CSAMA 2014 Tutorial - HDF5 based nucleotide tallies

Paul Theodor Pyl

19 Jun 2014

Contents

Introduction

In this tutorial we will work through the full process of generating an HDF5 tally file from a set of `.bam` files, calling variants and visualising our findings, as well as creating a report.

Motivation

The challenges researchers face with the advent of massive sequencing efforts aimed at sequencing RNA and DNA of thousands of samples will need to be addressed now, before the flood of data becomes a real problem.

The effects of the infeasibility of handling the sequencing data of large cohorts with the current standards (BAM, VCF, BCF, GTF, etc.) have become apparent in recent publications that performed population level analyses of mutations in many cancer samples and worked exclusively on the level of preprocessed variant calls stored in VCF/MAF files simply because there is no way to look at the data itself with reasonable resource usage (e.g. in Kandoth et. al 2013).

This challenge can be adressed by augmenting the available legacy formats typically used for sequencing analyses (SAM/BAM files) with an intermediate file format that stores only the most essential information and provides efficient access to the cohort level data whilst reducing the information loss relative to the raw alignments.

This file format will store nucleotide tallies rather than alignments and allow for easy and efficient real-time random access to the data of a whole cohort of samples. The details are described in the following section.

h5vc is a tool that is designed to provide researchers with a more intuitive and effective way of interacting with data from large cohorts of samples that have been sequenced with next generation sequencing technologies.

Nucleotide Tally Definition

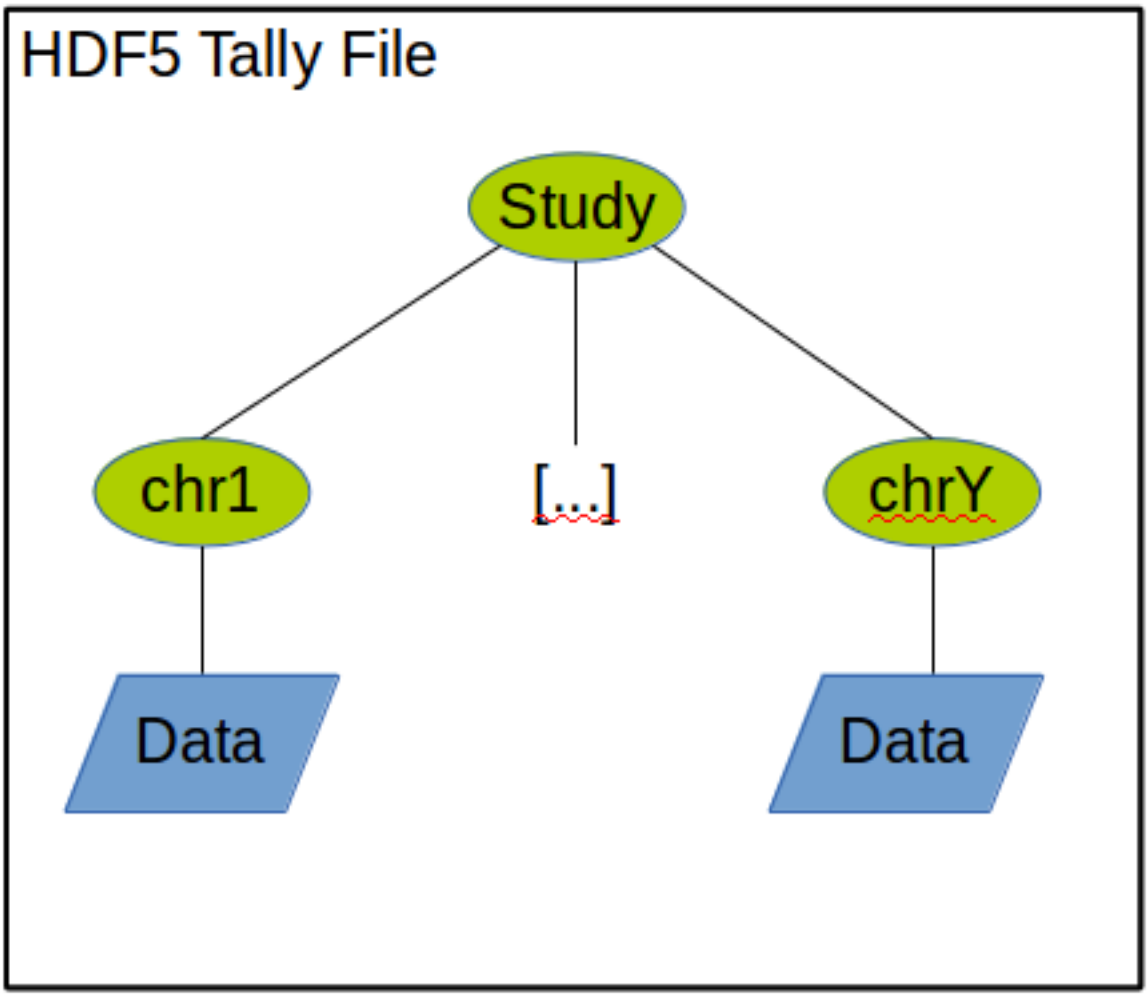
The tally data structure proposed here consists of 4 datasets that are stored for each chromosome (or contig). Those datasets are:

- Counts: A table that contains the number of observed mismatches at any combination of base, sample, strand and genomic position,
- Coverages: A table that contains the number of reads overlapping at any combination of sample, strand and genomic position
- Deletions: A Table that contains the number of observed deletions of bases at any combination of sample, strand and genomic position
- Reference: A Table that contains the reference base against which the calls in the 'Deletions' and 'Counts' table were made.

We outline the basic layout of this set of tables here:

Name	Dimensions	Datatype
Counts	[#bases, #samples, #strands, #positions]	int
Coverages	[#samples, #strands, #positions]	int
Deletions	[#samples, #strands, #positions]	int
Reference	[#positions]	int

An `HDF5` file has an internal structure that is similar to a file system, where groups are the directories and datasets are the files. The layout of the tally file is as follows:



A tally file can contain data from more than one study but each study will reside in a separate tree with a group named with the study-ID at its root and sub-groups for all the chromosomes / contigs that are present in the study. Attached to each of the chromosome groups are the 4 datasets described above.

Additionally each chromsome group stores sample data about the samples involved in the experiment (patientID, type, location in the sample dimension) as `HDF5` attributes. Convenience functions for extracting the metadata are provided, see examples below.

Intorduction to HDF5

The following code examples create an HDF5 file and fill it with some random data to illustrate the functions we can use to interact with HDF5 files.

```
#We use the rhdf5 package
library(rhdf5)
filename <- tempfile() #using a temporary file
h5createFile(filename) #creating the basic file layout
```

```
## [1] TRUE
```

We can use the `h5ls` function to list the content of the file (it is empty right now).

```
h5ls(filename)
```

```
## [1] group  name   otype  dclass dim
## <0 rows> (or 0-length row.names)
```

Let us use `h5write` to write some stuff into the file (we use small examples, in a nucleotide tally file the array dimensions will be several orders of magnitude larger).

```
h5write(obj = array(rnorm(20), dim=c(4,5)), file = filename, name = "/RandomArray")
h5write(obj = rpois(100, 10), file = filename, name = "/PoissonVector")
h5ls(filename)
```

```
##   group          name       otype  dclass   dim
## 0     / PoissonVector H5I_DATASET INTEGER   100
## 1     /   RandomArray H5I_DATASET   FLOAT 4 x 5
```

As you can see each dataset shows up in the listing of the file content (`h5ls(filename)`). We can use `h5read` to extract the data from the file again, if we simpy specify a filename and dataset name we will retrieve the whole dataset. Since this is a really bad idea for large scale projects (e.g. nucleotide tallies of human samples), we can also use subsetting to extract parts of the data. Have a look at `?h5read` if you want to know more about the possible ways of retrieving data.

```
h5read(file = filename, name = "/PoissonVector")
```

```
##   [1]  5 13 14 12 13  7  9 12 11 10 11  7 15  8 12 13 11  9 11  6 13 14 15
##  [24] 15  9  8 17 10  6 15 15 12 20  3 19  6  8 11  7  9 15 15 13 11  5 10
##  [47] 10 10  9 12 11  6 12 14  9  9  8  4 12  8  9  9 14 13 11  7 11 10  9
##  [70] 10 15  9 14  5  9  9  6 10 12  6 12 14  7  7  5 11 15 10 12 12 11 12
##  [93] 12 10  9 16 13 21 11 14
```

```
h5read(file = filename, name = "/RandomArray", index = list(2:4, 3:5))
```

```
##        [,1]     [,2]    [,3]
## [1,] 1.3869 -0.04903 -2.1607
## [2,] 0.3426 -0.32988 -0.5712
## [3,] 0.1115 -0.61792  0.5584
```

> **Question:** How would you extract all the numbers from the `/PoissonVector` dataset, that have an odd index (Hint: `7 %% 2` is equal to 1)

This should give you a basic idea of how we can interact with HDF5 files on a low level. Luckily there are wrapper functions in `h5vc` that we can use to access the file in a more efficient way. Those will be discussed in the following sections.

Getting Started: Loading example data

We use an example set of 24 pairs of `.bam` files (control and case each) of which the last 6 pairs (samples 37 through 48) are whole genome sequencing data and the others are whole exome sequencing data. The `.bam` files have been subsetted to only contain reads that span our region of interest (the NRAS gene, see

below). The samples are numbered 1 to 48 and successive samples belong together (e.g. samples 3 and 4 are the control and case sample of the second pair, ...). All samples are from human tissue and represent pairs of tumour and matched control.

If you have not obtained a copy of the example data, do so now: while connected to the `csama` wlan download [http://192.168.0.9/materials/04_Thursday/labs/ExampleData.zip] and extract the contents into a subfolder of your working directory that you name `ExampleData`. (You can find out the current working directory of your R session by typing `getwd()` at the R prompt.)

```
getwd()
```

```
## [1] "/Users/pyl/Projects/CSAMA2014"
```

The `.bam` files should now be located in the `ExampleData` folder and the first step is to load the required packages and get the list of `.bam` files.

```
library(h5vc)
library(Rsamtools)
```

```
bamFiles <- list.files("ExampleData", "bam$") #list available data files
bamFiles <- file.path( "ExampleData", bamFiles)
# we use scanBamHeader to extract information about the contigs
chromdim <- sapply( scanBamHeader(bamFiles), function(x) x$targets )
colnames(chromdim) <- bamFiles
chromdim[,1]
```

```
##         1         2         3         4         5         6         7
## 249250621 243199373 198022430 191154276 180915260 171115067 159138663
##         8         9        10        11        12        13        14
## 146364022 141213431 135534747 135006516 133851895 115169878 107349540
##        15        16        17        18        19        20        21
## 102531392  90354753  81195210  78077248  59128983  63025520  48129895
##        22         X         Y        MT
##  51304566 155270560  59373566     16569
```

In this tutorial we will look at a region on the genome from 115200000 to 115300000 bases on chromosome one (overlapping the NRAS gene). Note that the `.bam` files we use only contain reads overlapping this region.

```
chrom = "1"
startpos = 115200000
endpos   = 115300000
```

We can create nucleotide tallies using a number of different functions:

- `h5vc::tallyBAM` is a function that creates a nucleotide tally for one given `.bam` file, chromosome and region
- `h5vc::applyTallies` is a function that applies `h5vc::tallyBAM` to a list of `.bam` files and handles the correct merging of the results into one block of data. This implementation uses `BiocParallel::bplapply` with the last registered `BPPARAM`, which defaults to serial execution but can be used with multicore (`BiocParallel::MulticoreParam`) as well as multi-machine (`BiocParallel::BatchJobsParam`) setups.

We will use the `applyTallies` function and if your machine has more than one core available you could test the influence of registereing `BiocParallel::MulticoreParam` objects with different numbers of workers. It is not expected to yield significant speed-ups (and might even slow things down) on a normal laptop computer. Once you move to a server with a powerful RAID setup or a network fileserver, you should be able to speed things up considerably, since there the I/O performance will not be the limiting factor anymore. Furthermore a big part of the used runtime goes to merging the tallies from the different files into one block of data, this can not be parallelised easily and remains an influential factor in the calculation.

```
library(BiocParallel) #load the library to enable parallel execution
maxWorkers <- 2 #Set this to 1 if you want serial execution
tallyList <- list() #outputs go in this list
timeList <- list() #time measurements go here
for( nWorkers in 1:maxWorkers ){
  # set the number of concurrent jobs (see ?register for details)
  register(MulticoreParam(workers = nWorkers))
  timeList[[nWorkers]] <- system.time(
      tallyList[[nWorkers]] <- applyTallies(bamFiles, chrom, startpos, endpos )
    )
  print(paste("Tallied with", nWorkers, "parallel tasks!"))
}
```

```
## [1] "Tallied with 1 parallel tasks!"
## [1] "Tallied with 2 parallel tasks!"
```

```
timeList
```

```
## [[1]]
##    user  system elapsed
##   84.84    3.57   89.05
##
## [[2]]
##    user  system elapsed
##  90.465   7.269 102.447
```

In the tally calls we just ran we didn't specify a reference genome sequence and the algorithm will default to a majority vote amongst all samples in this case. In order to call also homozygous variants reliably we will have to specify the reference sequence, since at a homozygous position the majority-vote based best guess for what the reference base was, will in fact be the alternative allele of the homozygous variant. Please also read `?tallyBAM` for background on the function and an explanation of the parameters.

To avoid this problem we can provide a `DNAString` to the function which will be used as the reference sequence, we get this from one of the `BSGenome` packages. Note that the following code uses `BSgenome.Hsapiens.UCSC.hg19` which has a naming convention that is different from the names of the choromosomes used in the `.bam` files ("chrX" vs. "X") and we need to fix this through the call to `paste0("chr", chrom)`.

```
# we load a reference genome and use a subset of it as a parameter to applyTallies
suppressPackageStartupMessages(library(BSgenome.Hsapiens.UCSC.hg19))
tallies <- applyTallies(
  bamFiles, chrom = chrom, start = startpos, stop = endpos, ncycles = 10,
  reference = BSgenome.Hsapiens.UCSC.hg19[[paste0("chr", chrom)]][startpos:endpos] )
# the first and last 10 sequencing cycles are called unreliable
str(tallies)
```

```
## List of 4
##  $ Counts   : num [1:12, 1:48, 1:2, 1:100001] 0 0 0 0 0 0 0 0 0 0 ...
##   ..- attr(*, "dimnames")=List of 4
##   .. ..$ : chr [1:12] "A.front" "C.front" "G.front" "T.front" ...
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "+" "-"
##   .. ..$ : NULL
##  $ Coverages: int [1:48, 1:2, 1:100001] 0 0 0 0 0 0 0 0 0 0 ...
##   ..- attr(*, "dimnames")=List of 3
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "+" "-"
##   .. ..$ : NULL
##  $ Deletions: int [1:48, 1:2, 1:100001] 0 0 0 0 0 0 0 0 0 0 ...
##   ..- attr(*, "dimnames")=List of 3
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "+" "-"
##   .. ..$ : NULL
##  $ Reference: num [1:100001] 0 0 1 3 1 1 0 0 0 0 ...
```

It is advisable to have a look at `?tallyBAM` and `?applyTallies` for an explanation of the parameters.

> **Question:** Try to replace the `BSgenome` object that is used by one that follows ENSEMBL notation (e.g. "BSgenome.Hsapiens.NCBI.GRCh38") and remove the unneccessary `paste` command. Compare the results of `head(seqlevels(BSgenome.Hsapiens.NCBI.GRCh38))` and `head(seqlevels(BSgenome.Hsapiens.UCSC.hg19))`.

As you can see the resulting object of the call is a simple `list` containing a set of `array`s, which each correspond to one of the datasets we want to write to the HDF5 file.

Now that we have created the tally object we have to write the data to an HDF5 file so that we may reference it at a later point. In this simple example case we could always recreate the tally in each R session since it doesn't take long, but on a genome-wide scale this is completely impractical to do. You can see the substantial amount of time and compute resources used to create a whole-genome tally file as an investment that will pay off in the future, when you are (re-)running analyses and developing methods on the data. (have a look at the help page of `?batchTallies` to see how to calculate tallies for many samples genome-wide using a compute cluster).

We will use the `rhdf5::h5write` function to write the data to the tally file, but first we must set up the tally file with the correct structure of groups and datasets. Note that each element of the `list` that was returned by our call to `applyTallies` corresponds to one dataset in the file.

```
chromlength <- chromdim[chrom,1] #grab the chromosome length from the first sample
study <- "/NRAS" #This will be the name of the main folder in the HDF5 file
tallyFile <- "NRAS.tally.hfs5"
if( file.exists(tallyFile) ){
  file.remove(tallyFile)
}
```

```
## [1] TRUE
```

```
if( prepareTallyFile(tallyFile, study, chrom, chromlength, nsamples = length(bamFiles))){
  h5ls(tallyFile)
}else{
  message( paste( "Preparation of:", tallyFile, "failed" ) )
}
```

```
##     group      name       otype  dclass                    dim
## 0       /      NRAS   H5I_GROUP
## 1   /NRAS         1   H5I_GROUP
## 2 /NRAS/1    Counts H5I_DATASET INTEGER 12 x 48 x 2 x 249250621
## 3 /NRAS/1 Coverages H5I_DATASET INTEGER      48 x 2 x 249250621
## 4 /NRAS/1 Deletions H5I_DATASET INTEGER      48 x 2 x 249250621
## 5 /NRAS/1 Reference H5I_DATASET INTEGER               249250621
```

> **Question:** Investigate `?prepareTallyFile` and read about the influence of the `chunkSize` and `compressionLevel` parameters.

Writing to the HDF5 tally file

Now that we have set up the file we can start writing the tally data to it, this we do using the `rhdf5::h5write` function on each dataset, specifying the target file, location within the file and the exact block of data we are writing to. For example the code `index = list( NULL, NULL, startpos:endpos )` in the command used to write the "Coverages" dataset to the file, specifies that the block of data we are going to write covers all samples (the first `NULL`) on both strands (the second `NULL`) and goes from `startpos` to `endpos` in the genomic position dimension.

```
group <- paste(study, chrom, sep="/")
h5write( tallies$Counts, tallyFile, paste( group, "Counts", sep = "/" ),
         index = list( NULL, NULL, NULL, startpos:endpos ) )
h5write( tallies$Coverages, tallyFile, paste( group, "Coverages", sep = "/" ),
         index = list( NULL, NULL, startpos:endpos ) )
h5write( tallies$Deletions, tallyFile, paste( group, "Deletions", sep = "/" ),
         index = list( NULL, NULL, startpos:endpos ) )
h5write( tallies$Reference, tallyFile, paste( group, "Reference", sep = "/" ),
         index = list( startpos:endpos ) )
h5ls(tallyFile)
```

```
##     group      name       otype  dclass                    dim
## 0       /      NRAS   H5I_GROUP
## 1   /NRAS         1   H5I_GROUP
## 2 /NRAS/1    Counts H5I_DATASET INTEGER 12 x 48 x 2 x 249250621
## 3 /NRAS/1 Coverages H5I_DATASET INTEGER      48 x 2 x 249250621
## 4 /NRAS/1 Deletions H5I_DATASET INTEGER      48 x 2 x 249250621
## 5 /NRAS/1 Reference H5I_DATASET INTEGER               249250621
```

Another important aspect of using tally files is the sample meta-data. Since HDF5 datasets only store matrices without dimension names we need a way of knowing which id in the sample dimension corresponds to which sample and we probably want to keep some type of auxiliary information about each sample as well. We will construct the sample meta-data object (a `data.frame`) manually and use the `setSampleData` function to write is to the tally file. With the `getSampleData` function we can then retrieve the sample meta-data we wrote to the tally file to check if everything worked. To familiarise yourself with the required fields in a sample meta-data object, have a look at `?setSampleData`

```
sampleData <- data.frame(
  Sample = gsub( ".bam", "", gsub( pattern = "ExampleData/", "", bamFiles)),
  Column = seq_along(bamFiles),
  Type = "Control",
  Library = "WholeExome",
```

```
  stringsAsFactors = FALSE
  )
sampleData$SampleID <- sapply(
  strsplit( sampleData$Sample, "\\."),
  function(x) as.numeric(x[2])
  )
sampleData$Type[sampleData$SampleID %% 2 == 0] <- "Case"
sampleData$Patient <- paste0( "Patient", floor( (sampleData$SampleID + 1 ) / 2 ) )
sampleData$Library[sampleData$SampleID >= 37] <- "WholeGenome"
setSampleData( tallyFile, group, sampleData, largeAttributes = TRUE )
getSampleData( tallyFile, group )
```

```
##    Column     Library   Patient       Sample SampleID    Type
## 1       1  WholeExome  Patient1  Sample.1.rh        1 Control
## 2       2  WholeExome  Patient5 Sample.10.rh       10    Case
## 3       3  WholeExome  Patient6 Sample.11.rh       11 Control
## 4       4  WholeExome  Patient6 Sample.12.rh       12    Case
## 5       5  WholeExome  Patient7 Sample.13.rh       13 Control
## 6       6  WholeExome  Patient7 Sample.14.rh       14    Case
## 7       7  WholeExome  Patient8 Sample.15.rh       15 Control
## 8       8  WholeExome  Patient8 Sample.16.rh       16    Case
## 9       9  WholeExome  Patient9 Sample.17.rh       17 Control
## 10     10  WholeExome  Patient9 Sample.18.rh       18    Case
## 11     11  WholeExome Patient10 Sample.19.rh       19 Control
## 12     12  WholeExome  Patient1  Sample.2.rh        2    Case
## 13     13  WholeExome Patient10 Sample.20.rh       20    Case
## 14     14  WholeExome Patient11 Sample.21.rh       21 Control
## 15     15  WholeExome Patient11 Sample.22.rh       22    Case
## 16     16  WholeExome Patient12 Sample.23.rh       23 Control
## 17     17  WholeExome Patient12 Sample.24.rh       24    Case
## 18     18  WholeExome Patient13 Sample.25.rh       25 Control
## 19     19  WholeExome Patient13 Sample.26.rh       26    Case
## 20     20  WholeExome Patient14 Sample.27.rh       27 Control
## 21     21  WholeExome Patient14 Sample.28.rh       28    Case
## 22     22  WholeExome Patient15 Sample.29.rh       29 Control
## 23     23  WholeExome  Patient2  Sample.3.rh        3 Control
## 24     24  WholeExome Patient15 Sample.30.rh       30    Case
## 25     25  WholeExome Patient16 Sample.31.rh       31 Control
## 26     26  WholeExome Patient16 Sample.32.rh       32    Case
## 27     27  WholeExome Patient17 Sample.33.rh       33 Control
## 28     28  WholeExome Patient17 Sample.34.rh       34    Case
## 29     29  WholeExome Patient18 Sample.35.rh       35 Control
## 30     30  WholeExome Patient18 Sample.36.rh       36    Case
## 31     31 WholeGenome Patient19 Sample.37.rh       37 Control
## 32     32 WholeGenome Patient19 Sample.38.rh       38    Case
## 33     33 WholeGenome Patient20 Sample.39.rh       39 Control
## 34     34  WholeExome  Patient2  Sample.4.rh        4    Case
## 35     35 WholeGenome Patient20 Sample.40.rh       40    Case
## 36     36 WholeGenome Patient21 Sample.41.rh       41 Control
## 37     37 WholeGenome Patient21 Sample.42.rh       42    Case
## 38     38 WholeGenome Patient22 Sample.43.rh       43 Control
## 39     39 WholeGenome Patient22 Sample.44.rh       44    Case
## 40     40 WholeGenome Patient23 Sample.45.rh       45 Control
```

```
## 41     41 WholeGenome Patient23 Sample.46.rh       46    Case
## 42     42 WholeGenome Patient24 Sample.47.rh       47 Control
## 43     43 WholeGenome Patient24 Sample.48.rh       48    Case
## 44     44  WholeExome  Patient3  Sample.5.rh        5 Control
## 45     45  WholeExome  Patient3  Sample.6.rh        6    Case
## 46     46  WholeExome  Patient4  Sample.7.rh        7 Control
## 47     47  WholeExome  Patient4  Sample.8.rh        8    Case
## 48     48  WholeExome  Patient5  Sample.9.rh        9 Control
```

Modifying the sample meta-data is facilitated through the use of `getSampleData` and `setSampleData`, we can for example add another column to the `data.frame`.

```
sampleData <- getSampleData( tallyFile, group ) #read from file
sampleData$ClinicalVariable <- rnorm(nrow(x = sampleData)) # add some data
setSampleData( tallyFile, group, sampleData, largeAttributes = TRUE ) # write it back
head(getSampleData( tallyFile, group )) #did it work?
```

```
##   ClinicalVariable Column    Library  Patient       Sample SampleID
## 1           1.4223      1 WholeExome Patient1  Sample.1.rh        1
## 2          -0.6157      2 WholeExome Patient5 Sample.10.rh       10
## 3          -2.9283      3 WholeExome Patient6 Sample.11.rh       11
## 4          -0.6877      4 WholeExome Patient6 Sample.12.rh       12
## 5           0.7190      5 WholeExome Patient7 Sample.13.rh       13
## 6           1.1485      6 WholeExome Patient7 Sample.14.rh       14
##      Type
## 1 Control
## 2    Case
## 3 Control
## 4    Case
## 5 Control
## 6    Case
```

Special attention has to be paid when mixing `largeAttributes = TRUE` and the default of `largeAttributes = FALSE` when using the `setSampleData` function, since they will write the sample meta-data to the tally file in two different ways and the `getSampleData` function will always use the data stored with `largeAttributes = TRUE` if it is present.

```
sampleData <- getSampleData( tallyFile, group ) #read from file
sampleData$Sample <- "********"
head(sampleData)
```

```
##   ClinicalVariable Column    Library  Patient   Sample SampleID    Type
## 1           1.4223      1 WholeExome Patient1 ********        1 Control
## 2          -0.6157      2 WholeExome Patient5 ********       10    Case
## 3          -2.9283      3 WholeExome Patient6 ********       11 Control
## 4          -0.6877      4 WholeExome Patient6 ********       12    Case
## 5           0.7190      5 WholeExome Patient7 ********       13 Control
## 6           1.1485      6 WholeExome Patient7 ********       14    Case
```

```
setSampleData( tallyFile, group, sampleData ) # write it back without largeAttributes=TRUE
sampleData <- getSampleData( tallyFile, group ) #did it work?
head(sampleData) #apparently not
```

```
##   ClinicalVariable Column    Library  Patient       Sample SampleID
## 1           1.4223      1 WholeExome Patient1  Sample.1.rh        1
## 2          -0.6157      2 WholeExome Patient5 Sample.10.rh       10
## 3          -2.9283      3 WholeExome Patient6 Sample.11.rh       11
## 4          -0.6877      4 WholeExome Patient6 Sample.12.rh       12
## 5           0.7190      5 WholeExome Patient7 Sample.13.rh       13
## 6           1.1485      6 WholeExome Patient7 Sample.14.rh       14
##      Type
## 1 Control
## 2    Case
## 3 Control
## 4    Case
## 5 Control
## 6    Case
```

Did we save some space?

Let's have a look at how much smaller the tally file is compared to the input `.bam` files.

```
tallySize <- file.info(tallyFile)$size
bamSize <- sum(sapply(
  list.files("ExampleData/", pattern= "*.bam$", full.names=TRUE),
  function(x) file.info(x)$size )
  )
tallySize / bamSize
```

```
## [1] 0.05772
```

Interacting with the tally file

`h5vc` provides two functions to access data stored in an HDF5 based tally file:

1. `h5readBlock` is a function that can be used to extract a block of data from a tally file; the limits of the block can be specified in the genomic position as well as the sample dimension
2. `h5dapply` is a general purpose function which we can use to apply a function over blocks along the genomic position axis in a tally file; this is useful for creating binned statistics (e.g. GC content or coverage values) as well as applying functions whose results are independent of adjacent positions values (e.g. variant calling functions)

Reading blocks of data

Let us first investigate the usage of the `h5readBlock` function, it has the following parameters:

1. `filename`: the name of the tally file (this must be an absoulte path or a path relative to the current working directory)
2. `group`: the group within the tally file that we want to extract data from (`/NRAS/1` in our case)
3. `names`: the names of the datasets we want to extract (this can be any subset of `c( "Counts", "Coverages", "Deletions", "Reference")`)

4. `dims`: the dimensions along which the datasets shall be subsetted for the block (the defaults select the genomic position but there are cases where another dimension might be used)
5. `range`: the first and last index of the block in the dimension specified in `dims`
6. `samples = NULL`: a subselector for extracting data for specific samples, useful if the analysis is not using cohort-level data
7. `sampleDimMap = .sampleDimMap`: a function that maps dataset names to their respective sample dimensions
8. `verbose = FALSE`: boolean flag that specifies how much info is printed

Question: Try extracting different blocks of data using selectors for samples, datasets and genomic positions to see the effects of all those parameters.

```
startpos <- 115247090
endpos <- 115259515
data <- h5readBlock(
    filename = tallyFile,
    group = group,
    range = c(startpos, endpos)
  )
str(data)
```

```
## List of 5
##  $ Counts      : int [1:12, 1:48, 1:2, 1:12426] 0 0 0 0 0 0 0 0 0 0 ...
##  $ Coverages   : int [1:48, 1:2, 1:12426] 0 0 0 0 0 0 0 1 0 0 ...
##  $ Deletions   : int [1:48, 1:2, 1:12426] 0 0 0 0 0 0 0 0 0 0 ...
##  $ Reference   : int [1:12426(1d)] 3 3 0 3 2 0 1 3 0 0 ...
##  $ h5dapplyInfo:List of 4
##   ..$ Blockstart: num 1.15e+08
##   ..$ Blockend  : num 1.15e+08
##   ..$ Datasets  :'data.frame':   4 obs. of  7 variables:
##   .. ..$ group   : chr [1:4] "/NRAS/1" "/NRAS/1" "/NRAS/1" "/NRAS/1"
##   .. ..$ name    : chr [1:4] "Counts" "Coverages" "Deletions" "Reference"
##   .. ..$ otype   : Factor w/ 7 levels "H5I_FILE","H5I_GROUP",..: 5 5 5 5
##   .. ..$ dclass  : chr [1:4] "INTEGER" "INTEGER" "INTEGER" "INTEGER"
##   .. ..$ dim     : chr [1:4] "12 x 48 x 2 x 249250621" "48 x 2 x 249250621" "48 x 2 x 249250621" "249
##   .. ..$ DimCount: int [1:4] 4 3 3 1
##   .. ..$ PosDim  : int [1:4] 4 3 3 1
##   ..$ Group     : chr "/NRAS/1"
```

```
data <- h5readBlock(
    filename = tallyFile,
    group = group,
    names = c("Coverages", "Reference"),
    range = c(startpos, endpos)
  )
str(data)
```

```
## List of 3
##  $ Coverages   : int [1:48, 1:2, 1:12426] 0 0 0 0 0 0 0 1 0 0 ...
##  $ Reference   : int [1:12426(1d)] 3 3 0 3 2 0 1 3 0 0 ...
```

```
##  $ h5dapplyInfo:List of 4
##   ..$ Blockstart: num 1.15e+08
##   ..$ Blockend  : num 1.15e+08
##   ..$ Datasets  :'data.frame':   2 obs. of  7 variables:
##   .. ..$ group   : chr [1:2] "/NRAS/1" "/NRAS/1"
##   .. ..$ name    : chr [1:2] "Coverages" "Reference"
##   .. ..$ otype   : Factor w/ 7 levels "H5I_FILE","H5I_GROUP",..: 5 5
##   .. ..$ dclass  : chr [1:2] "INTEGER" "INTEGER"
##   .. ..$ dim     : chr [1:2] "48 x 2 x 249250621" "249250621"
##   .. ..$ DimCount: int [1:2] 3 1
##   .. ..$ PosDim  : int [1:2] 3 1
##   ..$ Group     : chr "/NRAS/1"
```

```
data <- h5readBlock(
    filename = tallyFile,
    group = group,
    names = c("Coverages", "Reference"),
    samples = sampleData$Sample[3:7],
    range = c(startpos, endpos)
  )
str(data)
```

```
## List of 3
##  $ Coverages   : int [1:5, 1:2, 1:12426] 0 0 0 0 0 0 0 0 0 0 ...
##  $ Reference   : int [1:12426(1d)] 3 3 0 3 2 0 1 3 0 0 ...
##  $ h5dapplyInfo:List of 4
##   ..$ Blockstart: num 1.15e+08
##   ..$ Blockend  : num 1.15e+08
##   ..$ Datasets  :'data.frame':   2 obs. of  7 variables:
##   .. ..$ group   : chr [1:2] "/NRAS/1" "/NRAS/1"
##   .. ..$ name    : chr [1:2] "Coverages" "Reference"
##   .. ..$ otype   : Factor w/ 7 levels "H5I_FILE","H5I_GROUP",..: 5 5
##   .. ..$ dclass  : chr [1:2] "INTEGER" "INTEGER"
##   .. ..$ dim     : chr [1:2] "48 x 2 x 249250621" "249250621"
##   .. ..$ DimCount: int [1:2] 3 1
##   .. ..$ PosDim  : int [1:2] 3 1
##   ..$ Group     : chr "/NRAS/1"
```

The `data` object that are returned are lists of arrays with one entry per requested dataset, the layout is essentially the same as the results of a call to `tallyBAM` or `applyTallies` would have yielded. In addition there is always the special slot named `h5dapplyInfo` which contains a list specifying the start and end of the current block as well as a `data.frame` with information about the returned datasets and the group they were extracted from.

While extracting a single block of data can be interesting for the investigation of specific regions within specific samples, when we want to calculate more general statistics the `h5dapply` function is of more use. The interface of the function is essentially identical to that of `h5readBlock` and only contains two additional paramters, which are the blocksize and the function to apply.

An example of usage is this bit of code, that calculates binned coverage for all the samples. Note that they are whole exome sequencing data and therefore the binned coverage signal is heavily influenced by the presence or absence of coding sequence within each bin. You can tell which samples are whole exome and which are whole genome simply by looking at the numbers below.

```
startpos = 115200000
endpos   = 115300000
data <- h5dapply( # extracting coverage binned at 1000 bases
    filename = tallyFile,
    group = group,
    blocksize = 1000,
    FUN = binnedCoverage,
    sampledata = sampleData,
    gccount = TRUE,
    names = c( "Coverages", "Reference" ),
    range = c(startpos, endpos)
  )
  data <- do.call(rbind, data)
  rownames(data) <- NULL
  head(data)
```

```
##   Sample.1.rh Sample.10.rh Sample.11.rh Sample.12.rh Sample.13.rh
## 1         395            0          201          300          197
## 2         921          716          899         1109          737
## 3         770         1157          932          674          811
## 4         414          800          542          606          343
## 5           0            0          177          186          385
## 6        1203          594          597          813          369
##   Sample.14.rh Sample.15.rh Sample.16.rh Sample.17.rh Sample.18.rh
## 1          605          396          200          563            0
## 2          473          404          773          812          337
## 3         1856          592          231          797         1005
## 4         1580          596          572          605          840
## 5          278           14            0            0            0
## 6         1108          969          363         1507          490
##   Sample.19.rh Sample.2.rh Sample.20.rh Sample.21.rh Sample.22.rh
## 1          307           0          400          388          458
## 2          322         621         1060          940          733
## 3          396         510         1029          497            0
## 4          535        1356          675          760          321
## 5           88         101          202            0            0
## 6          858        1211         1260         1091          197
##   Sample.23.rh Sample.24.rh Sample.25.rh Sample.26.rh Sample.27.rh
## 1          169          197          235          560          355
## 2          327          383          513         1338         1002
## 3          744          987         1107          975          239
## 4          742          400         1308         1231          846
## 5          397          191          389            0            0
## 6          701          347          389          712          394
##   Sample.28.rh Sample.29.rh Sample.3.rh Sample.30.rh Sample.31.rh
## 1          202          397         659            0          404
## 2          611          610        2302          805          598
## 3          300          628         575         1007          621
## 4          371          786        1508          601         1129
## 5            0           66           0          202          202
## 6          805          337         403         1008          798
##   Sample.32.rh Sample.33.rh Sample.34.rh Sample.35.rh Sample.36.rh
## 1          604          217          303          495          426
```

```
## 2          362         1390          218          296          862
## 3          199          786         1826         1091          850
## 4          558          720         1289         1109          600
## 5           31          404          202          213          602
## 6         1331          604          804          791          403
##   Sample.37.rh Sample.38.rh Sample.39.rh Sample.4.rh Sample.40.rh
## 1        28603        27125        56168         681        59472
## 2        27959        24008        49592         605        51395
## 3        20460        20195        38953         591        40939
## 4        23075        19342        47689        1333        40777
## 5        26071        28562        63268         202        51423
## 6        21846        29644        54014        1610        51261
##   Sample.41.rh Sample.42.rh Sample.43.rh Sample.44.rh Sample.45.rh
## 1        67059        55770        52634        43861        54070
## 2        58227        44666        53105        33861        54047
## 3        42877        41245        47637        33508        37812
## 4        41228        32405        40293        35153        51363
## 5        75804        49853        52495        38889        53600
## 6        56492        50890        49215        34889        49276
##   Sample.46.rh Sample.47.rh Sample.48.rh Sample.5.rh Sample.6.rh
## 1        45876        43447        40732         238         198
## 2        48049        37404        39476         402           0
## 3        38221        38956        39059         534         143
## 4        44523        38318        34201        1075         120
## 5        45096        40383        40283           0          99
## 6        47027        40833        40850         804         438
##   Sample.7.rh Sample.8.rh Sample.9.rh     Start       End GCCount
## 1         390         296         401 115200000 115200999     389
## 2         137         614         169 115201000 115201999     373
## 3         977         300        1366 115202000 115202999     427
## 4         542         866         201 115203000 115203999     424
## 5           0         333         200 115204000 115204999     315
## 6        1101         594         773 115205000 115205999     369
```

Upon investigating the signature of the `binnedCoverage` function we can see that certain parameters must always be present in the signature of a function to be used with `h5dapply`, i.e. the `data` parameter is always the same with the name `data` and the structure of a `list` of entries with one slot per dataset.

```
binnedCoverage
```

```
## function (data, sampledata, gccount = FALSE)
## {
##     ret = rowSums(data$Coverages[sampledata$Column, , ])
##     names(ret) = sampledata$Sample[order(sampledata$Column)]
##     ret = as.data.frame(t(ret))
##     ret$Start = data$h5dapplyInfo$Blockstart
##     ret$End = data$h5dapplyInfo$Blockend
##     if (gccount) {
##         stopifnot("Reference" %in% names(data))
##         ret$GCCount <- sum(table(data$Reference)[c("1", "2")])
##     }
##     ret
## }
## <environment: namespace:h5vc>
```

Since we have some whole genome samples available within the datasets, we will re-run the `binnedCoverage` function only on those samples since we can make more reliable fits to GC content in them. At this point the subsetting by sample functionality build into `h5dapply` is helpful. Using this subsetting approach reduces time and memory requirements, since only the corresponding data is loaded from the HDF5 file. We do have to adapt the sampleData a bit, since the positions of the samples in the respective sample dimension are shifted by the subsetting.

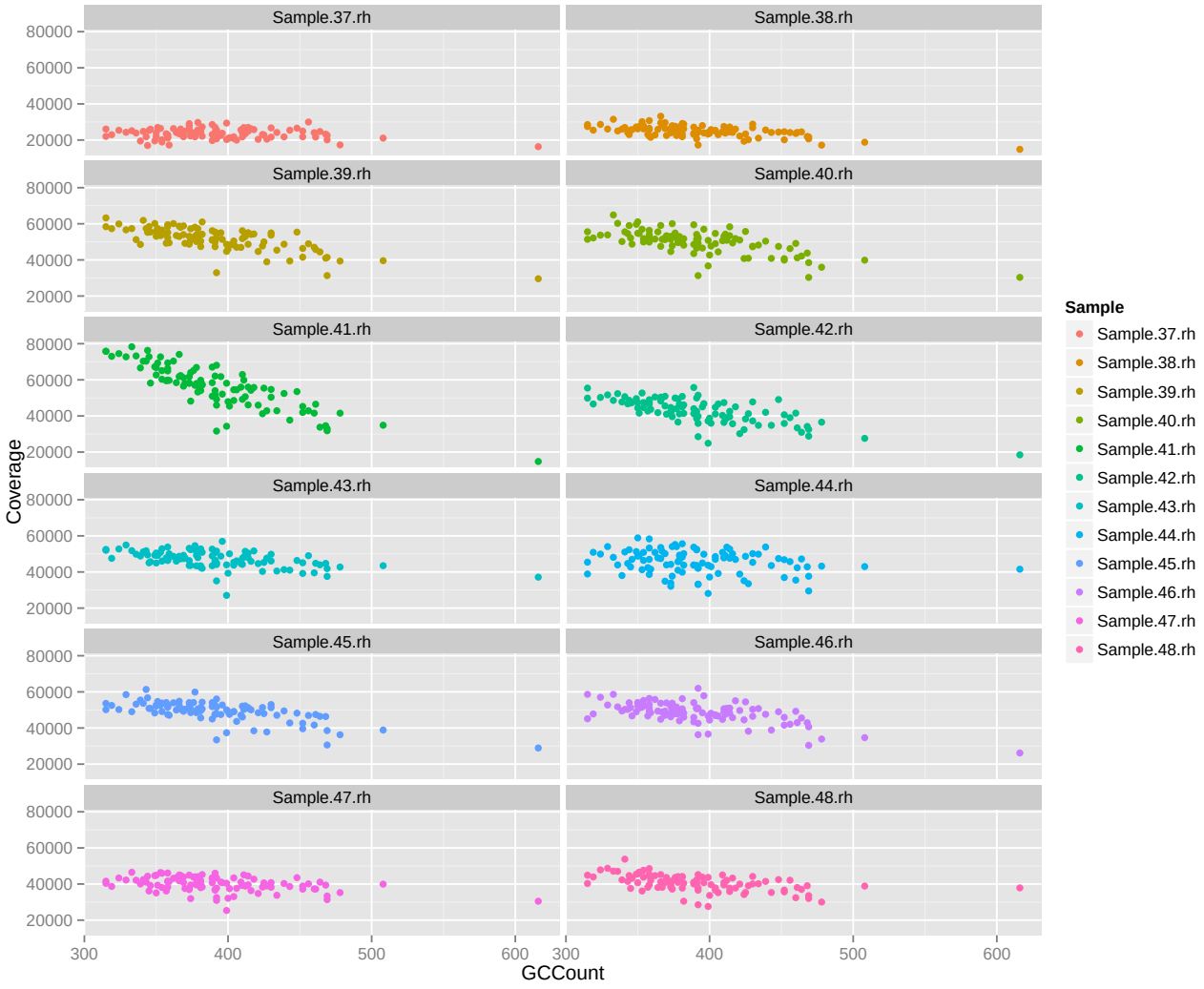
```
# Grab only the whole genome samples
selectedSamples <- sampleData$Sample[sampleData$Library == "WholeGenome"]
sampleDataSubset <- subset( sampleData, Sample %in% selectedSamples)
# The Column propertie specifies the position of the sample in the datasets,
# since we subset, the indexes are off and need to be adjusted
sampleDataSubset$Column <- rank(sampleDataSubset$Column)
# Apply the binnedCoverage function
data <- h5dapply( # extracting coverage binned at 1000 bases
    filename = tallyFile,
    group = group,
    blocksize = 1000,
    samples = selectedSamples,
    FUN = binnedCoverage,
    sampledata = sampleDataSubset,
    gccount = TRUE,
    names = c( "Coverages", "Reference" ),
    range = c(startpos, endpos)
  )
# h5dapply returns a list, we need to merge the results by row
data <- do.call(rbind, data)
rownames(data) <- NULL
head(data)
```

```
##   Sample.37.rh Sample.38.rh Sample.39.rh Sample.40.rh Sample.41.rh
## 1        28603        27125        56168        59472        67059
## 2        27959        24008        49592        51395        58227
## 3        20460        20195        38953        40939        42877
## 4        23075        19342        47689        40777        41228
## 5        26071        28562        63268        51423        75804
## 6        21846        29644        54014        51261        56492
##   Sample.42.rh Sample.43.rh Sample.44.rh Sample.45.rh Sample.46.rh
## 1        55770        52634        43861        54070        45876
## 2        44666        53105        33861        54047        48049
## 3        41245        47637        33508        37812        38221
## 4        32405        40293        35153        51363        44523
## 5        49853        52495        38889        53600        45096
## 6        50890        49215        34889        49276        47027
##   Sample.47.rh Sample.48.rh     Start       End GCCount
## 1        43447        40732 115200000 115200999     389
## 2        37404        39476 115201000 115201999     373
## 3        38956        39059 115202000 115202999     427
## 4        38318        34201 115203000 115203999     424
## 5        40383        40283 115204000 115204999     315
## 6        40833        40850 115205000 115205999     369
```

This simple function returns a dataset with one column per sample as well as the gc-count in each bin. This is a pretty good starting point for a copy number analysis using e.g. the HMMCopy package. We can for

example plot the coverages in bins vs. their GC count to check for biases.

```
library(ggplot2)
library(reshape2)
data <- melt(data, id.vars = c("Start", "End", "GCCount"))
colnames(data)[4] <- "Sample"
colnames(data)[5] <- "Coverage"
p <- ggplot(data, aes(x=GCCount, y = Coverage, col = Sample)) +
  geom_point() + facet_wrap(~ Sample, ncol = 2)
print(p)
```



As we can see here, there are some differences between the samples (e.g. Sample.41 seems to have the strongest coverage-dependency on GC), in general most samples look rather flat, which indicates little influence of the GC content on the local coverage. If we wanted to we could now use e.g. `locfit.robust` to fit a model `GCCount ~ Coverage` for each sample and obtain a correction factor for each possible value of `GCCount` for each sample, which we could use to correct for GC-dependencies, which otherwise could generate noise in our log2-ratios between the matched normal and control samples, which we might want to use to produce copy number calls.

> **Question:** I will leave the GC-dependency fitting, correction and calculation of log2-ratios as an exercise for the reader. **Hint:** have a look at `?locfit.robust` after calling `library(locfit)` as well as `?predict`

There is only one requirement on the `FUN` argument to `h5dapply` (i.e. the function we want to apply to each block). We can use any function as `FUN` that expects its first argument to be a list of arrays corresponding to the datasets in the current block. Further arguments are passed through by `h5dapply` but their names may not collide with the named arguments of `h5dapply` (filename, group, blocksize, FUN, names, dims, range, samples, sampleDimMap, verbose and BPPARAM).

With the `BPPARAM` parameter we can control parallelisation behaviour of `h5dapply`, which can yield substantial runtime improvements, as exemplified below (note that this code uses 4 concurrent processes for the calculations, if you are operating on a dual-core machine set this to 2, if you are on a bigger machine, increase the number of workers accordingly).

In this simple benchmark we will use `system.time` to measure the runtime of a single call to `h5dapply`.

```
startpos = 115200000
endpos   = 115300000
system.time(
  dataS <- h5dapply( # extracting coverage binned at 1000 bases
    filename = tallyFile,
    group = group,
    blocksize = 1000,
    FUN = binnedCoverage,
    sampledata = sampleData,
    gccount = TRUE,
    names = c( "Coverages", "Reference" ),
    range = c(startpos, endpos)
  )
)
```

```
##    user  system elapsed
##   5.472   1.071   6.590
```

```
dataS <- do.call(rbind, dataS)
system.time(
  dataP <- h5dapply( # extracting coverage binned at 1000 bases
    filename = tallyFile,
    group = group,
    blocksize = 1000,
    FUN = binnedCoverage,
    sampledata = sampleData,
    gccount = TRUE,
    names = c( "Coverages", "Reference" ),
    range = c(startpos, endpos),
    BPPARAM = MulticoreParam(workers = 4)
  )
)
```

```
##    user  system elapsed
##   3.899   1.054   2.928
```

```
dataP <- do.call(rbind, dataP)
# Are results of serial and parallel execution identical?
identical(dataS, dataP)
```

```
## [1] TRUE
```

Calling variants

As an example of how to develop new functions for usage with `h5dapply` we will now implement a naive, single-sample variant calling function. The function will be called `myVariantCaller` and will take four arguments:

1. `data`: the list of arrays corresponding to the current block
2. `sampleNames`: a vector of names to be used for the samples in the same order as the samples are stored in the sample dimension of the nucleotide tally.
3. `minAF`: minimal allelic frequency a variant should have to be called
4. `minCov`: minimal coverage a position must have in order for a variant to be called there

Our implementation is exceedingly naive, in that it simply adds up all mismatches per sample irrespective of the alternative allele and then calculates the allelic frequency from that. We then build a filter on the coverage and allelic frequency and return all positions which fulfill our two criteria (allelic freqeuncy and coverage minimum).

```
myVariantCaller <- function( data, sampleNames, minAF = 0.2, minCov = 20 ){
  #sanity checks for the data
  stopifnot("Counts" %in% names(data))
  stopifnot("Coverages" %in% names(data))
  #sum up coverages for each sample (forward strand + reverse strand coverage)
  coverages <- apply(data$Coverages, c(1,3), sum)
  # sum up all counts for each sample (i.e. #A + #C + #G + #T on both strands)
  counts <- apply(data$Counts, c(2,4), sum)
  # Now counts and coverage have a compatible shape
  # and we can calculate estimates for the allelic frequency
  afs <- counts / coverages
  # Here we do the calling, by filtering for a minimal allelic Frequency and
  # Coverage in each sample and position
  filter <- afs >= minAF & coverages >= minCov
  require(reshape2) #use melt to convert the array into a data.frame
  filter <- melt(filter)
  #rename the columns
  colnames(filter) <- c("Sample", "Position", "Called")
  filter$Sample <- sampleNames[filter$Sample]
  # fix the position to reflect genomic coordinates
  filter$Position <- filter$Position + data$h5dapplyInfo$Blockstart - 1
  # We return only those positions that pass the filter
  return(subset(filter, Called == TRUE))
}
```

We will now apply this variant caller to the whole genome sequencing samples we had extracted earlier.

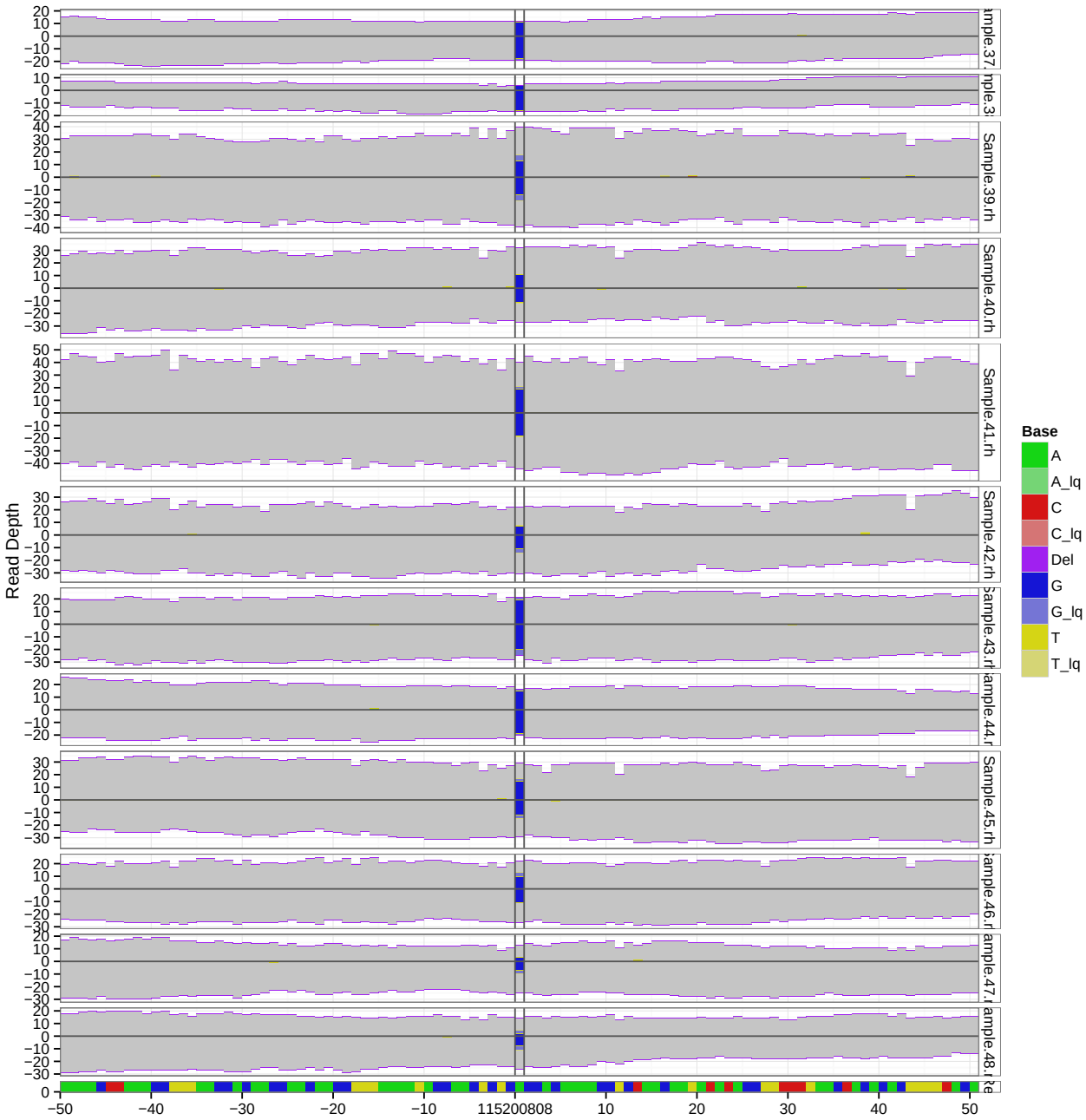
```
selectedSamples <- sampleData$Sample[sampleData$Library == "WholeGenome"]
sampleDataSubset <- subset( sampleData, Sample %in% selectedSamples)
sampleDataSubset$Column <- rank(sampleDataSubset$Column)
data <- h5dapply( # calling variants in bins of size 10kb
    filename = tallyFile,
    group = group,
    blocksize = 10000,
```

```
    samples = selectedSamples,
    FUN = myVariantCaller,
    sampleNames = selectedSamples,
    names = c( "Coverages", "Counts" ),
    range = c(startpos, endpos)
  )
data <- do.call(rbind, data)
rownames(data) <- NULL
head(data)
```

```
##         Sample  Position Called
## 1 Sample.37.rh 115200808   TRUE
## 2 Sample.38.rh 115200808   TRUE
## 3 Sample.39.rh 115200808   TRUE
## 4 Sample.40.rh 115200808   TRUE
## 5 Sample.41.rh 115200808   TRUE
## 6 Sample.42.rh 115200808   TRUE
```

Let's have a look at the first positions where a variant has been called. We extract the nucleotide tally data around that position with a call to `h5readBlock` and use it to create a `mismatchPlot`, which gives us an overview over the region in the different samples.

```
varpos <- data$Position[1]
windowsize <- 50
plotData <- h5readBlock(
    filename = tallyFile,
    group = group,
    samples = selectedSamples,
    range = c(varpos - windowsize, varpos + windowsize)
  )
p <- mismatchPlot(
  data = plotData,
  sampledata = sampleDataSubset,
  position = varpos, windowsize = windowsize)
print(p)
```



We can see a nice heterozygous variant (actually homozygous in some of the patients) that we found at this position (not bad for a variant caller written in 5 minutes), but we can also see that it occurs always in both samples of all patients (the sample pairs are in order always control and case) and that it is therefore not a very interesting variant in the current setting (a cohort of paired cancer samples). Maybe we need a smarter approach to finding the interesting positions.

Question: I strongly recommend the user to think about how to implement a pairwise comparative variant callign scheme and experiment with adapting the `myVariantCaller` function to add comparative calling and more importantly calling of the alternative allele.

Question: Familiarise yourself with the different step of the variant caller you just implemented. Load some data with a call to `h5readBlock`, store it in an object that you name `data` and try out some of the calls, e.g. `coverages <- apply(data$Coverages, c(1,3), sum)` and `counts`

`<- apply(data$Counts, c(2,4), sum)` as well as the filter call (`filter <- afs >= minAF & coverages >= minCov`). Also have a look at `?melt` to find out what this function does.

Question: Modify the `myVariantCaller` function so that is uses `binom.test` to give a p-Value for each variant call testing for an assumed allelix freqeuncy of `0.5` (i.e. heterozygousity in diploid regions)

An example implementation of a comparative SNV caller is provided in the `h5vc::callVariantsPaired` function, which we can run on the whole dataset with the following code:

```
startpos = 115200000
endpos   = 115300000
data <- h5dapply( # calling variants pairwise in bins of size 5kb
    filename = tallyFile,
    group = group,
    blocksize = 10000,
    FUN = callVariantsPaired,
    sampledata = sampleData,
    cl = vcConfParams(
      returnDataPoints = TRUE,
      annotateWithBackground = TRUE
      ),
    names = c( "Coverages", "Counts", "Reference" ),
    range = c(startpos, endpos),
    verbose = TRUE
  )
```

```
## Processing Block #1: 115200000:115209999
## Processing Block #2: 115210000:115219999
## Processing Block #3: 115220000:115229999
## Processing Block #4: 115230000:115239999
## Processing Block #5: 115240000:115249999
## Processing Block #6: 115250000:115259999
## Processing Block #7: 115260000:115269999
## Processing Block #8: 115270000:115279999
## Processing Block #9: 115280000:115289999
## Processing Block #10: 115290000:115299999
```

```
data <- do.call(rbind, data)
rownames(data) <- NULL
head(data)
```

```
##   Chrom     Start       End       Sample altAllele refAllele caseCountFwd
## 1     1 115256529 115256529 Sample.28.rh         C         T            9
## 2     1 115269674 115269674 Sample.30.rh         A         C            2
##   caseCountRev caseCoverageFwd caseCoverageRev controlCountFwd
## 1           22              55              83               0
## 2            2             108             114               0
##   controlCountRev controlCoverageFwd controlCoverageRev
## 1               0                 50                85
## 2               0                101               103
##   backgroundFrequencyFwd backgroundFrequencyRev pValueFwd pValueRev
## 1              0.0000000                      0  0.000000         0
## 2              0.0007692                      0  0.003239         0
```

It's behaviour is controlled through the parameter `cl` which is a simple `list` of configuration parameters. The function `vcConfParams` is the default constructor for that list and a look at it's parameters reveals the underlying concepts in the variant calling function.

```
vcConfParams()
```

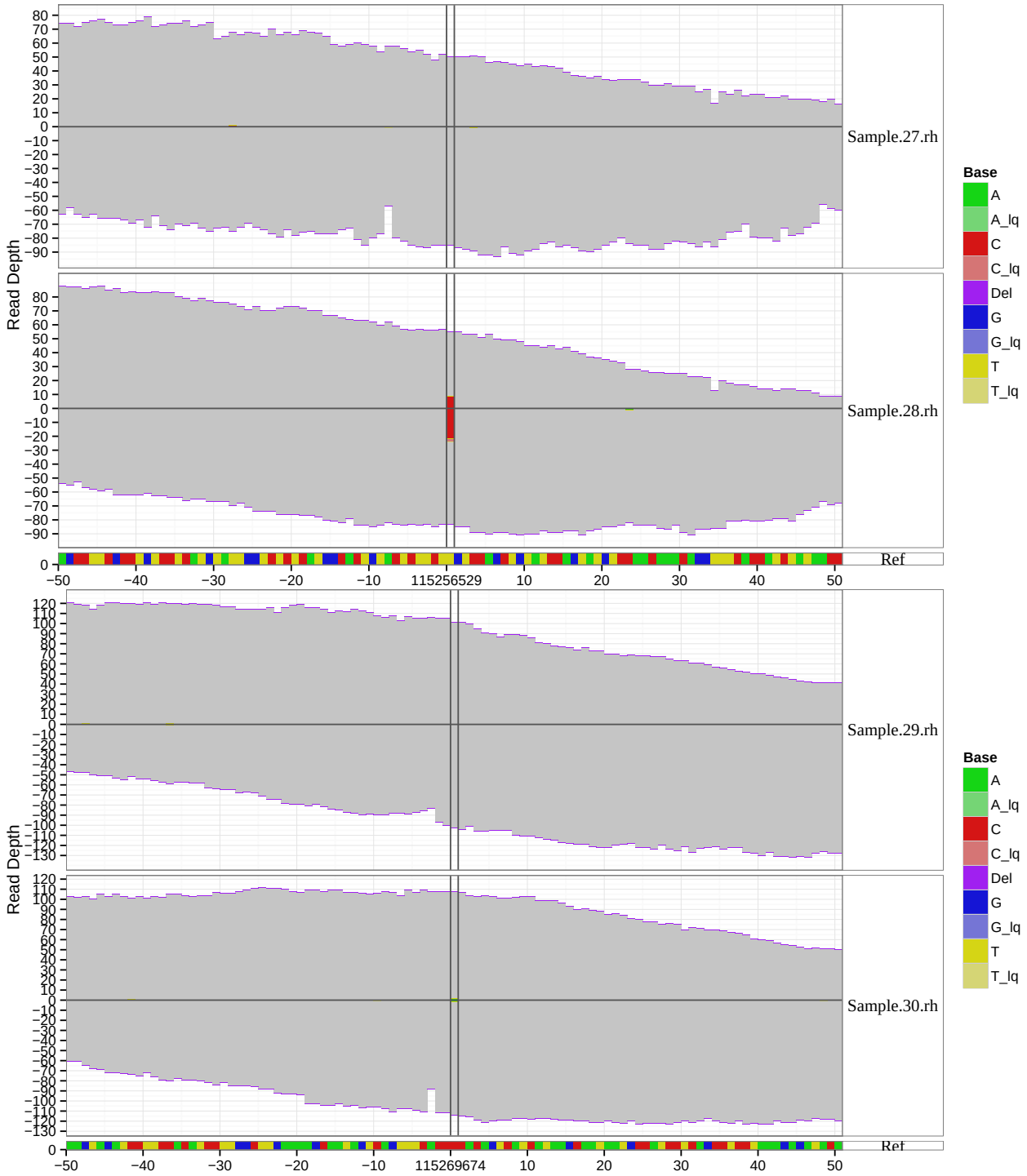
```
## $minStrandCov
## [1] 5
##
## $maxStrandCov
## [1] 200
##
## $minStrandAltSupport
## [1] 2
##
## $maxStrandAltSupportControl
## [1] 0
##
## $minStrandDelSupport
## [1] 2
##
## $maxStrandDelSupportControl
## [1] 0
##
## $minStrandCovControl
## [1] 5
##
## $maxStrandCovControl
## [1] 200
##
## $bases
## [1] 5 6 7 8
##
## $returnDataPoints
## [1] FALSE
##
## $annotateWithBackground
## [1] FALSE
##
## $mergeCalls
## [1] FALSE
##
## $mergeAggregator
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x7f89e0e18790>
## <environment: namespace:base>
##
## $pValueAggregator
## function (..., na.rm = FALSE)  .Primitive("max")
```

For a detailed description see `?vcConfParams`. It should be noted that the `callVariantsPaired` function is really just an example of how I go about calling variants. If you have your own ideas for how to call variants, you can implement those easily in a similar function to `myVariantCaller` or `callVariantsPaired`.

Let's plot the variants we have detected with the comparative variant calling approach. Here we also make use of the fact that `mismatchPlot` returns a `ggplot` plot object which can be further manipulated by addition of layers or themes (the usage of `ggplot2` will not be part of this tutorial; have a look at `?theme` for an overview of how to style the plots according to your wishes, e.g. have a look at what happens when you add `theme(panel.background = element_rect(color = "red"))` to the plot object `p`).

> **Question:** After runnign the following code you will find the object `p` in your R environment. Look at `?theme` as well as `?element_rect` and `?element_text` to find out about some of the ways in which you can modify the plot by changing font sizes, angles and colours.

```
windowsize = 50
plots = list()
for(idx in seq(length=nrow(data))){
  varpos <- data$Start[idx]
  sample <- data$Sample[idx]
  selectedSamples <- subset(
    sampleData,
    Patient == sampleData$Patient[sampleData$Sample == data$Sample[idx]]
    )$Sample
  sampleDataSubset <- subset( sampleData, Sample %in% selectedSamples)
  sampleDataSubset$Column <- rank(sampleDataSubset$Column)
  plotData <- h5readBlock(
    filename = tallyFile,
    group = group,
    samples = selectedSamples,
    range = c(varpos - windowsize, varpos + windowsize)
  )
  p <- mismatchPlot(
    data = plotData,
    sampledata = sampleDataSubset,
    position = varpos,
    windowsize = windowsize)
  plots[[idx]] <- p
  print(p + theme(strip.text.y = element_text(family="serif", angle = 0, size = 12)))
}
```



We have found two variants and we can see that while the first looks promising, the second is actually present at a very low frequency. This doesn't make it any less interesting, just more likely to be a clonal variant present in a small fraction of the cell population that was sequenced. We can see this also by looking at the variant call table again.

```
data$Support <- data$caseCountFwd + data$caseCountRev
data$Coverage <- data$caseCoverageFwd + data$caseCoverageRev
data$AllelicFrequency <- data$Support / data$Coverage
```

```
data
```

```
##   Chrom     Start       End       Sample altAllele refAllele caseCountFwd
## 1     1 115256529 115256529 Sample.28.rh         C         T            9
## 2     1 115269674 115269674 Sample.30.rh         A         C            2
##   caseCountRev caseCoverageFwd caseCoverageRev controlCountFwd
## 1           22              55              83               0
## 2            2             108             114               0
##   controlCountRev controlCoverageFwd controlCoverageRev
## 1               0                 50                85
## 2               0                101               103
##   backgroundFrequencyFwd backgroundFrequencyRev pValueFwd pValueRev
## 1              0.0000000                      0  0.000000         0
## 2              0.0007692                      0  0.003239         0
##   Support Coverage AllelicFrequency
## 1      31      138          0.22464
## 2       4      222          0.01802
```

> **Question:** Think about how you could annotate genomic regions with an estimate of their local density of mismatches per sample using a list of positions (e.g. some variant calls) and the `h5readBlock` function to retrieve data located at and around those positions. **Hint:** A combination of matrix operations on the data can help you get an idea of how many mismatches are present in the blocks in each of the samples.

Creating variant reports

Using `ReportingTools` we can create beautiful interactive HTML reports that contain our list of variant calls and links to usefull other information (e.g. ENSEMBL gene entries, `mismatchPlots`, etc.). An example is given below.

```
library(ReportingTools)
```

```
## Loading required package: knitr
## Loading required package: DBI
## Loading required package: AnnotationDbi
## Loading required package: Biobase
## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with
##     'browseVignettes()'. To cite Bioconductor, see
##     'citation("Biobase")', and for packages 'citation("pkgname")'.
##
##
## Attaching package: 'AnnotationDbi'
##
## The following object is masked from 'package:BSgenome':
##
##     species
##
##
##
##
```

```
## Attaching package: 'ReportingTools'
##
## The following object is masked from 'package:Rsamtools':
##
##     path
```

```
library(hwriter)

for(idx in seq_len(length(plots))){
  png(paste("Var", idx, "png", sep="."), 1200, 800)
  print(plots[[idx]])
  dev.off()
}

reportData <- data
reportData$mismatchPlot <- paste0(
  "<a target=\"_blank\" href=\"Var.",
  seq(length=nrow(data)),
  ".png\">Link</a>" ) # add a link to the variant plot files
htmlRep <- HTMLReport(shortName = "VariantReport", baseDirectory = getwd())
publish( reportData, htmlRep ) # publish the data.frame to the HTML reports
finish(htmlRep)
```

```
## [1] "./VariantReport.html"
```

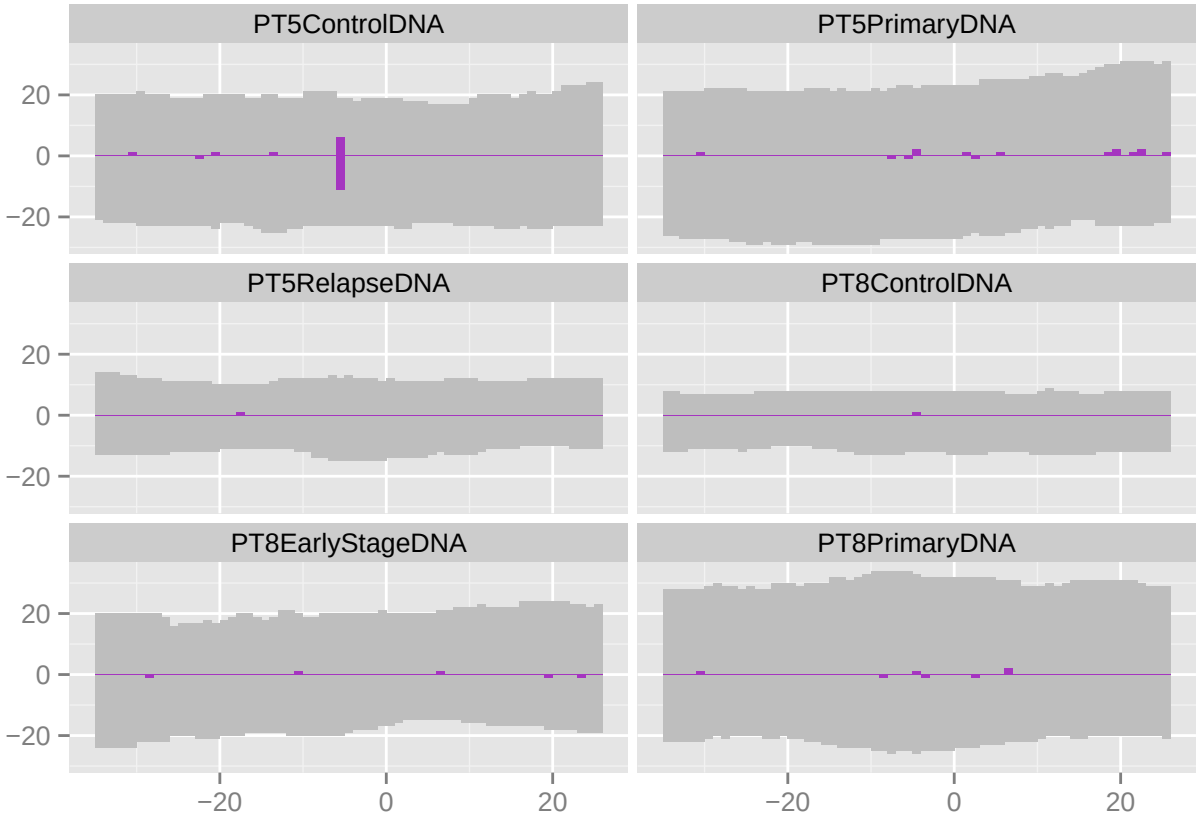
Question: Open the VariantReport.html file in a web browser and have a look around the report.

Creating custom plots

We can generate our own plots using the `geom_h5vc` function to add layers to a `ggplot2` plot object. In the following example we use the example file provided by the `h5vcData` package to plot overviews of total number of mismatches.

```
tallyFile <- system.file( "extdata", "example.tally.hfs5", package = "h5vcData" )
sampleData <- getSampleData( tallyFile, "/ExampleStudy/16" )
position <- 29979629
windowsize <- 30
samples <- sampleData$Sample[sampleData$Patient == "Patient8"]
data <- h5readBlock(
  filename = tallyFile,
  group = "/ExampleStudy/16",
  names = c("Coverages", "Counts"),
  range = c(position - windowsize, position + windowsize)
)
# Summing up all mismatches irrespective of the alternative allele
data$CountsAggregate = colSums(data$Counts)
# Simple overview plot showing number of mismatches per position
p <- ggplot() +
geom_h5vc( data=data,
           sampledata=sampleData,
           windowsize = 35, position = 500,
           dataset = "Coverages", fill = "gray" ) +
```

```
geom_h5vc( data=data,
           sampledata=sampleData,
           windowsize = 35, position = 500,
           dataset = "CountsAggregate", fill = "#A535C0" ) +
facet_wrap( ~ Sample, ncol = 2 )
print(p)
```



Summary

In this tutorial we have learned how to read and write from HDF5 files, create nucleotide tallies from `.bam` files and write them to a tally file. We introduced the concept of sample meta-data and how to retrieve, modify and store it in a nucleotide tally file and explored ways of fetching data from nucleotide tally files and applying functions to the data in blocks along the genome. You should be able to use variant calling functions and indeed be able to implement such functions yourself, using whichever calling aproach you are interested in. Ww also covered the visualisation features that allow us to create overview plots of genomic regions of interest, e.g. the regions around variant calls.

> **Question:** If you still have time and motivation, go ahead and have a look at the example data in different ways, to see what you can find. You could start with running the `callVariantsSingle` and `callDeletionsPaired` function of the region to see how the different samples behave.

> **Question:** You can also work through the simgle genome browser vignette (from the `h5vc` package), adapting it to use the `NRAS.tally.hfs5` tally file that we created with this tutorial.