An Introduction to the "genoset" Package

Peter M. Haverty

July 28, 2011

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 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", universe = "hg19")
> baf.ds = BAFSet(locData = locData.rd, lrr = fake.lrr, baf = fake.baf,
+     pData = fake.pData, annotation = "SNP6", universe = "hg19")
> cn.ds = CNSet(locData = locData.rd, cn = fake.lrr, pData = fake.pData,
+     annotation = "SNP6", universe = "hg19")
> 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>   |
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]   |
...        ...                  ... ...
p992      chr8 [12730000, 12730000]   |
p993      chr8 [12760000, 12760000]   |
p994      chr8 [12790000, 12790000]   |
p995      chr8 [12820000, 12820000]   |
p996      chr8 [12850000, 12850000]   |
p997      chr8 [12880000, 12880000]   |
p998      chr8 [12910000, 12910000]   |
p999      chr8 [12940000, 12940000]   |
p1000     chr8 [12970000, 12970000]   |
Universe:  hg19
```

```
> 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(1.1, 1000)), row.names = rownames(fake.lrr)), 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,
+     1000)), row.names = rownames(fake.baf)), 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 RangedData object that keeps track of the feature positions and allows for quick subsetting. The RangedData object 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
```

```
> universe(genoset.ds)
```

```
[1] "hg19"
```

```
> annotation(genoset.ds)
```

```
[1] "SNP6"
```

```
> locData(genoset.ds)
```

```
RangedData with 1000 rows and 0 value columns across 3 spaces
         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]   |
...        ...                  ... ...
p992      chr8 [12730000, 12730000]   |
p993      chr8 [12760000, 12760000]   |
p994      chr8 [12790000, 12790000]   |
p995      chr8 [12820000, 12820000]   |
p996      chr8 [12850000, 12850000]   |
p997      chr8 [12880000, 12880000]   |
p998      chr8 [12910000, 12910000]   |
p999      chr8 [12940000, 12940000]   |
p1000     chr8 [12970000, 12970000]   |
```

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
p1 0.991486487 0.991486487 0.991486487
p2 0.023145160 0.023145160 0.023145160
p3 0.487946861 0.487946861 0.487946861
p4 0.007287484 0.007287484 0.007287484
p5 0.504090535 0.504090535 0.504090535
p6 0.010493410 0.010493410 0.010493410
```

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

```
               K          L         M
p1 -0.031437699 0.13533593 0.3373249
p2  0.019032118 0.08129865 0.3072335
p3  0.023608452 0.13984117 0.3751406
p4  0.031817999 0.08205377 0.3509499
p5  0.002553463 0.11647135 0.5062820
p6 -0.008285740 0.07887775 0.3668297
```

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

```
               K          L         M
p1 -0.031437699 0.13533593 0.3373249
p2  0.019032118 0.08129865 0.3072335
p3  0.023608452 0.13984117 0.3751406
p4  0.031817999 0.08205377 0.3509499
p5  0.002553463 0.11647135 0.5062820
p6 -0.008285740 0.07887775 0.3668297
```

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

```
               K          L         M
p1 -0.031437699 0.13533593 0.3373249
p2  0.019032118 0.08129865 0.3072335
p3  0.023608452 0.13984117 0.3751406
p4  0.031817999 0.08205377 0.3509499
p5  0.002553463 0.11647135 0.5062820
p6 -0.008285740 0.07887775 0.3668297
```

1.2 Accessing Genome Information

Now lets look at some special functions for accessing genome information from a genoset object. Methods for RangedData have also been added to give the 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. “orderedChrs” returns this ordered vector of unique chromosome names. “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.

```
> names(genoset.ds)
```

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

> orderedChrs(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
chr12     1  400      0
chr17   401  600    400
chr8    601 1000    600

> elementLengths(genoset.ds)

chr12 chr17  chr8
  400   200   400

> head(ranges(genoset.ds))

CompressedIRangesList of length 3
$chr12
IRanges of length 400
         start      end width names
[1]          1        1     1    p1
[2]      32501    32501     1    p2
[3]      65001    65001     1    p3
[4]      97501    97501     1    p4
[5]     130001   130001     1    p5
[6]     162501   162501     1    p6
[7]     195001   195001     1    p7
[8]     227501   227501     1    p8
[9]     260001   260001     1    p9
...        ...      ...   ...   ...
[392] 12707501 12707501     1  p392
[393] 12740001 12740001     1  p393
[394] 12772501 12772501     1  p394
[395] 12805001 12805001     1  p395
[396] 12837501 12837501     1  p396
[397] 12870001 12870001     1  p397
[398] 12902501 12902501     1  p398
[399] 12935001 12935001     1  p399
[400] 12967501 12967501     1  p400

...
<2 more elements>
```

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

[1] "chr12" "chr12" "chr12" "chr12" "chr12" "chr12"

> head(start(genoset.ds))

[1]      1  32501  65001  97501 130001 162501

> head(end(genoset.ds))

[1]      1  32501  65001  97501 130001 162501

> head(pos(genoset.ds))

[1]      1  32501  65001  97501 130001 162501

> tail(pos(genoset.ds))

[1] 12820000 12850000 12880000 12910000 12940000 12970000

> tail(genoPos(genoset.ds))

    chr8     chr8     chr8     chr8     chr8     chr8
12820000 12850000 12880000 12910000 12940000 12970000
```

1.3 Using the Subset Features

GenoSet objects can be subset using chromosome names, or with a set of ranges. Please note that the double bracket operator has be re-tasked to subset by chromosome, rather than accessing columns of the phenoData slot. The dollar operator still retains this functionality. Chromosome names will always be treated as character, never numeric or integer.

Subset by chromosome

```
> chr12.ds = 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]   |
Universe:  hg19
```

Subset by a collection of gene locations

```
> gene.rd = RangedData(ranges = IRanges(start = c(3.5e+07, 1.27e+08),
+     end = c(35500000, 1.29e+08), names = c("HER2", "CMYC")), space = c("chr17",
+     "chr8"))
> gene.ds = cn.ds[gene.rd, ]
> 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
Universe:  hg19
```

Dollar still pulls a column from phenoData

```
> genoset.ds$a
```

```
K L M
A B C
Levels: A B C
```

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> |
p1    chr12 [    1,     1] |
p2    chr12 [32501, 32501] |
p3    chr12 [65001, 65001] |
p4    chr12 [97501, 97501] |
Universe:  hg19
```

```
> 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] |
Universe:  hg19
```

1.4 Genome Order

GenoSet and RangedData objects can be set to and checked for genome order. Weak genome order requires that features be ordered within each chromsome. 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. In this package, the use of a RangedData object for feature location info (the locData slot) keeps data for each chromosome in a contiguous block of rows. Users are free to rearrange rows within chromosome as they please. The function "isGenomeOrder" can check if a RangedData 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(names(genoset.ds))

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

> head(genomeOrder(genoset.ds, strict = FALSE))

[1] 1 2 3 4 5 6

> isGenomeOrder(genoset.ds, strict = TRUE)

[1] FALSE
```

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 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(1.27e+08, 127500000), c(1.27e+08, 1.28e+08), start(chr12.ds))

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

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

              K         L         M
HER2 14.0058852 -0.603966 0.2051765
CMYC -0.1778418  2.995809 2.0567748
```

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 chromsome 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)

Using mclapply for segmentation ...
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(cn.ds, "cn.segs")[1:5, 1:3]

DataFrame with 5 rows and 3 columns
        K      L      M
    <Rle>  <Rle>  <Rle>
p1 -0.001 0.1007 0.4005
p2 -0.001 0.1007 0.4005
p3 -0.001 0.1007 0.4005
p4 -0.001 0.1007 0.4005
p5 -0.001 0.1007 0.4005

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

```
$K
  ID chrom loc.start  loc.end num.mark seg.mean
1  K chr12         1  6467501      200  -0.0010
2  K chr12   6500001 12967501      200   2.9978
3  K chr17   1000000  6970000      200  13.9959
4  K  chr8   1000000  6970000      200   0.0980
5  K  chr8   7000000 12970000      200  -0.0541

$L
  ID chrom loc.start  loc.end num.mark seg.mean
1  L chr12         1  6467501      200   0.1007
2  L chr12   6500001 12967501      200   0.9992
3  L chr17   1000000  6970000      200  -0.4952
4  L  chr8   1000000 12970000      400   2.9961

$M
  ID chrom loc.start  loc.end num.mark seg.mean
1  M chr12         1  6467501      200   0.4005
2  M chr12   6500001 12967501      200   0.2010
3  M chr17   1000000  6970000      200   0.2319
4  M  chr8   1000000  6970000      200   0.1412
5  M  chr8   7000000 12970000      200   1.9988
```

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.

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. The function "loadGC" can calculate GC content in bins of a specified size using the genome sequence object from a package like BSgenome. Be careful to use a version of BSgenome that matches the "universe", or mapping, of your data set. loadGC adds a "gc" column to the locData slot.

```
> library(BSgenome.Hsapiens.UCSC.hg19)
> cn.ds2 = cn.ds
> cn.ds2 = loadGC(cn.ds2, expand = 1e+06, bsgenome = Hsapiens)
> head(locData(cn.ds2)$gc)

[1] 40.57496 40.65643 40.81395 40.92443 40.95536 40.92624

> cn(cn.ds2) = gcCorrect(cn(cn.ds2), locData(cn.ds2)$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.

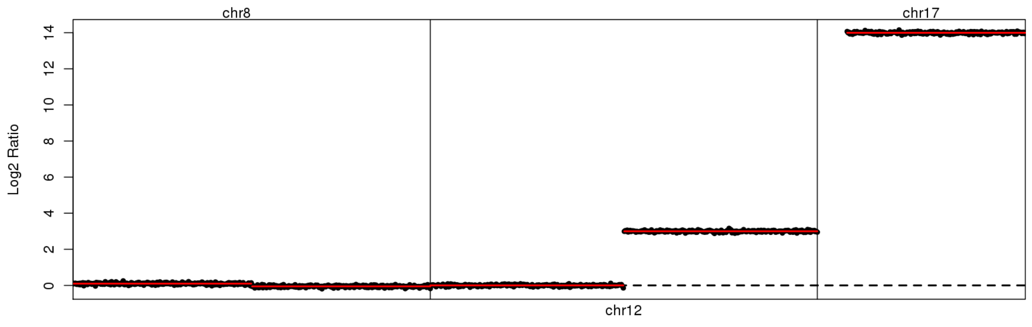


Figure 1: Segmented copy number across the genome for 1st sample

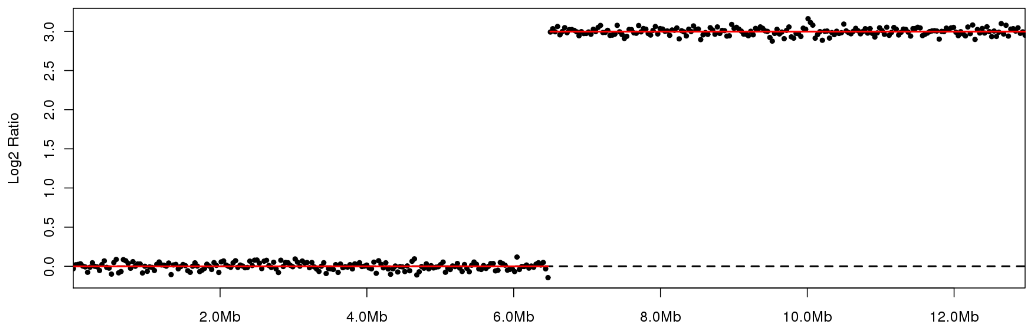


Figure 2: Segmented copy number across chromosome 12 for 1st sample

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

The result is shown in Fig. 1.

```
> genoPlot(cn.ds, 1, chr = "chr12")
> genoPlot(cn.ds, 1, chr = "chr12", element = "cn.segs", 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[["chr12"]]
> genoPlot(pos(chr12.ds), cn(chr12.ds)[, 1], locs = locData(chr12.ds))
> genoPlot(pos(chr12.ds), assayDataElement(chr12.ds, "cn.segs")[, 1],
+     add = TRUE, col = "red")
```

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 = 0.9)
> 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] 3.255168
```

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.

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

Using mclapply for segmentation ...
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))

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

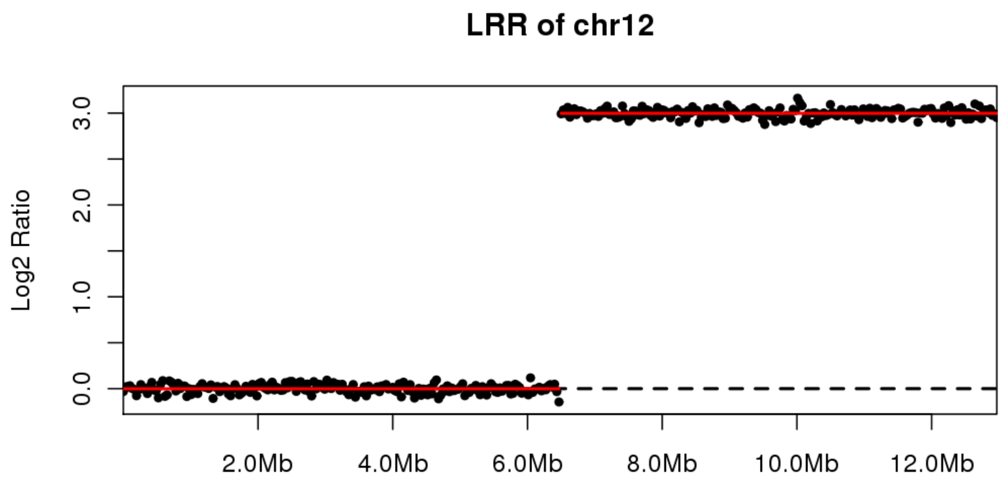


Figure 3: Segmented copy number across the genome for 1st sample

3.2 Plots

BAFSets (and GenoSets) have a genoPlot function as well that additionally requires a "element" argument, which can be "lrr" or "baf" or any other element you happen to have put in "assayData".

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

The result is shown in Fig. 3.

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

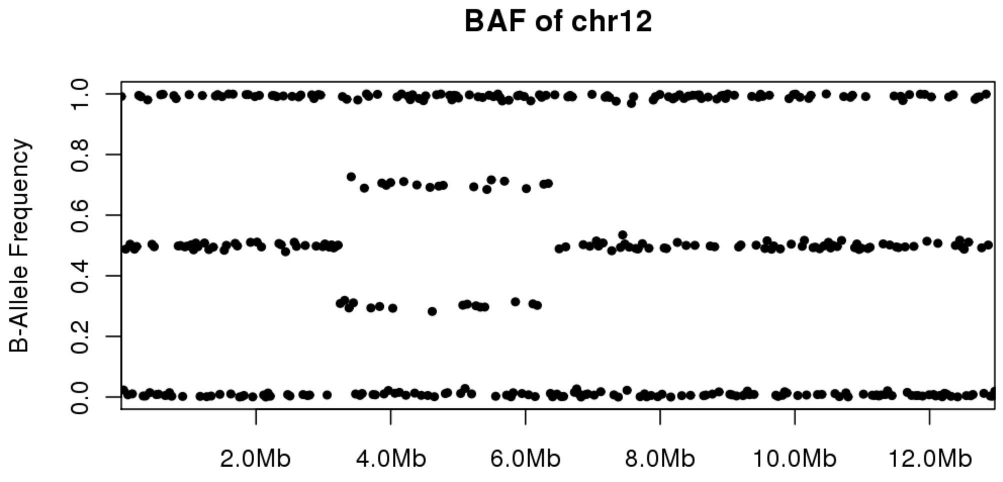
The result is shown in Fig. 4.

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.



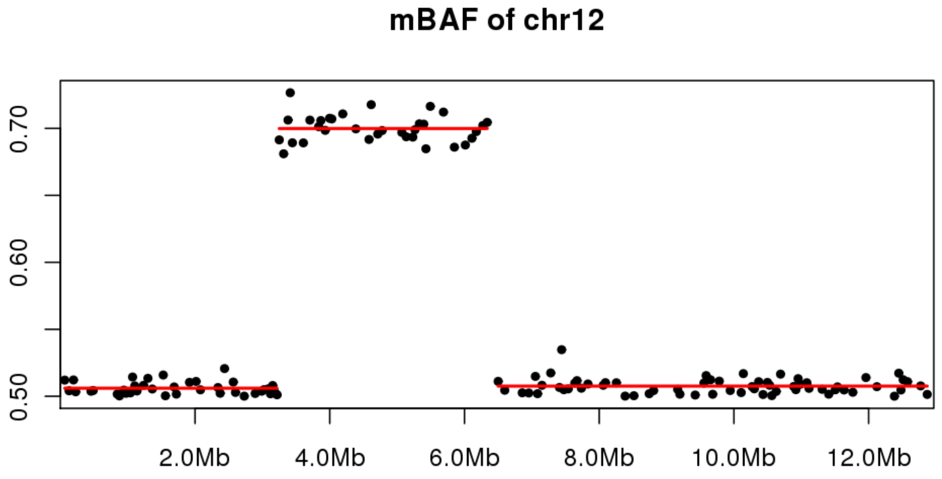


Figure 4: Segmented copy number across the genome for 1st sample