order" means that the rows within each chromosome are ordered by start position. "Strict genome order" requires that the chromosome be in a certain order too. The proper order of chromosomes is desirable for full-genome plots and is specified by the "chrOrder" function. The function "genomeOrder" returns a list of integer indices to use to set a GenoSet or RangedData to weak, or optionaly, strict genome order. The object creation methods "Genoset", "BAFSet", and "CNSet" order their objects in strict genome order. Methods, like those in the next section, that require genome order use "genomeOrder" to set it themselves, so users are not required to keep track of whether or not they have changed the row order.

```
> chrOrder(chrNames(genoset.ds))

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

> isGenomeOrder(genoset.ds, strict=FALSE)

[1] TRUE

> isGenomeOrder(genoset.ds, strict=TRUE)

[1] TRUE

> genoset.ds = toGenomeOrder(genoset.ds, strict=TRUE)
> isGenomeOrder(genoset.ds, strict=TRUE)

[1] TRUE
```

1.5 Gene Level Summaries

GenoSets contain feature level data. Often it is desirable to get summaries of assayData matrices over an arbitrary set of ranges, like genes or cytobands. The function "rangeSampleMeans" serves this purpose. Given a RangedData or GRanges of arbitrary genome ranges, a GenoSet-based object, and the name of a member of that objects assayData slot, "rangeSampleMeans" will return a matrix of values for each of those ranges and for each sample. For each range, "rangeSampleMeans" uses "boundingIndices" to select the features bounding that range. The bounding features are the features with locations equal to the start and end of the range or those outside the range and closest to the ends of the range. 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.

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:

```
> boundingIndices( c(127e6,127.5e6), c(127e6,128e6), start(chr12.ds) )

     [,1] [,2]
[1,]  400  400
[2,]  400  400

> rangeSampleMeans(gene.gr, baf.ds, "lrr")

               K          L         M
CMYC -0.04203469  2.9548015 1.9974572
HER2 14.02803305 -0.5325792 0.2115682
```

2 Processing Data

Segmentation is the process of identifying blocks of the genome in each sample that have the same copynumber 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 (CBS). GenoSet adds features to split jobs among processor cores. When the library "multicore" 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 directlyq

```
> library(DNAcopy)
> cbs.cna = CNA(cn(cn.ds), chr(cn.ds), pos(cn.ds) )
> 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

```
> assayDataElement(cn.ds,"cn.segs") = runCBS(cn(cn.ds),locData(cn.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
```

Try it with multicore

```
> library(multicore)
> assayDataElement(cn.ds,"cn.segs") = runCBS(cn(cn.ds),locData(cn.ds), n.cores=3)
> assayDataElement(cn.ds,"cn.segs")[1:5,1:3]
> segTable( assayDataElement(cn.ds,"cn.segs"), locData(cn.ds) )
```

Other segmenting methods can also be used of course. See "segs2Rle", "segs2RleDataFrame", and "segs2RangedData" for ways to turn their results into a DataFrame of Rle.

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

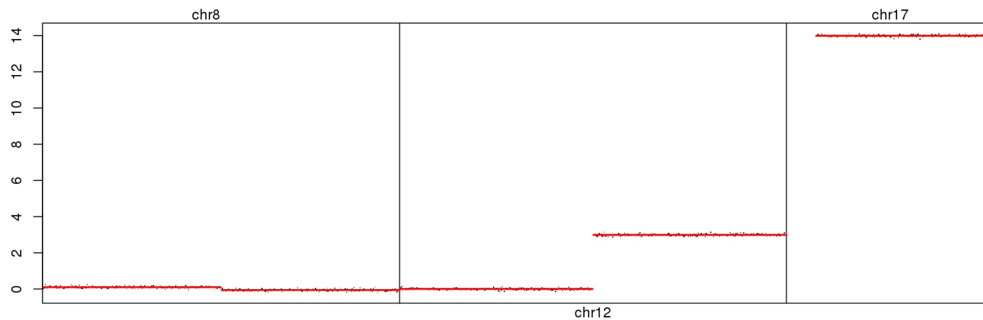


Figure 1: Segmented copy number across the genome for 1st sample

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(cn.ds))
> cn(cn.ds) = gcCorrect(cn(cn.ds),gc)
```

2.2 Plots

CNSet has some handy plots 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. "genoPlot" can plot single samples from a GenoSet (or derived) object. "genoPlot" marks chromosome boundaries and labels positions in "bp", "kb", "Mb", or "Gb" units as appropriate. Simple data and numeric position data can be plotted too.

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

The result is shown in Fig. 1.

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

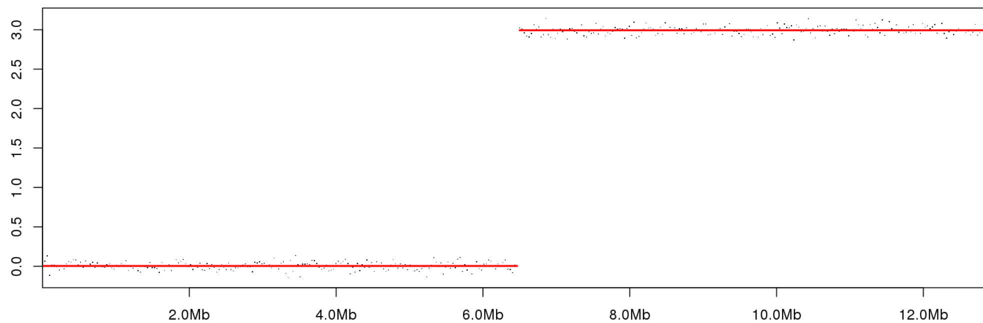


Figure 2: Segmented copy number across chromosome 12 for 1st sample

3 BAFSet Objects

BAFSet objects are similar in most ways to CNSet objects, but require at least two types of data, LRR and BAF, in AssayData. "BAF" represent the B-Allele Frequency or the fraction of array signal coming from one allele. BAF provides a measure of Loss of Heterozygosity (LOH)."LRR" is the Log R Ratio, or log2(tumor/expected), which is basically the traditional copynumber logratio by another name. Both terms come from Illumina and are discussed in Peiffer et al., 2008.

3.1 Processing Data

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 values over 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:

```
> mbaf.data = baf2mbaf(baf(baf.ds),hom.cutoff = 0.90)
> assayDataElement(baf.ds,"mbaf") = mbaf.data
```

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

[1] 3.042874
```

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.

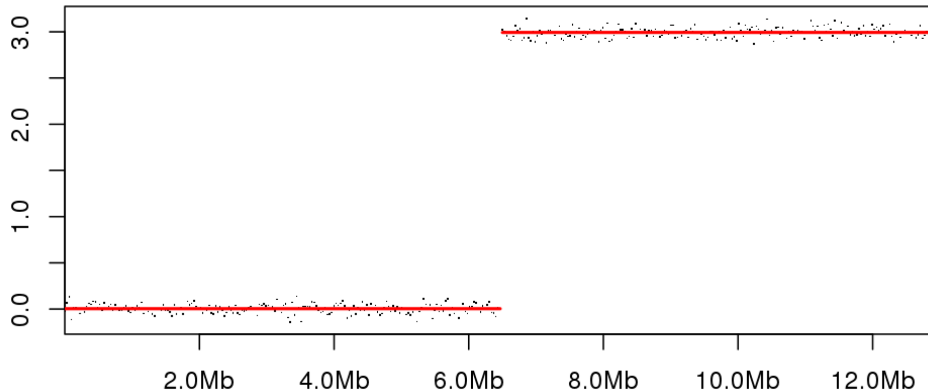


Figure 3: Segmented copy number across the genome for 1st sample

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.

```
> assayDataElement(baf.ds,"baf.segs") = runCBS( assayDataElement(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

> assayDataElement(baf.ds,"lrr.segs") = runCBS( lrr(baf.ds), 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
```

3.2 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.

3.3 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.

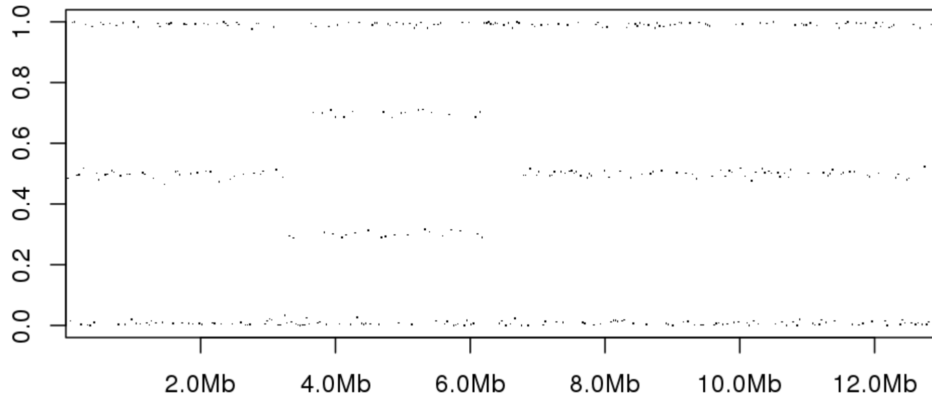
```
> gain.list = lapply(sampleNames(baf.ds),
+   function(sample.name) {
+     as.logical( assayDataElement(baf.ds,"lrr.segs")[,sample.name] > 0.3 )
+ })
> gain.mat = do.call(cbind, gain.list)
> gain.freq = rowMeans(gain.mat,na.rm=TRUE)
```

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.

4 Big Data and bigmemory

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 BigMatrix 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 BigMatrix vignette has more details about using eSet-derived classes and BigMatrix objects.



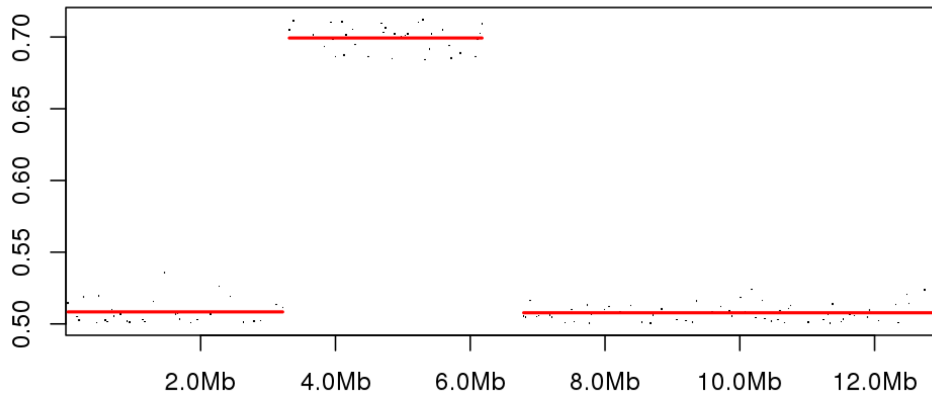


Figure 4: Segmented copy number across the genome for 1st sample