CrispRVariants User Guide

Helen Lindsay, Mark Robinson

7 October 2015

Contents

Introduction **2**

Case study: Analysis of ptena mutant spectrum in zebrafish **2**

Convert AB1-format Sanger sequences to FASTQ . . . 2

Map the FASTQ reads . . . 3

Read in BAM files and initialize CrisprSet object . . . 3

Creating a CrisprSet . . . 5

Creating summary plots of variants . . . 6

Calculating the mutation efficiency . . . 8

Plot chimeric alignments . . . 9

Changing the appearance of plots **10**

Filtering data in plotVariants . . . 10

plotAlignments . . . 15

Insertion symbols . . . 15

Whitespace between rows . . . 17

Box around guide . . . 18

Text sizes . . . 19

Box around PAM . . . 20

plotFreqHeatmap . . . 21

Controlling the data plotted . . . 22

Changing colours of x-labels . . . 23

Controlling the appearance of the legend . . . 24

barplotAlleleFreqs . . . 25

Other modifications . . . 28

Using CrispRVariants plotting functions independently **29**

Plot the reference sequence . . . 29

Introduction

The CRISPR-Cas9 system is an efficient method of introducing mutations into genomic DNA. A guide RNA directs nuclease activity to a 20 nucleotide target region, resulting in efficient mutagenesis. Repair of the cleaved DNA can introduce insertions and deletions centred around the cleavage site. Once the target sequence is mutated, the guide RNA will no longer bind and the DNA will not be cleaved again. SNPs within the target region, depending on their location, may also disrupt cleavage. The efficiency of a CRISPR-Cas9 experiment is typically measured by amplifying and sequencing the region surrounding the target sequence, then counting the number of sequenced reads that have insertions and deletions at the target site. The **CrispRVariants** package formalizes this process and takes care of various details of managing and manipulating data for such confirmatory and exploratory experiments.

This guide shows an example illustrating how raw data is preprocessed and mapped and how mutation information is extracted relative to the reference sequence. The package comprehensively summarizes and plots the spectrum of variants introduced by CRISPR-Cas9 or similar genome editing experiments.

Case study: Analysis of ptena mutant spectrum in zebrafish

The data used in this case study is from the Mosimann laboratory, UZH.

Convert AB1-format Sanger sequences to FASTQ

This data set is from 5 separate clutches of fish (1 control - uninjected, 2 injected with strong phenotype, 2 injected with mild phenotype), with injections from a guide against the *ptena* gene. For this exercise, the raw data comes as AB1 (Sanger) format. To convert AB1 files to FASTQ, we use ab1ToFastq(), which is a wrapper for functions in **sangerseqR** package with additional quality score trimming.

Although there are many ways to organize such a project, we organize the data (raw and processed) data into a set of directories, with a directory for each type of data (e.g., 'ab1' for AB1 files, 'fastq' for FASTQ files, 'bam' for BAM files, etc.); this can continue with directories for scripts, for figures, and so on. With this structure in place, the following code sets up various directories.

```
library(CrispRVariants)
library(sangerseqR)

# List AB1 filenames, get sequence names,  make names for the fastq files
# Note that we only include one ab1 file with CrispRVariants because
# of space constraints.  All bam files are included

data_dir <- system.file(package="CrispRVariants", "extdata/ab1/ptena")
fq_dir <- tempdir()
ab1_fnames <- dir(data_dir, "ab1$", recursive=TRUE, full=TRUE)
sq_nms <- gsub(".ab1","",basename(ab1_fnames))

# Replace spaces and slashes in filename with underscores
fq_fnames  <- paste0(gsub("[\ |\\/]", "_", dirname(ab1_fnames)), ".fastq")

# abifToFastq to read AB1 files and write to FASTQ
dummy <- mapply( function(u,v,w) {
        abifToFastq(u,v,file.path(fq_dir,w))
}, sq_nms, ab1_fnames, fq_fnames)
```

We will collect sequences from each embryo into the same FASTQ file. Note that abifToFastq *appends* output to existing files where possible. In this experiment, there are 1 sequences, which will be output to 1 files:

```
length(unique(ab1_fnames))
```

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

```
length(unique(fq_fnames))
```

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

Some of the AB1 files may not have a sufficient number of bases after quality score trimming (default is 20 bases). In these cases, abifToFastq() issues a warning (suppressed here).

Map the FASTQ reads

We use FASTQ format because it is the major format used by most genome alignment algorithms. At this stage, the alignment *could* be done outside of R (e.g., using command line tools), but below we use R and a call to system() to keep the whole workflow within R. Note that this also requires various software tools (e.g., **bwa**, **samtools**) to already be installed.

The code below iterates through all the FASTQ files generated above and aligns them to a pre-indexed genome.

```
library("Rsamtools")

# BWA indices were generated using bwa version 0.7.10
bwa_index <- "GRCHz10.fa.gz"
bam_dir <- system.file(package="CrispRVariants", "extdata/bam")
fq_fnames <- file.path(fq_dir,unique(fq_fnames))
bm_fnames <- gsub(".fastq$",".bam",basename(fq_fnames))
srt_bm_fnames <- file.path(bam_dir, gsub(".bam","_s",bm_fnames))

# Map, sort and index the bam files, remove the unsorted bams
for(i in 1:length(fq_fnames)) {
  cmd <- paste0("bwa mem ", bwa_index, " ", fq_fnames[i],
                " | samtools view -Sb - > ", bm_fnames[i])
  message(cmd, "\n"); system(cmd)
  indexBam(sortBam(bm_fnames[i],srt_bm_fnames[i]))
  unlink(bm_fnames[i])
}
```

See the help for **bwa index** at the bwa man page and for general details on mapping sequences to a genome reference.

Read in BAM files and initialize CrisprSet object

Given a set of BAM files with the amplicon sequences of interest mapped to the reference genome, we need to collect a few additional pieces of information about the guide sequence and define the area around the guide that we want to summarize the mutation spectrum over.

If you already know the coordinates, these can be typed in or put in a BED file that can be read in using the **rtracklayer** package. The import() below command returns a GRanges object.

```
library(rtracklayer)
# Represent the guide as a GenomicRanges::GRanges object
gd_fname <- system.file(package="CrispRVariants", "extdata/bed/guide.bed")
gd <- rtracklayer::import(gd_fname)
gd
```

```
## GRanges object with 1 range and 2 metadata columns:
##       seqnames              ranges strand |        name     score
##          <Rle>           <IRanges>  <Rle> | <character> <numeric>
##   [1]    chr17 [23648474, 23648496]      - |   ptena_ccA         0
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Below, we'll extend the guide region by 5 bases on each side when counting variants. The guide designed for *ptena* (including PAM) is 23bp and is located on chromosome chr17 from 23648474-23648496. Note that the expected cut site (used later for labeling variants), after extension, is at base 22 with respect to the start of the guide sequence.

```
gdl <- resize(gd, width(gd) + 10, fix = "center")
```

With the Bioconductor **BSgenome** packages, the reference sequence itself can be retrieved directly into a DNAStringSet object. For other genomes, the reference sequence can be retrieved from a genome by first indexing the genome with samtools faidx and then fetching the required region. Here we are using the GRCHz10 zebrafish genome. The reference sequence was fetched and saved as follows:

```
system("samtools faidx GRCHz10.fa.gz")

reference=system(sprintf("samtools faidx GRCHz10.fa.gz %s:%s-%s",
                         seqnames(gdl)[1], start(gdl)[1], end(gdl)[1]),
                 intern = TRUE)[[2]]

# The guide is on the negative strand, so the reference needs to be reverse complemented
reference=Biostrings::reverseComplement(Biostrings::DNAString(reference))
save(reference, file = "ptena_GRCHz10_ref.rda")
```

We'll load the previously saved reference sequence.

```
ref_fname <- system.file(package="CrispRVariants", "extdata/ptena_GRCHz10_ref.rda")
load(ref_fname)
reference
```

```
##   33-letter "DNAString" instance
## seq: GCCATGGGCTTTCCAGCCGAACGATTGGAAGGT
```

Note the NGG sequence (here, TGG) is present with the 5 extra bases on the end.

To allow easy matching to experimental condition (e.g., useful for colour labeling) and for subsetting to experiments of interest, we often organize the list of BAM files together with accompanying metadata in a machine-readable table beforehand:

```
# The metadata and bam files for this experiment are included with CrispRVariants
library("gdata")
md_fname <- system.file(package="CrispRVariants", "extdata/metadata/metadata.xls")
md <- gdata::read.xls(md_fname, 1)
md
```

```
##                                     bamfile                        directory
## 1        ab1_ptena_phenotype_embryo_1_s.bam        ptena/phenotype/embryo 1
## 2        ab1_ptena_phenotype_embryo_2_s.bam        ptena/phenotype/embryo 2
## 3 ab1_ptena_wildtype_looking_embryo_1_s.bam ptena/wildtype looking/embryo 1
## 4 ab1_ptena_wildtype_looking_embryo_2_s.bam ptena/wildtype looking/embryo 2
## 5       ab1_ptena_uninjected_embryo_1_s.bam       ptena/uninjected/embryo 1
##   Short.name Targeting.type    sgRNA1 sgRNA2   Group
## 1    ptena 1         single ptena_ccA     NA  strong
## 2    ptena 2         single ptena_ccA     NA  strong
## 3    ptena 3         single ptena_ccA     NA    mild
## 4    ptena 4         single ptena_ccA     NA    mild
## 5    control         single ptena_ccA     NA control
```

```
# Get the bam filenames from the metadata table
bam_dir <- system.file(package="CrispRVariants", "extdata/bam")
bam_fnames <- file.path(bam_dir, md$bamfile)

# check that all files exist
all( file.exists(bam_fnames) )
```

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

Creating a CrisprSet

The next step is to create a CrisprSet object, which is the container that stores the relevant sequence information, alignments, observed variants and their frequencies.

```
# Note that the zero point (target.loc parameter) is 22
crispr_set <- readsToTarget(bam_fnames, target = gdl, reference = reference,
                            names = md$Short.name, target.loc = 22)
crispr_set
```

```
## CrisprSet object containing 5 CrisprRun samples
## Target location:
## GRanges object with 1 range and 2 metadata columns:
##       seqnames               ranges strand |        name     score
##          <Rle>            <IRanges>  <Rle> | <character> <numeric>
##   [1]    chr17 [23648469, 23648501]      - |   ptena_ccA         0
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
## [1] "Most frequent variants:"
##            ptena 1 ptena 2 ptena 3 ptena 4 control
## no variant       3       4       4       0       7
## -1:4D            0       0       0       2       0
## 6:1D             0       0       0       1       1
```

```
## 1:7I             1       0       0       0       0
## 2:1D,4:5I        0       0       0       1       0
## Other            0       0       1       1       0
```

```
# The counts table can be accessed with the "variantCounts" function
vc <- variantCounts(crispr_set)
print(class(vc))
```

```
## [1] "matrix"
```

You can see that in the table of variant counts, variants are summarised by the location of their insertions and deletions with respect to the target site. Non-variant sequences and sequences with a single nucleotide variant (SNV) but no insertion or deletion (indel) are displayed first, followed by the indel variants from most to least frequent For example, the most frequent non-wild-type variant, "-1:4D" is a 4 base pair deletion starting 1 base upstream of the zero point.

Creating summary plots of variants

We want to plot the variant frequencies along with the location of the guide sequence relative to the known transcripts. If you do this repeatedly for the same organism, it is worthwhile to save the database in a local file and read in with loadDb(), since this is quicker than retrieving it from UCSC (or Ensembl) each time.

We start by creating a transcript database of Ensembl genes. The gtf was downloaded from Ensembl version 81. We first took a subset of just the genes on chromosome 17 and then generated a transcript database.

```
# Extract genes on chromosome 17 (command line)
# Note that the Ensembl gtf does not include the "chr" prefix, so we add it here
gtf=Danio_rerio.GRCz10.81.gtf.gz
zcat ${gtf} | awk '($1 == 17){print "chr"$0}' > Danio_rerio.GRCz10.81_chr17.gtf
```

```
# In R
library(GenomicFeatures)
gtf_fname <- "Danio_rerio.GRCz10.81_chr17.gtf"
txdb <- GenomicFeatures::makeTxDbFromGFF(gtf_fname, format = "gtf")
saveDb(txdb, file= "GRCz10_81_chr17_txdb.sqlite")
```

We now load the the previously saved database

plotVariants() is a wrapper function that groups together a plot of the transcripts of the gene/s overlapping the guide (optional), CrispRVariants::plotAlignments(), which displays the alignments of the consensus variant sequences to the reference, and CrispRVariants::plotFreqHeatmap(), which produces a table of the variant counts per sample, coloured by either their counts or percentage contribution to the variants observed for a given sample. If a transcript database is supplied, the transcript plot is annotated with the guide location. Arguments for plotAlignments() and plotFreqHeatmap() can be passed to plotVariants() as lists named plotAlignments.args and plotFreqHeatmap.args, respectively.

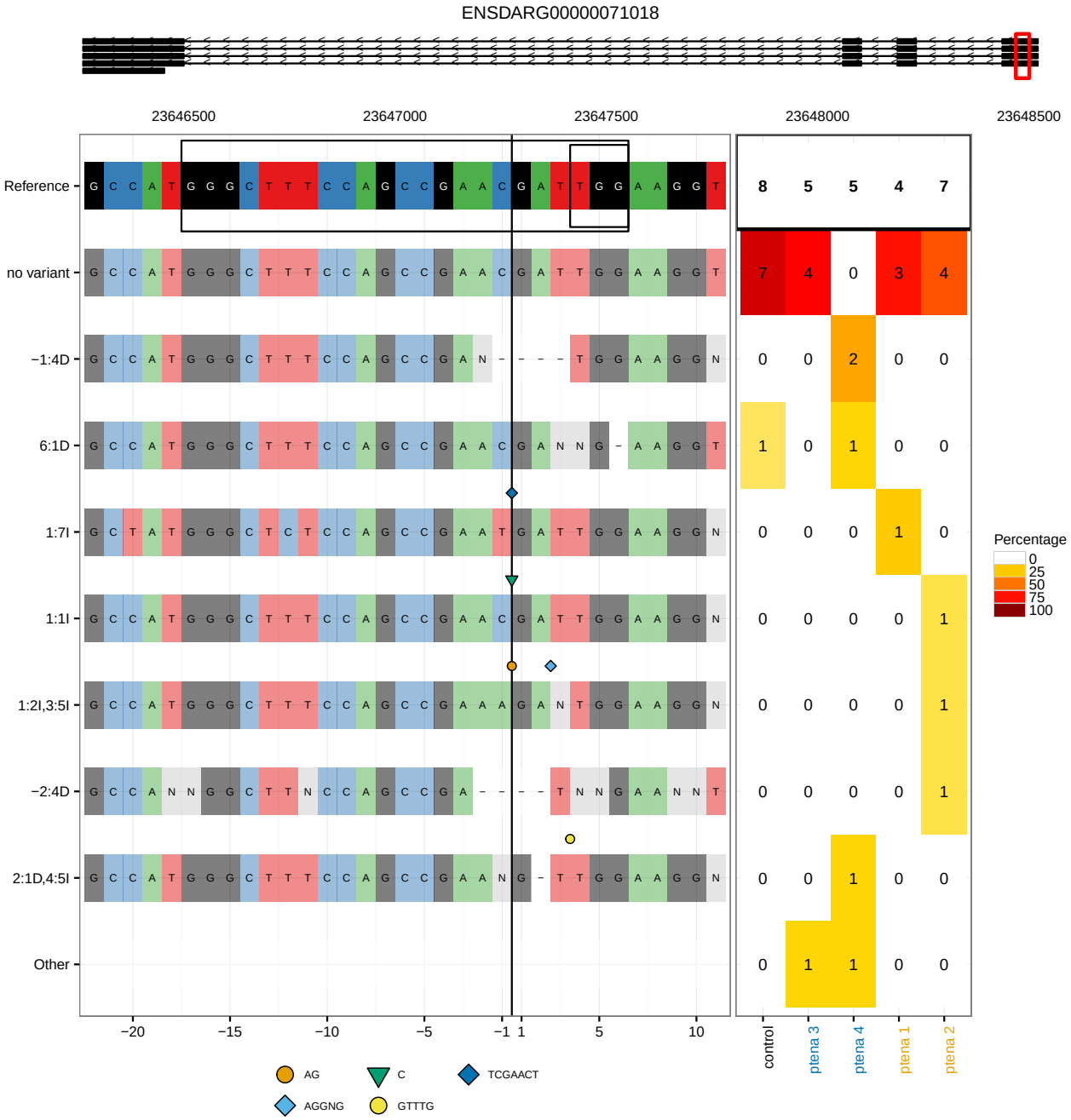
```
# The gridExtra package is required to specify the legend.key.height
# as a "unit" object.  It is not needed to call plotVariants() with defaults
library(gridExtra)
```

```
##
## Attaching package: 'gridExtra'
```

```
## The following object is masked from 'package:Biobase':
##
##     combine

## The following object is masked from 'package:gdata':
##
##     combine

## The following object is masked from 'package:BiocGenerics':
##
##     combine
```

```
# Match the clutch id to the column names of the variants
group <- md$Group
```

```
p <- plotVariants(crispr_set, txdb = txdb, gene.text.size = 8,
    row.ht.ratio = c(1,8), col.wdth.ratio = c(4,2),
    plotAlignments.args = list(line.weight = 0.5, ins.size = 2,
                               legend.symbol.size = 4),
    plotFreqHeatmap.args = list(plot.text.size = 3, x.size = 8, group = group,
                                legend.text.size = 8,
                                legend.key.height = grid::unit(0.5, "lines")))
```



The plotVariants() options set the text size of the transcript plot annotation (gene.text.size) and the relative heights (row.ht.ratio) and widths (col.wdth.ratio) of the plots.

The plotAlignments arguments set the symbol size in the figure (ins.size) and in the legend (legend.symbol), the line thickness for the (optional) annotation of the guide region and cleavage site (line.weight).

For plotFreqHeatmap we define an grouping variable for colouring the x-axis labels (group), the size of the text within the plot (plot.text.size) and on the x-axis (x.size) and set the size of the legend text (legend.text.size).

Calculating the mutation efficiency

The mutation efficiency is the number of reads that include an insertion or deletion. Chimeric reads and reads containing single nucleotide variants near the cut site may be counted as variant reads, non-variant

reads, or excluded entirely. See the help page for the function **mutationEfficiency** for more details.

We can see in the plot above that the control sample includes a variant sequence 6:1D, also present in sample ptena 4. We will exclude all sequences with this variant from the efficiency calculation. We also demonstrate below how to exclude particular variants.

```
# Calculate the mutation efficiency, excluding indels that occur in the "control" sample
# and further excluding the "control" sample from the efficiency calculation
eff <- mutationEfficiency(crispr_set, filter.cols = "control", exclude.cols = "control")
eff
```

```
##   ptena 1   ptena 2   ptena 3   ptena 4   Average    Median   Overall     StDev
##     25.00     42.86     20.00     80.00     41.96     33.93     42.86      1.50
## ReadCount
##     21.00
```

```
# Suppose we just wanted to filter particular variants, not an entire sample.
# This can be done using the "filter.vars" argument
eff2 <- mutationEfficiency(crispr_set, filter.vars = "6:1D", exclude.cols = "control")

# The results are the same in this case as only one variant was filtered from the control
identical(eff,eff2)
```

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

We see above that sample ptena 4 has an efficiency of 80%, i.e. 4 variant sequences, plus one sequence "6:1D" which is counted as a non-variant sequence as it also occurs in the control sample.

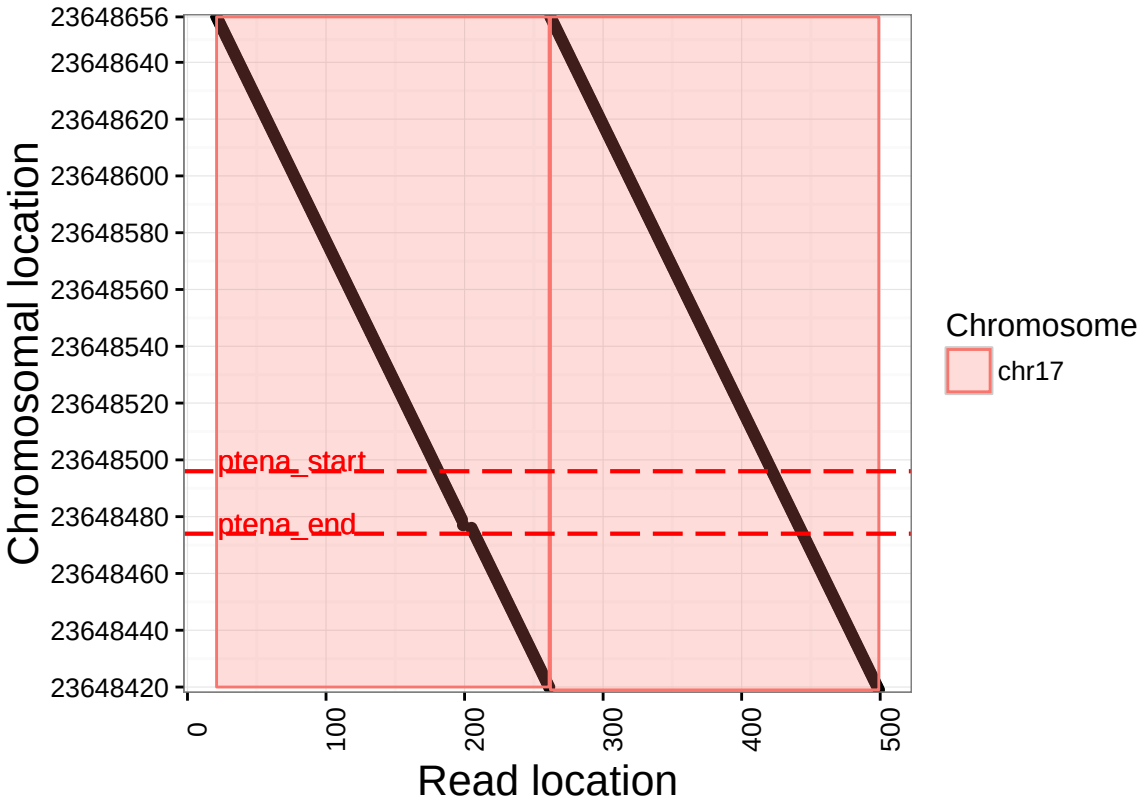
Plot chimeric alignments

When deciding whether chimeric alignments should be considered as variant sequences, it can be useful to plot the frequent chimeras.

```
ch <- getChimeras(crispr_set, sample = "ptena 4")

# Confirm that all chimeric alignments are part of the same read
length(unique(names(ch))) == 1
```

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

```
# Set up points to annotate on the plot
annotations <- c(resize(gd, 1, fix = "start"), resize(gd, 1, fix = "end"))
annotations$name <- c("ptena_start", "ptena_end")

plotChimeras(ch, annotations = annotations)
```



Here we see the read aligns as two tandem copies of the region chr17:23648420-23648656. The endpoint of each copy is not near the guide sequence. We do not consider this a genuine mutation, so we'll recalculate the mutation efficiency excluding the chimeric reads and the control variant as before.

```
mutationEfficiency(crispr_set, filter.cols = "control", exclude.cols = "control",
                   include.chimeras = FALSE)
```

```
##   ptena 1   ptena 2   ptena 3   ptena 4   Average    Median   Overall     StDev
##     25.00     42.86      0.00     75.00     35.71     33.93     36.84      1.50
## ReadCount
##     19.00
```

We see that the mutation effiency for "ptena 4" is now 75%, i.e. 3 genuine variant sequences, 1 sequence counted as "non-variant" because it occurs in the control, and the chimeric read excluded completely

Changing the appearance of plots

Note that arguments for CrispRVariants::plotAlignments described below can be passed to CrispRVariants::plotVariants as a list, e.g. `plotAlignments.args = list(axis.text.size = 14)`. Similarly, arguments for CrispRVariants::plotFreqHeatmap are passed through plotVariants via `plotFreqHeatmap.args`.

Filtering data in plotVariants

For the following examples, we will use the *ptena* data set. We must first load the data and create a `CrispRVariants::CrisprSet` object.

```
# Setup for ptena data set
library("CrispRVariants")
library("rtracklayer")
library("GenomicFeatures")
library("gdata")

# Load the guide location
gd_fname <- system.file(package="CrispRVariants", "extdata/bed/guide.bed")
gd <- rtracklayer::import(gd_fname)
gdl <- resize(gd, width(gd) + 10, fix = "center")

# The saved reference sequence corresponds to the guide
# plus 5 bases on either side, i.e. gdl
ref_fname <- system.file(package="CrispRVariants",
                         "extdata/ptena_GRCHz10_ref.rda")
load(ref_fname)

# Load the metadata table, which gives the sample names
md_fname <- system.file(package="CrispRVariants",
                        "extdata/metadata/metadata.xls")
md <- gdata::read.xls(md_fname, 1)

# Get the list of bam files
bam_dir <- system.file(package="CrispRVariants", "extdata/bam")
bam_fnames <- file.path(bam_dir, md$bamfile)

# Check that all files were found
all(file.exists(bam_fnames))
```

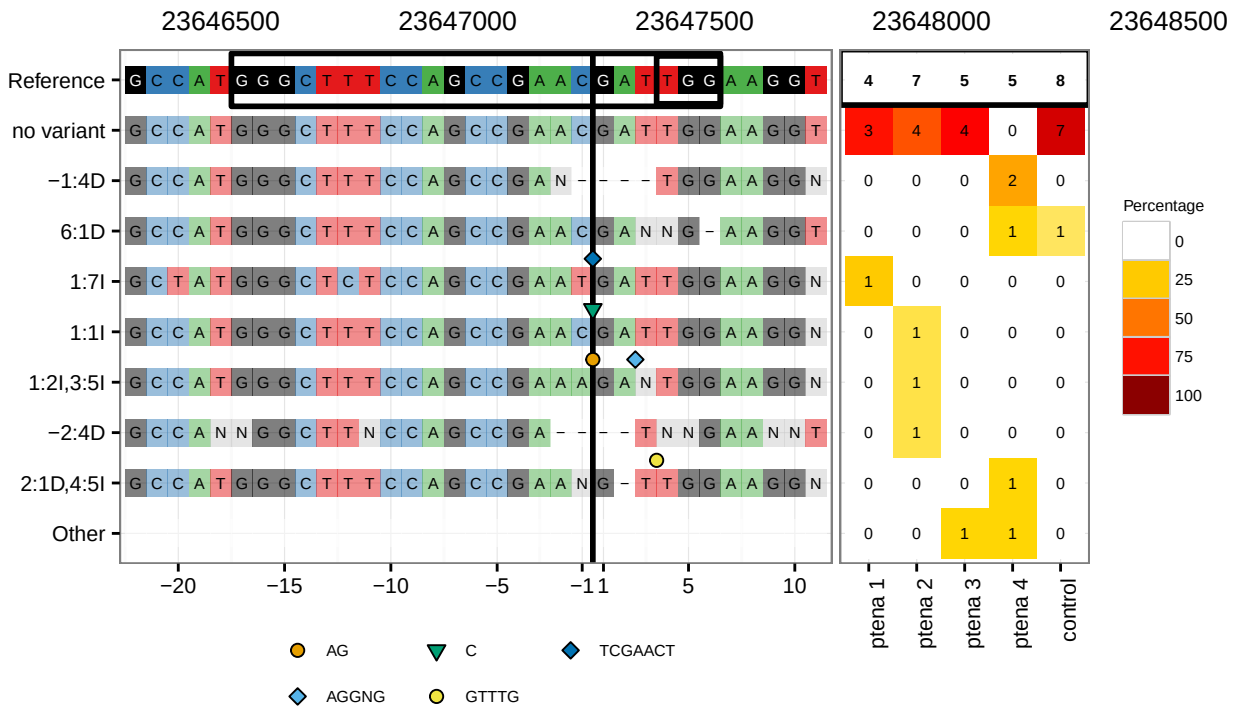
```
## [1] TRUE
```

```
crispr_set <- readsToTarget(bam_fnames, target = gdl, reference = reference,
                            names = md$Short.name, target.loc = 22,
                            verbose = FALSE)

# Load the transcript database
txdb_fname <- system.file("extdata/GRCz10_81_ptena_txdb.sqlite",
                          package="CrispRVariants")
txdb <- AnnotationDbi::loadDb(txdb_fname)
```

Here is the ptena data set plotted with default options:

```
p <- plotVariants(crispr_set, txdb = txdb)
```

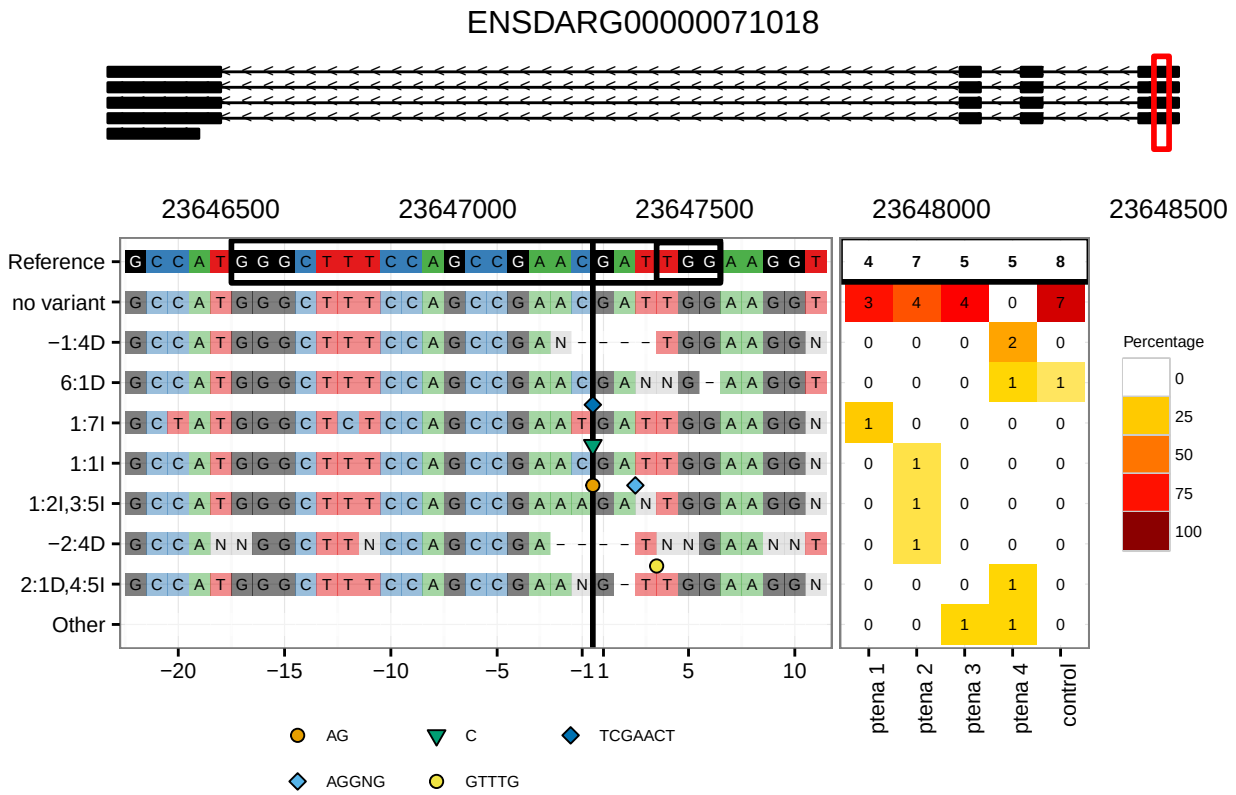
```
## 'select()' returned 1:many mapping between keys and columns
## 'select()' returned 1:many mapping between keys and columns
```



The layout of this plot is controlled mainly by two parameters: `row.ht.ratio` and `col.wdth.ratio`. `row.ht.ratio` (default `c(1,6)`) controls the relative sizes of the transcript plot and the other plots. Below we show how to change the ratio so that the transcript plot is relatively larger:

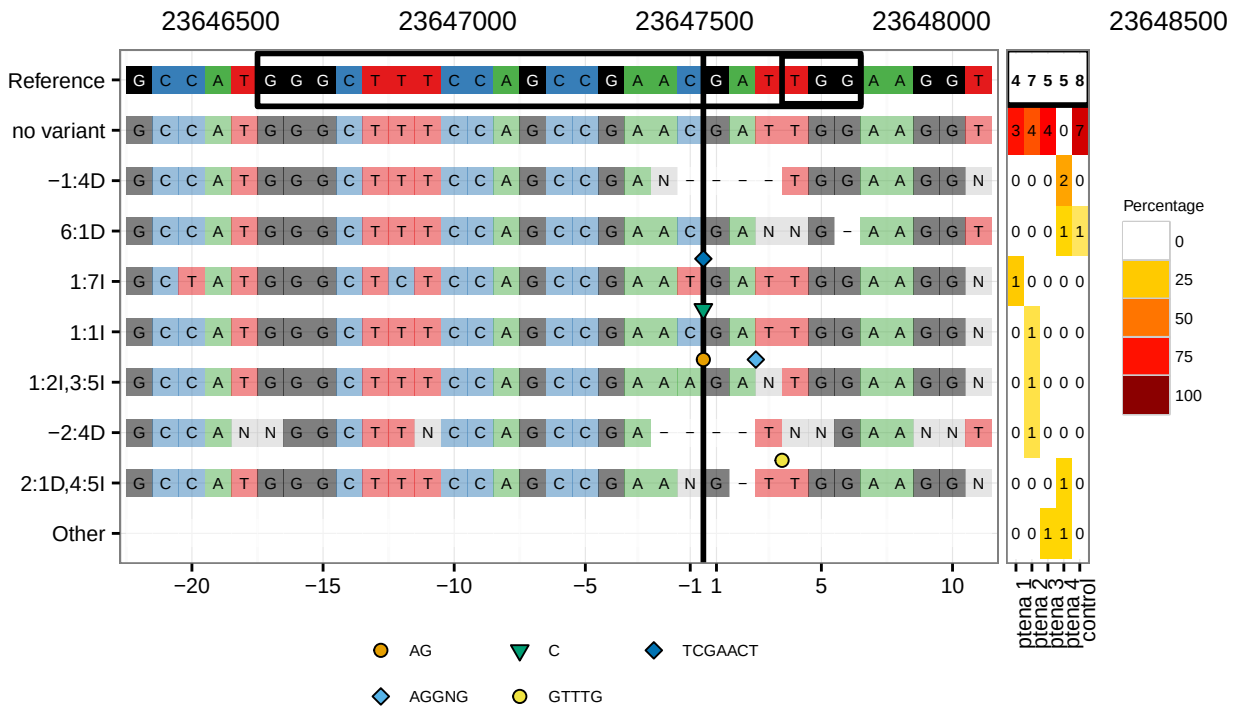
```
p <- plotVariants(crispr_set, txdb = txdb, row.ht.ratio = c(1,3))
```

```
## 'select()' returned 1:many mapping between keys and columns
## 'select()' returned 1:many mapping between keys and columns
```



Similarly, `col.wdth.ratio` controls the width ratio of the alignment plot and the heatmap (default `c(2,1)`, i.e. the alignment plot is twice as wide as the heatmap). Below we alter this to make the alignment plot comparatively wider:

```
p <- plotVariants(crispr_set, txdb = txdb, col.wdth.ratio = c(4,1))
```

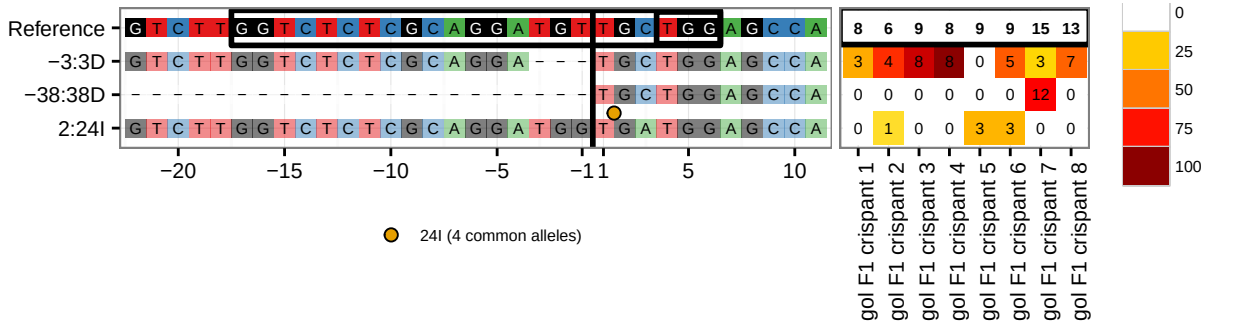


The remaining examples in this section use the *gol* data set.

```
# Load gol data set
library("CrispRVariants")
data(gol_clutch1)
```

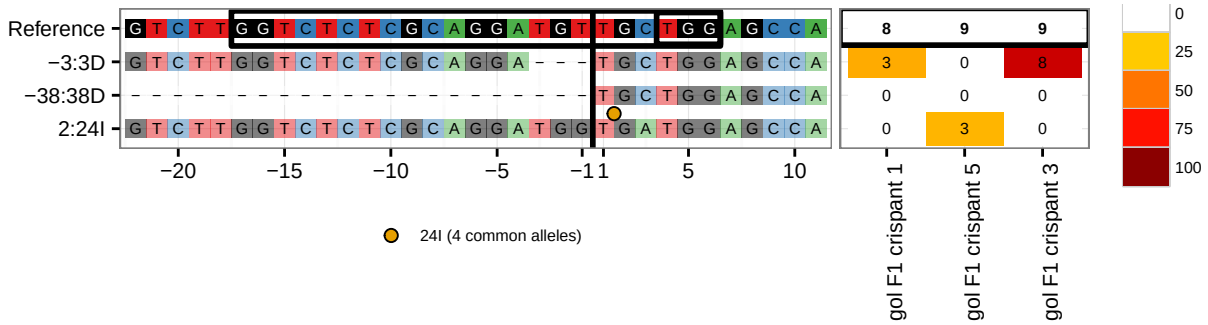
The data used in `plotAlignments` and `plotFreqHeatmap` can be filtered by either frequency via `min.freq`, count via `min.count`, or to show a set number of alleles sorted by frequency, via `top.n`. Within `plotVariants`, these filtering options need to be set for both `plotAlignments" and`plotFreqHeatmap"'. We also add space to the bottom of the plot to prevent clipping of the labels.

```
library(GenomicFeatures)
p <- plotVariants(gol, plotAlignments.args = list(top.n = 3),
             plotFreqHeatmap.args = list(top.n = 3),
             left.plot.margin = ggplot2::unit(c(0.1,0,5,0.2), "lines"))
```



At present, filtering by sample (column) is possible for `plotFreqHeatmap` via the `order` parameter (which can also be used to reorder columns), but not '`plotAlignments`.

```
plotVariants(gol, plotAlignments.args = list(top.n = 3),
             plotFreqHeatmap.args = list(top.n = 3, order = c(1,5,3)),
             left.plot.margin = ggplot2::unit(c(0.1,0,5,0.2), "lines"))
```



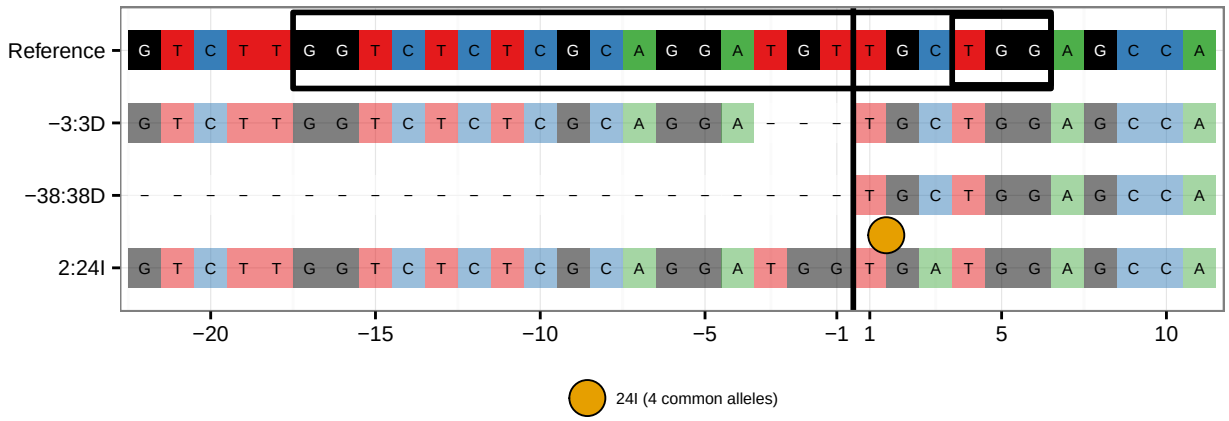

```
## TableGrob (2 x 1) "arrange": 2 grobs
##   z     cells    name                grob
## 1 1 (1-1,1-1) arrange rect[GRID.rect.802]
## 2 2 (2-2,1-1) arrange     gtable[arrange]
```

plotAlignments

Insertion symbols

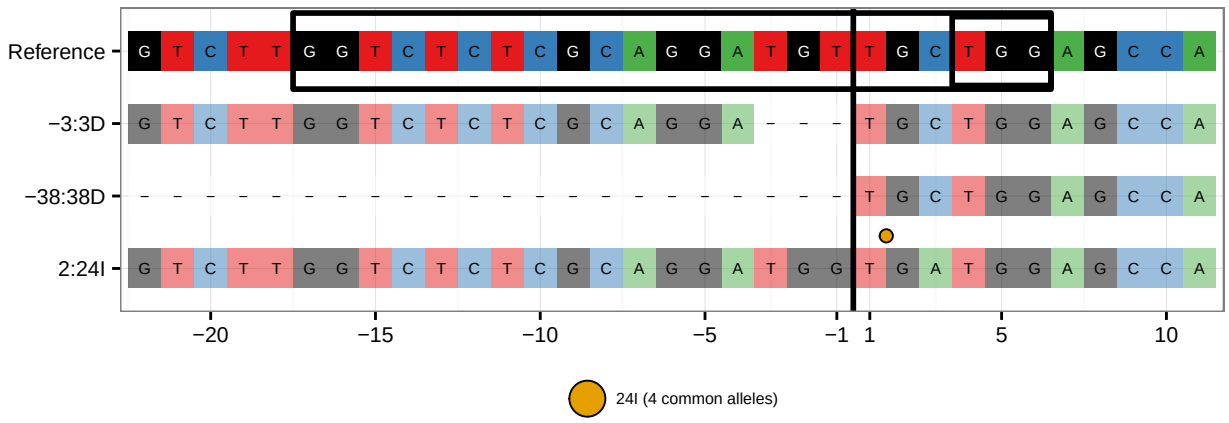
The symbols indicating insertions are controlled by four parameters. `ins.size` (default 3) controls the size of the symbols within the plot area.

```
plotAlignments(gol, top.n = 3, ins.size = 6)
```



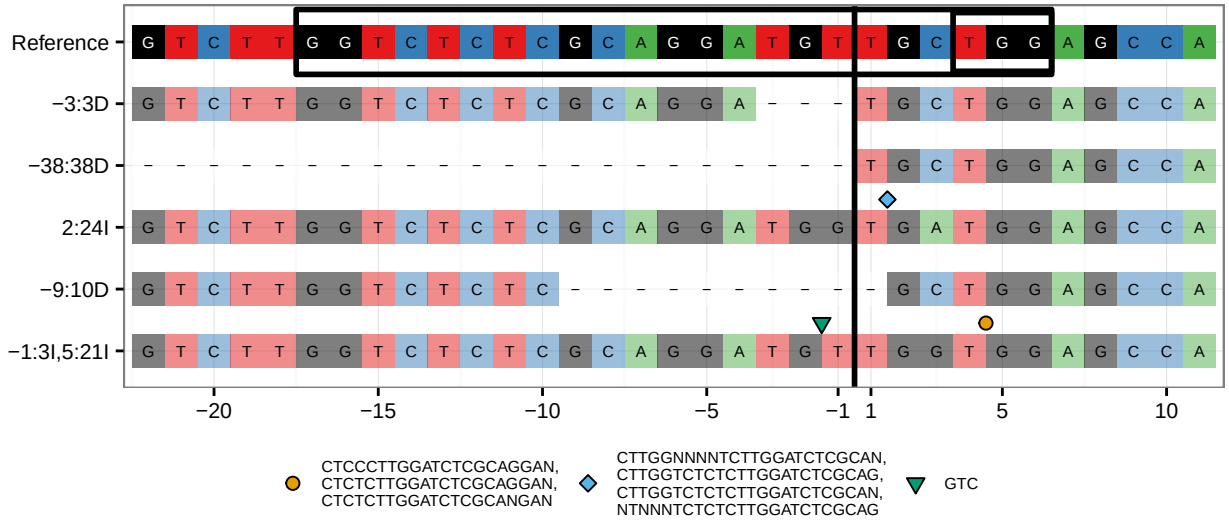
By default the symbols in the legend are the same size as those in the plot, but this can be controlled separately with `legend.symbol.size`.

```
plotAlignments(gol, top.n = 3, legend.symbol.size = 6)
```



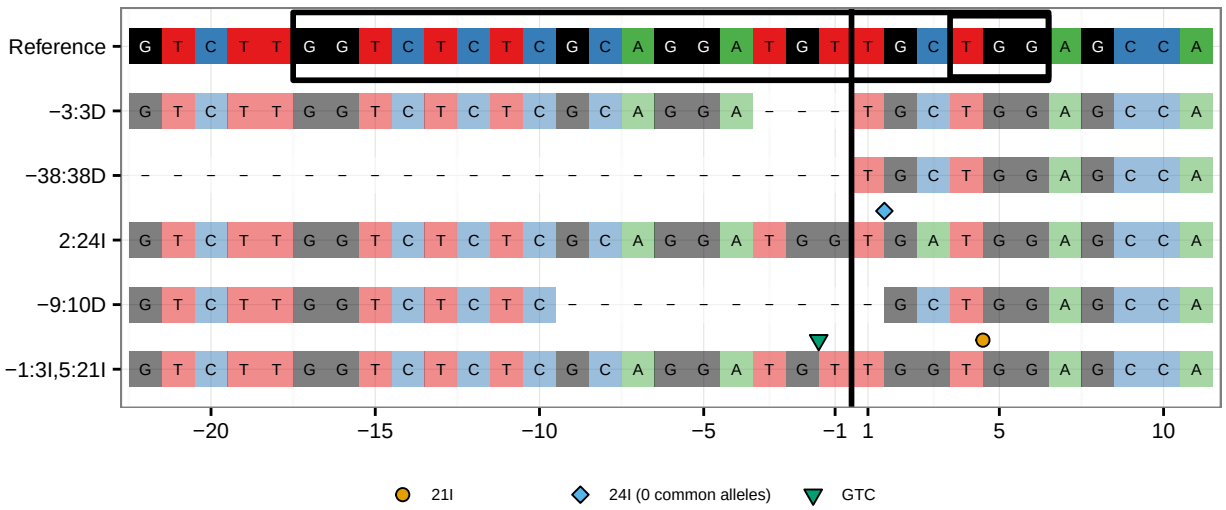
As long sequences can make the plot difficult to read, by default only the length of insertions greater then 20bp is shown. This can be changed with the `max.insertion.size` parameter. If there is more than one allele, the number of (frequent) alleles is indicated.

```
plotAlignments(gol, top.n = 5, max.insertion.size = 25)
```



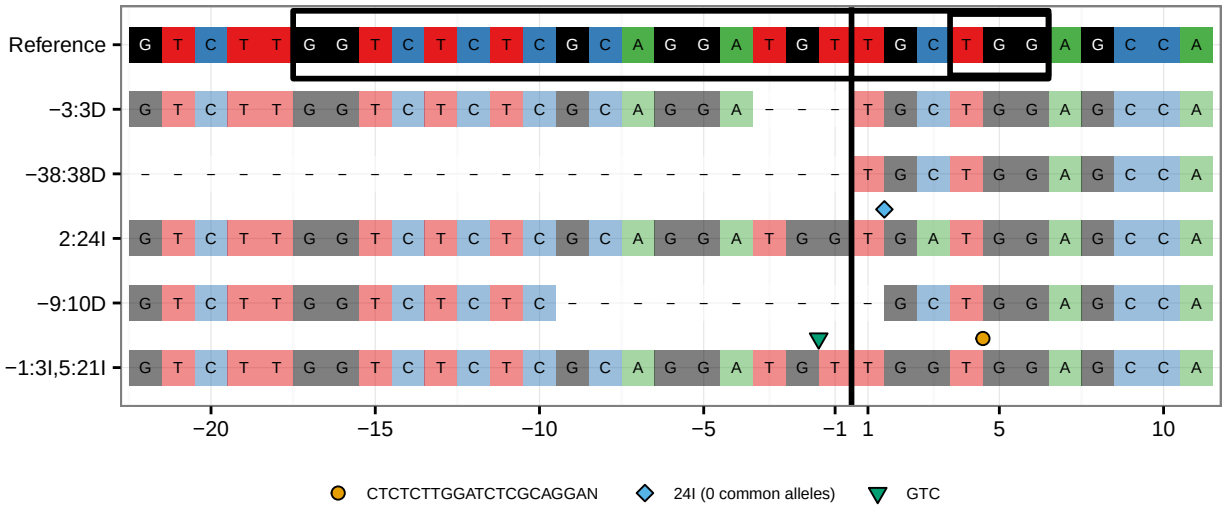
Finally, the parameter `min.insertion.freq` (default 5%) controls how many alleles are displayed at each insertion locus. In large data sets, there will be a substantial proportion of reads with sequencing errors, and we may only wish to display the most common sequences.

```
# Here we set a fairly high value of 50% for min.insertion.freq
# As ambiguous nucleotides occur frequently in this data set,
# there are no alleles passing this cutoff.
plotAlignments(gol, top.n = 5, min.insertion.freq = 50)
```



`max.insertion.size` and `min.insertion.freq` can be combined. In this case, alleles longer than `max.insertion.size` but less frequent than `min.insertion.freq` will be collapsed.

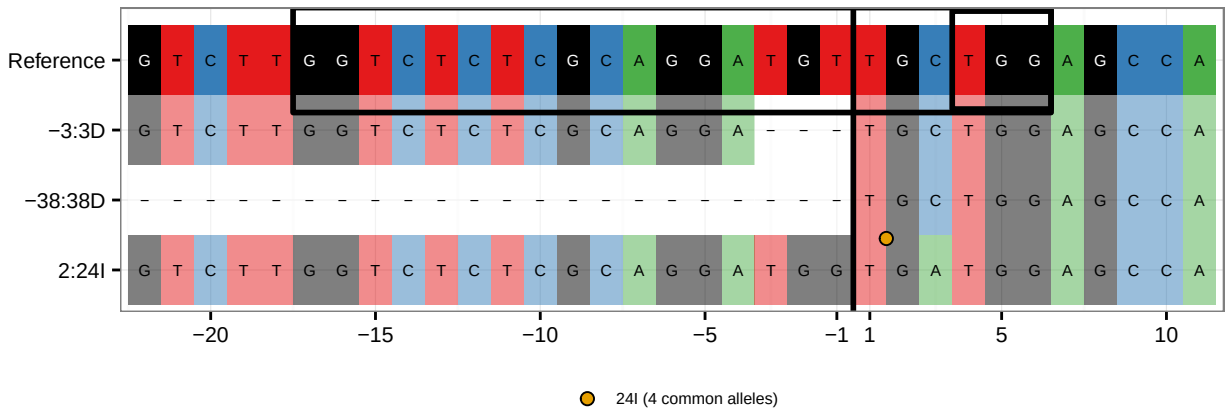
```
plotAlignments(gol, top.n = 5, max.insertion.size = 25, min.insertion.freq = 50)
```



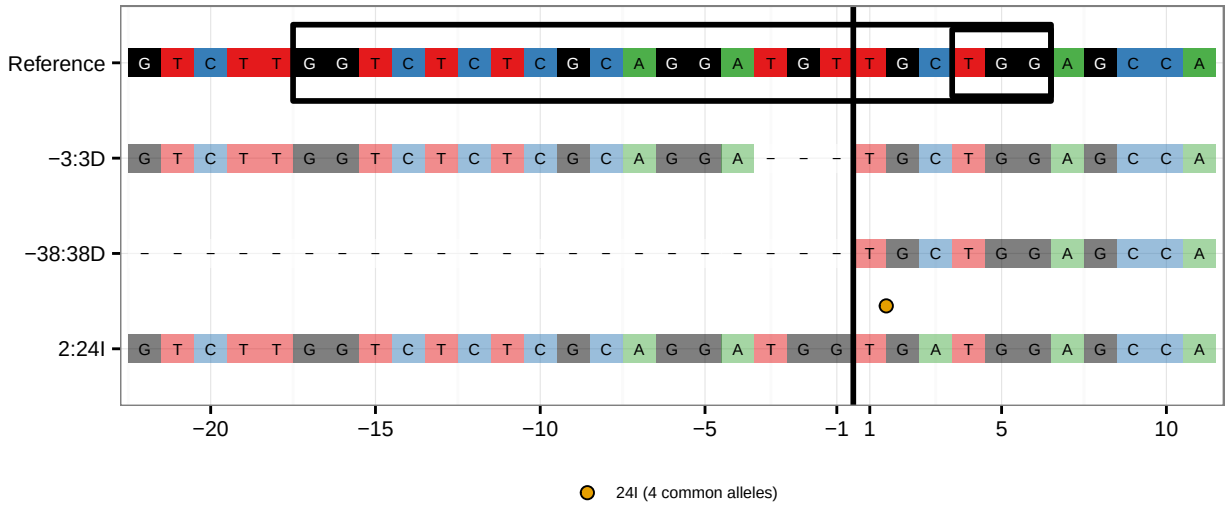
Whitespace between rows

The space between rows is controlled with the `tile.height` parameter (default 0.55). Values closer to 0 increase the space between rows, whilst values closer to 1 decrease the space between rows.

```
# No white space between rows
plotAlignments(gol, top.n = 3, tile.height = 1)
```



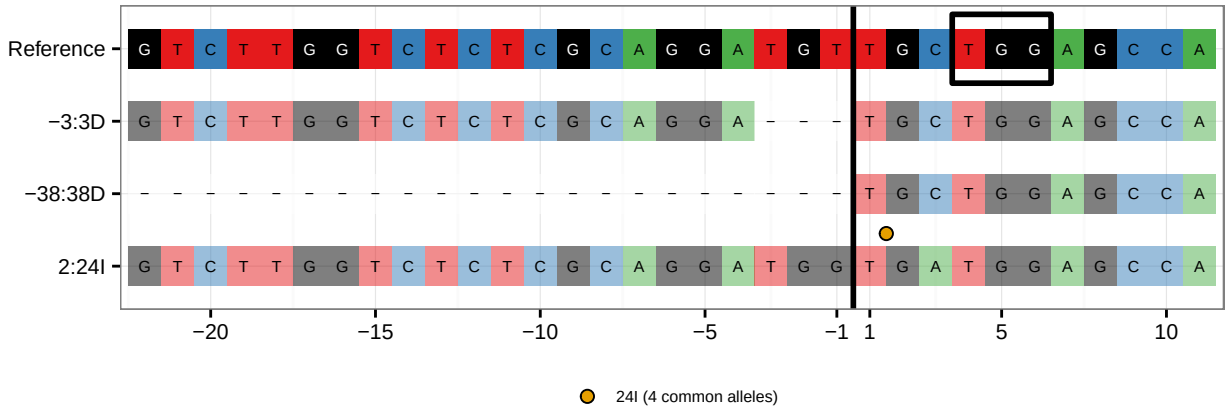

```
# More white space between rows
plotAlignments(gol, top.n = 3, tile.height = 0.3)
```



Box around guide

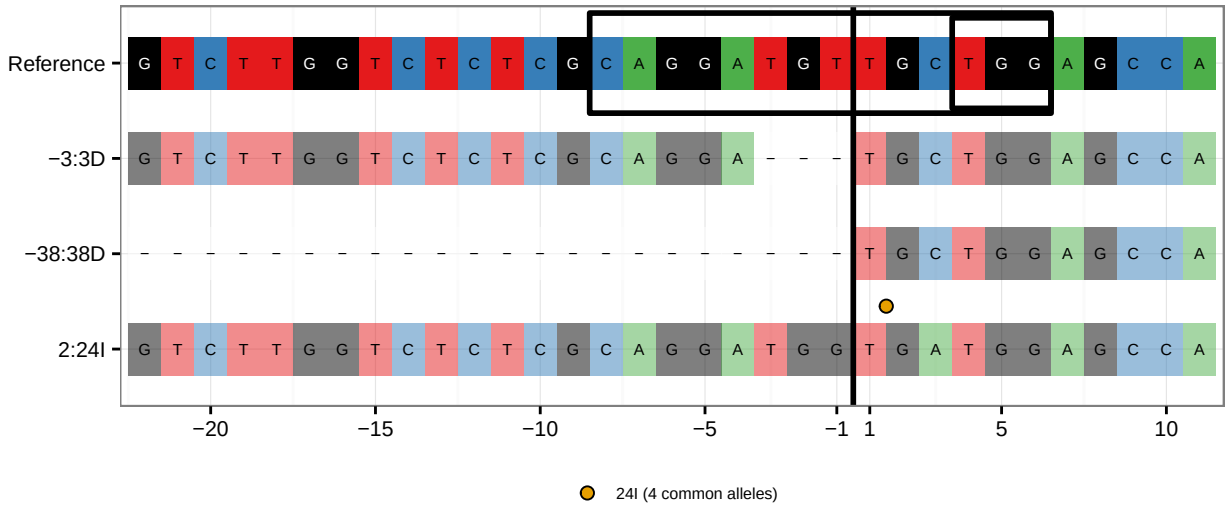
The black box around the guide sequence can be removed by setting `highlight.guide = FALSE`.

```
plotAlignments(gol, top.n = 3, highlight.guide = FALSE)
```



By default, the box around the guide is drawn from 17 bases upstream of the `target.loc` to 6 bases downstream. For experiments with a truncated guide, or other non-standard guide location, the box must be manually specified. The guide location can be altered by setting the `guide.loc` parameter. This can be either an IRanges::IRanges or GenomicRanges::GRanges object.

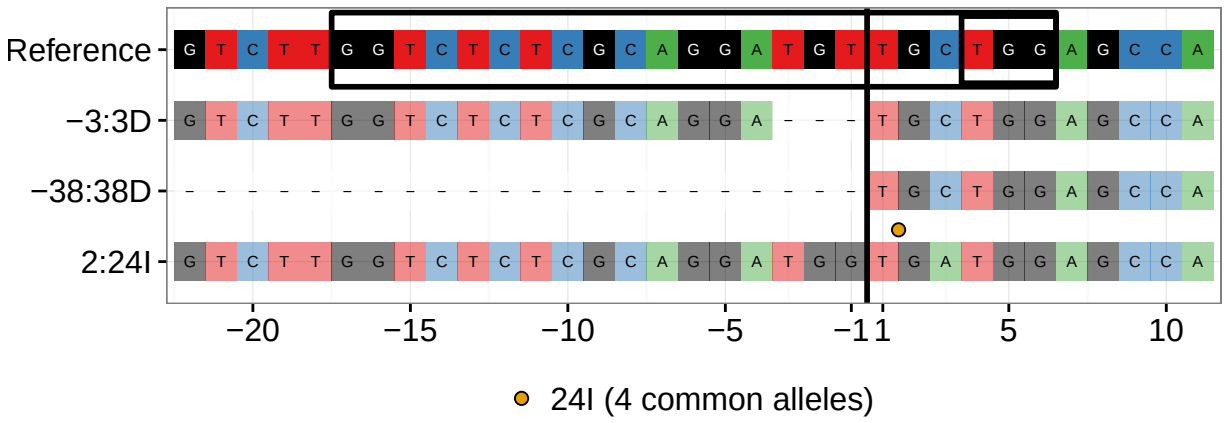
```
library(IRanges)
guide <- IRanges::IRanges(15,28)
plotAlignments(gol, top.n = 3, guide.loc = guide)
```



Text sizes

The text within the alignments is controlled by `plot.text.size` (default 0), and can be removed completely by setting `plot.text.size = 0`. The axis labels and legend labels are controlled with `axis.text.size` (default 8) and `legend.text.size` (default 6) respectively. The number of columns in the legend is controlled by `legend.cols` (default 3).

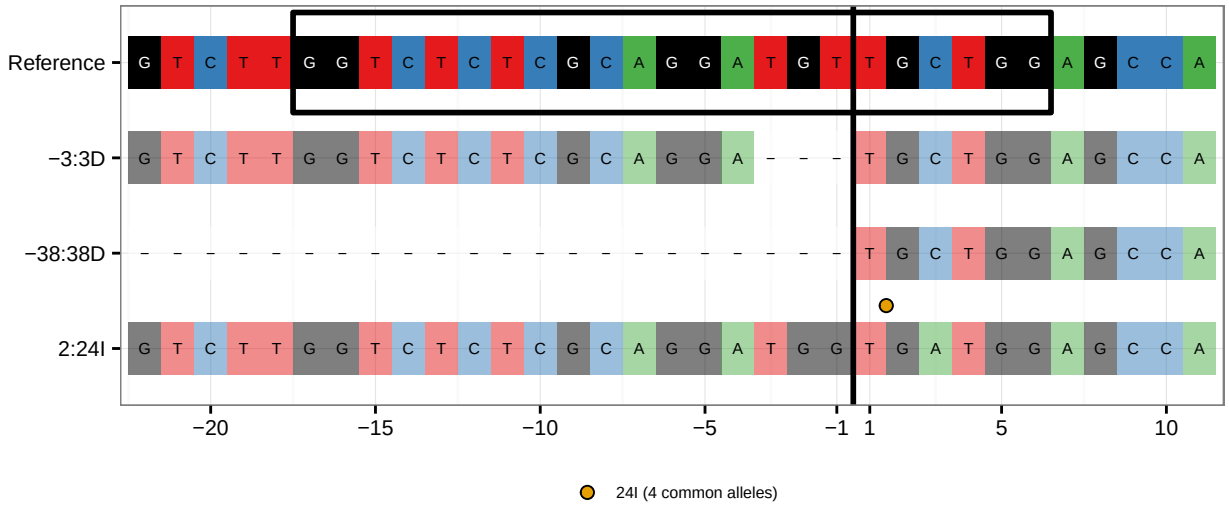
```
# Here we increase the size of the axis labels and make
# two columns for the legend
plotAlignments(gol, top.n = 3, axis.text.size = 12,
               legend.text.size = 12, legend.cols = 2)
```



Box around PAM

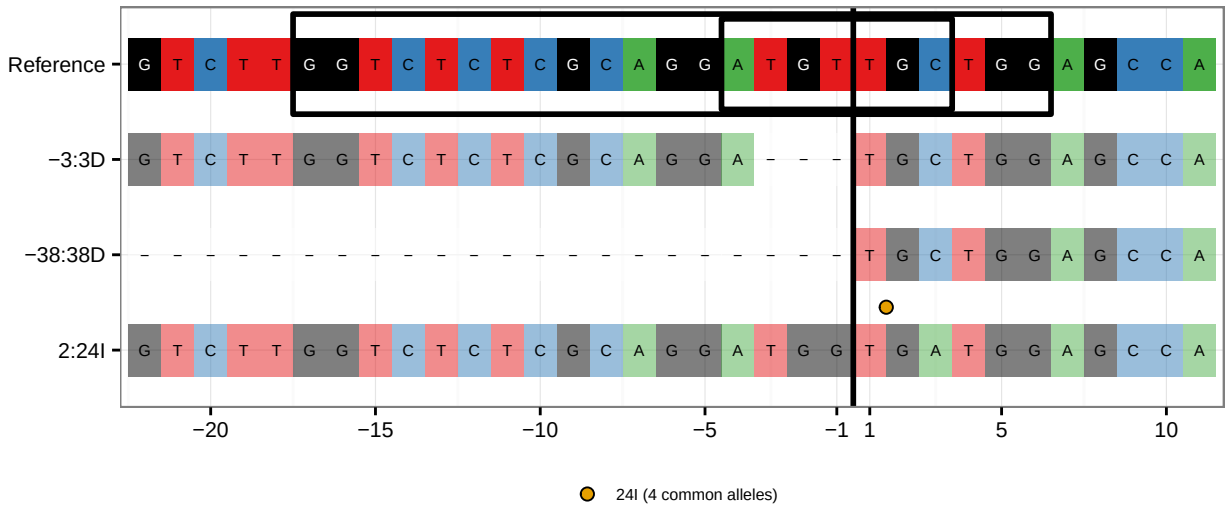
The argument `highlight.pam` determines whether a box around the PAM should be drawn.

```
# Don't highlight the PAM sequence
plotAlignments(gol, top.n = 3, highlight.pam = FALSE)
```



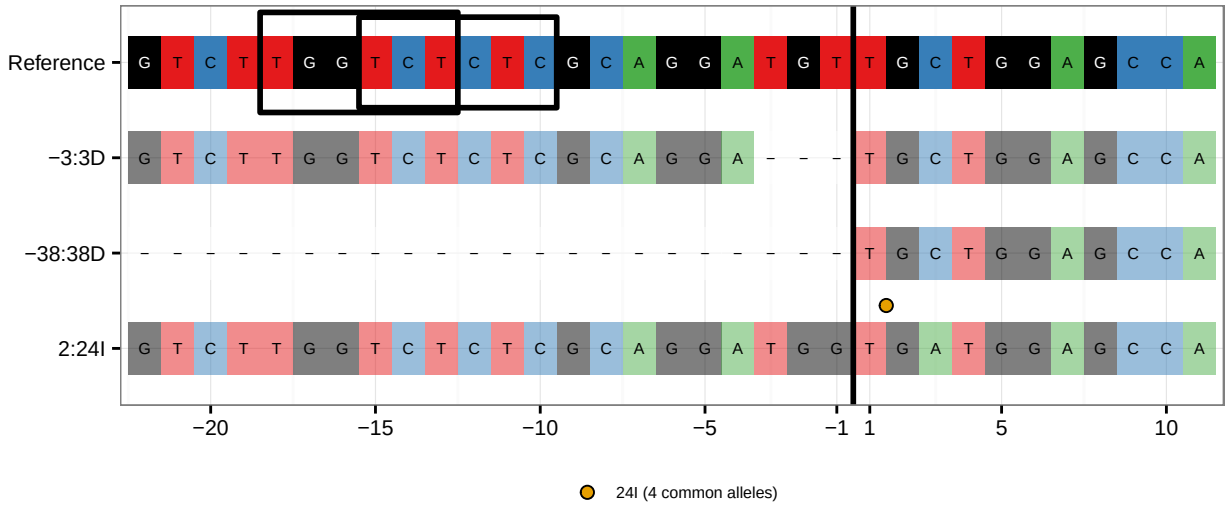
By default this box is drawn 3 nucleotides downstream of the `target.loc`. Other applications might require a different region highlighted. This can be achieved by explicitly setting the start and end positions of the box, with respect to the reference sequence.

```
# Highlight 3 bases upstream to 3 bases downstream of the target.loc
plotAlignments(gol, top.n = 3, pam.start = 19, pam.end = 25)
```



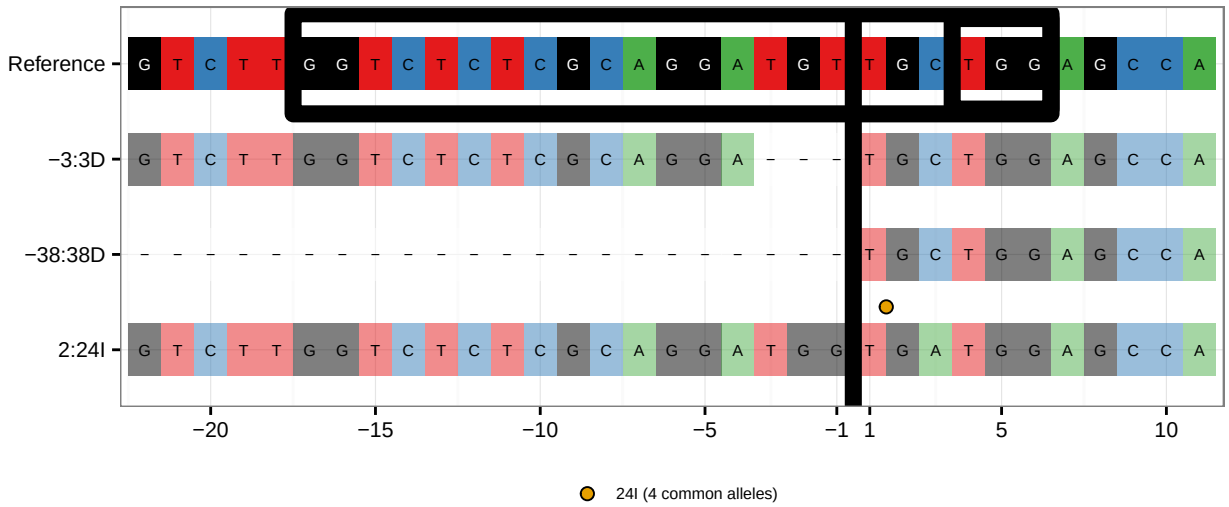
The boxes around the guide and the PAM can both be changed to arbitrary locations, however note that the guide box is specified by a ranges object whilst the PAM box is specified by start and end coordinates. Both coordinates are with respect to the start of the reference sequence. The box around the guide is slightly wider than the box around the PAM.

```
plotAlignments(gol, top.n = 3, guide.loc = IRanges(5,10),
               pam.start = 8, pam.end = 13)
```



The thickness of the lines showing the cut site, the guide and the PAM are controlled with `line.weight` (default 1).

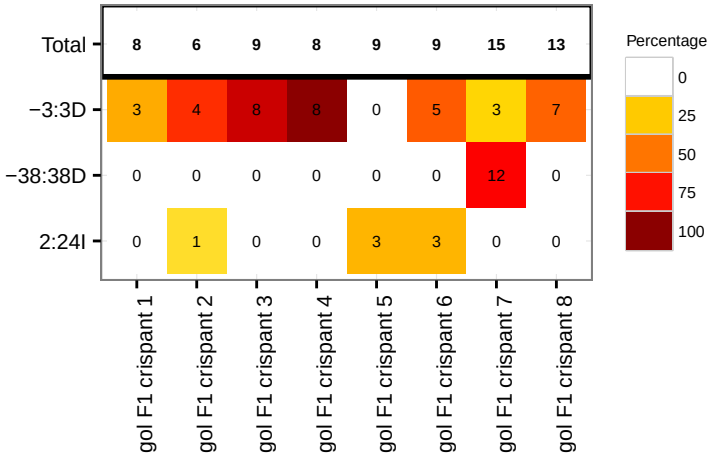
```
plotAlignments(gol, top.n = 3, line.weight = 3)
```



plotFreqHeatmap

Here is the result of calling `plotFreqHeatmap` with default values, showing the three most common variant alleles.

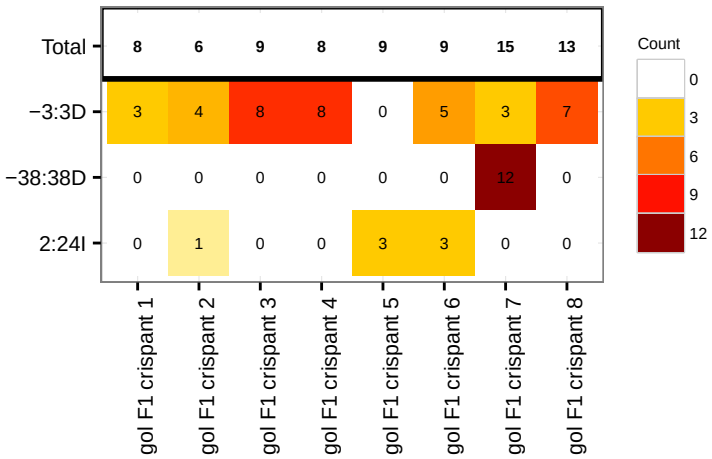
```
plotFreqHeatmap(gol, top.n = 3)
```



Controlling the data plotted

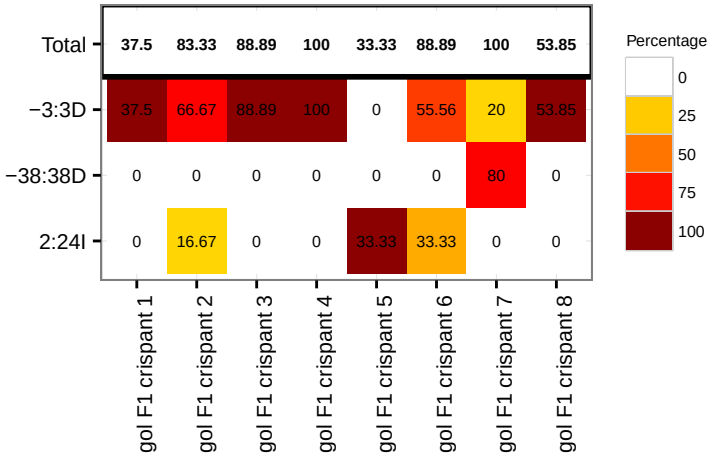
The tiles may be coloured by either the percentage of the column totals (default), or by the counts, by setting `as.percent = FALSE`. The column headers show the total number of sequences in the data. Typically, rare variants are excluded, so the displayed variants do not add up to the column totals.

```
plotFreqHeatmap(gol, top.n = 3, as.percent = FALSE)
```



When calling `plotFreqHeatmap.CrisprSet`, the data can be displayed as percentages instead of raw counts by setting `type = "proportions"` instead of the default `type = "counts"`.

```
plotFreqHeatmap(gol, top.n = 3, type = "proportions")
```



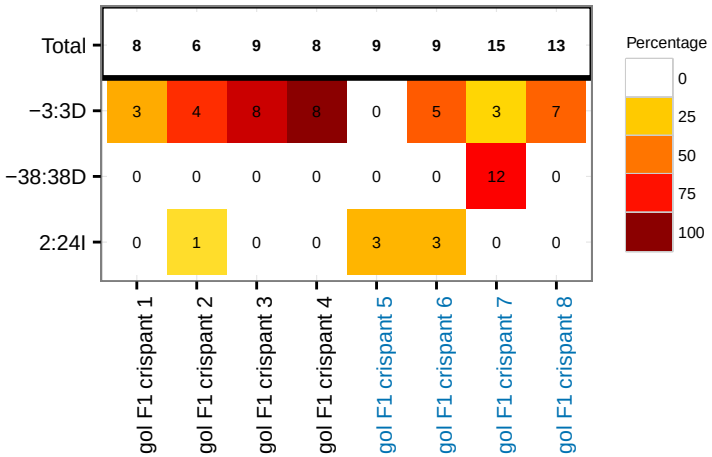
Changing colours of x-labels

The x-labels can be coloured by experimental group. To do this, a grouping vector must be supplied by setting parameter `group`. Columns are ordered according to the levels of the group. There should be one group value per column in the data.

```
ncolumns <- ncol(variantCounts(gol))
ncolumns
```

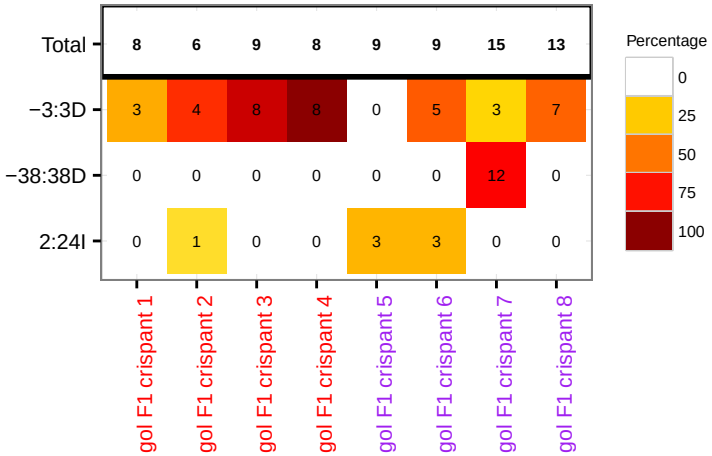
```
## [1] 8
```

```
grp <- rep(c(1,2), each = ncolumns/2)
plotFreqHeatmap(gol, top.n = 3, group = grp)
```



The default colours are designed to be readable on a white background and colour-blind safe. These can be changed by supplying a vector of colours for each level of the group. Colours must be supplied if there are more than 7 experimental groups.

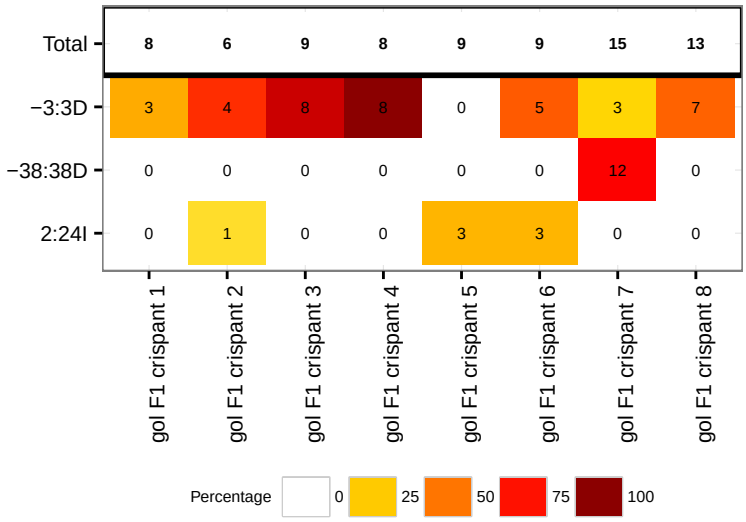
```
grp_clrs <- c("red", "purple")
plotFreqHeatmap(gol, top.n = 3, group = grp, group.colours = grp_clrs)
```



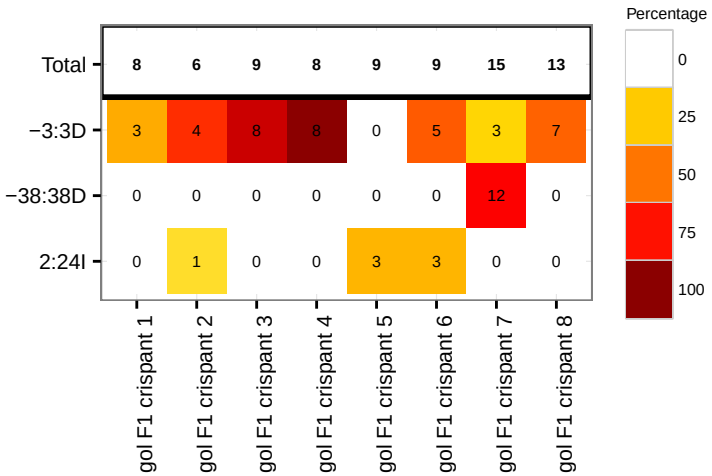
Controlling the appearance of the legend

The legend position is controlled via `legend.position`, which is passed to `ggplot2::theme`. Similarly `legend.key.height` controls the height of the legend. See the ggplot docs for more information.

```
plotFreqHeatmap(gol, top.n = 3, legend.position = "bottom")
```




```
plotFreqHeatmap(gol, top.n = 3,
                legend.key.height = ggplot2::unit(1.5, "lines"))
```



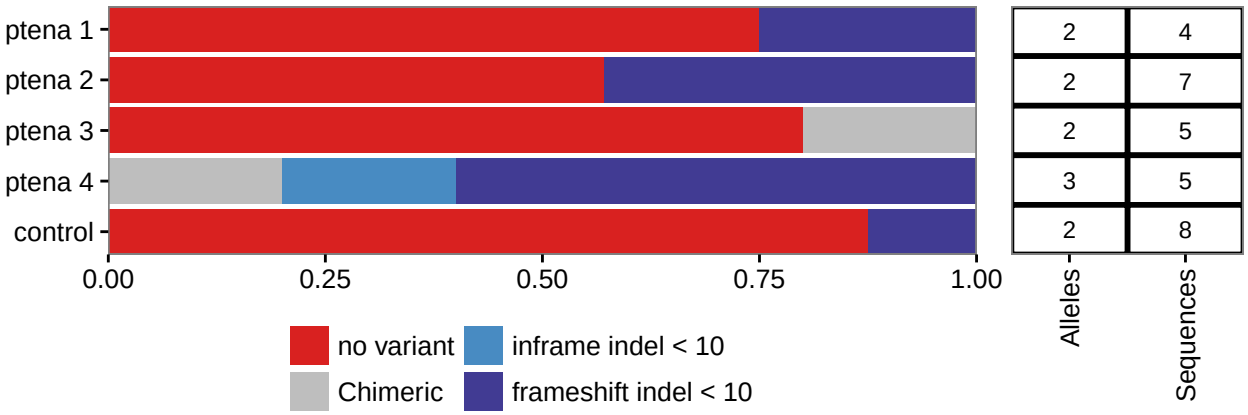
barplotAlleleFreqs

`barplotAlleleFreqs` includes two different colour schemes - a default rainbow scheme and a blue-red gradient. Note that the transcript database `txdb` must be passed by name as this function accepts ellipsis arguments.

Here `barplotAlleleFreqs` is run with the default parameters:

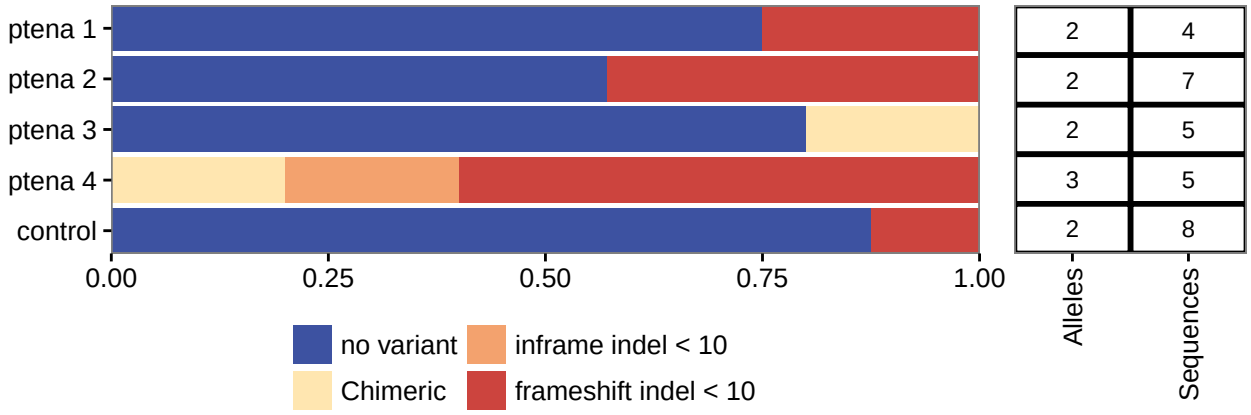
```
barplotAlleleFreqs(crispr_set, txdb = txdb)
```

```
## Looking up variant locations

## Loading required namespace: VariantAnnotation

## 'select()' returned many:1 mapping between keys and columns
## 'select()' returned many:1 mapping between keys and columns

## Classifying variants
```



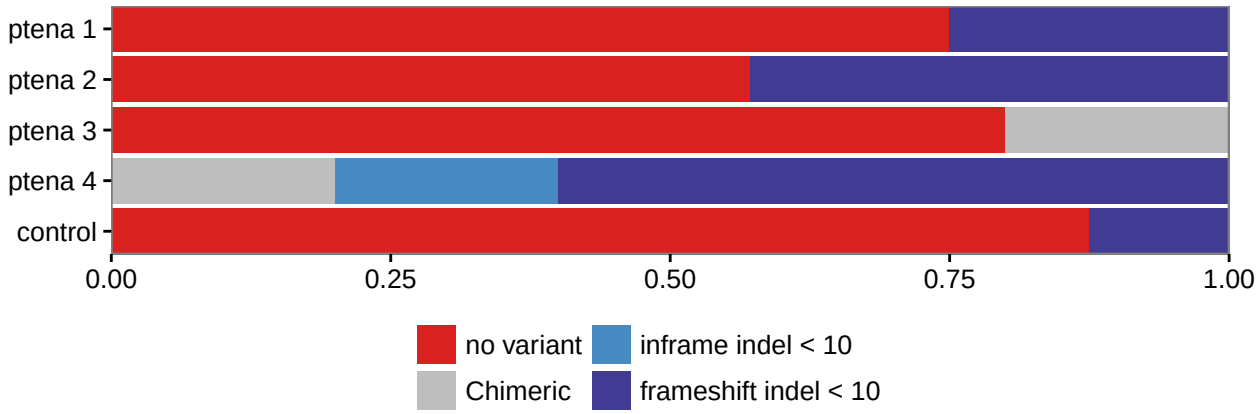
In this case `barplotAlleleFreqs` is run with the alternative palette.

```
barplotAlleleFreqs(crispr_set, txdb = txdb, palette = "bluered")
```



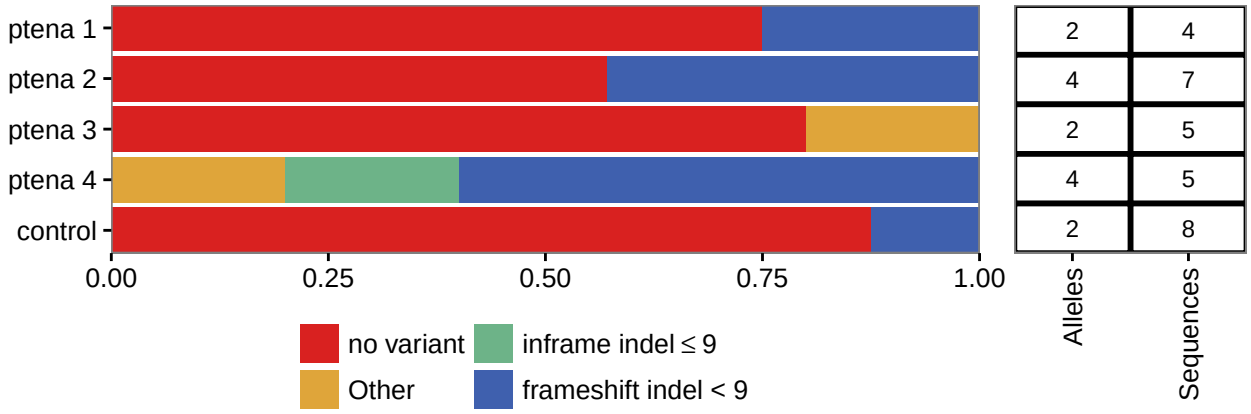
By default, a table of the number of sequences and alleles is plotted next to the barplot. This can be switched off. In this case, `barplotAlleleFreqs` will return an `ggplot` object, allowing further alteration of the appearance through the usual `ggplot2::theme` settings.

```
barplotAlleleFreqs(crispr_set, txdb = txdb, include.table = FALSE)
```



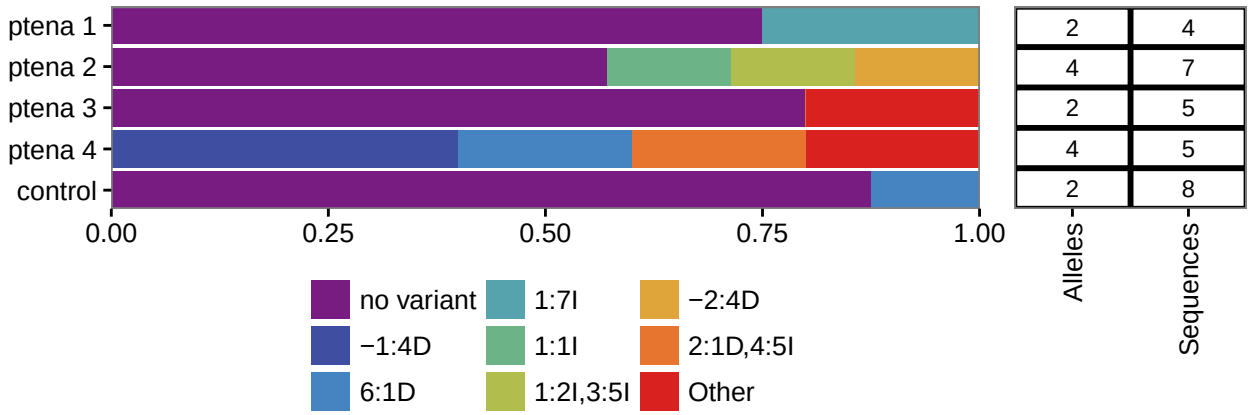
`barplotAlleleFreqs.CrisprSet` uses `VariantAnnotation::locateVariants` to look up the variant locations with respect to a transcript database. The default behaviour of `barplotAlleleFreqs.matrix` is to perform a naive classification of the variants as frameshift or non-frameshift by size. This approach ignores transcript scructure, but can be useful to give a faster overview, or in cases where the transcript structure is unknown.

```
var_counts <- variantCounts(crispr_set)
barplotAlleleFreqs(var_counts)
```



If the parameter `classify` is set to FALSE, the variants are plotted with no further aggregation. If there are more than seven variants, colours must be provided.

```
rainbowPal9 <- c("#781C81","#3F4EA1","#4683C1",
                 "#57A3AD","#6DB388","#B1BE4E",
                 "#DFA53A","#E7742F","#D92120")

barplotAlleleFreqs(var_counts, classify = FALSE, bar.colours = rainbowPal9)
```



An arbitrary classification can also be used. `CrispRVariants` provides some utility functions to assist in classifying variants. Note that methods of the `CrisprSet` class are accessed with `crisprSet$function()` rather than `function(crisprSet)`.

Here are some examples of variant classification:

```
# Classify variants as insertion/deletion/mixed
byType <- crispr_set$classifyVariantsByType()
byType
```

```
##            no variant                 -1:4D                  6:1D
##          "no variant"            "deletion"            "deletion"
##                 1:7I                  1:1I               1:2I,3:5I
##           "insertion"           "insertion" "multiple insertions"
##                -2:4D              2:1D,4:5I                 Other
##            "deletion"  "insertion/deletion"               "Other"
```

```
# Classify variants by their location, without considering size
byLoc <- crispr_set$classifyVariantsByLoc(txdb=txdb)
```

```
## Looking up variant locations
```

```
## 'select()' returned many:1 mapping between keys and columns
## 'select()' returned many:1 mapping between keys and columns
```

```
## Classifying variants
```

```
byLoc
```

```
##   no variant        -1:4D         6:1D         1:7I         1:1I    1:2I,3:5I
## "no variant"     "coding"     "coding"     "coding"     "coding"     "coding"
##        -2:4D    2:1D,4:5I        Other
##     "coding"     "coding"      "Other"
```

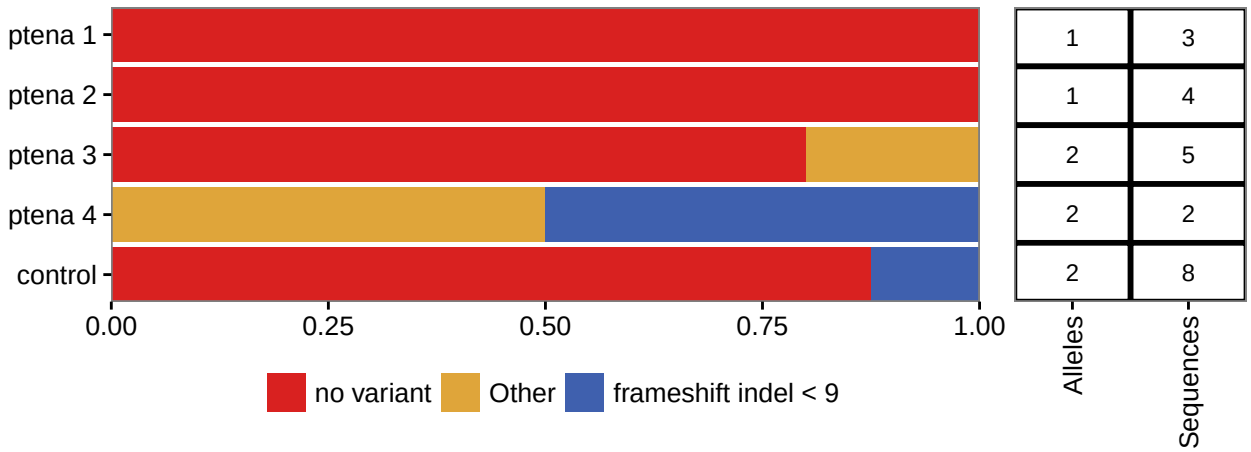
```
# Coding variants can then be classified by setting a size cutoff
byLoc <- crispr_set$classifyCodingBySize(byLoc, cutoff = 6)
byLoc
```

```
##             no variant                  -1:4D                  6:1D
##           "no variant" "frameshift indel < 6" "frameshift indel < 6"
##                  1:7I                   1:1I              1:2I,3:5I
## "frameshift indel > 6" "frameshift indel < 6" "frameshift indel > 6"
##                 -2:4D              2:1D,4:5I                  Other
## "frameshift indel < 6"    "inframe indel > 6"               "Other"
```

```
# Combine filtering and variant classification, using barplotAlleleFreqs.matrix
vc <- variantCounts(crispr_set)

# Select variants that occur in at least two samples
keep <- names(which(rowSums(vc > 0) > 1))
keep
```

```
## [1] "no variant" "6:1D"       "Other"
```

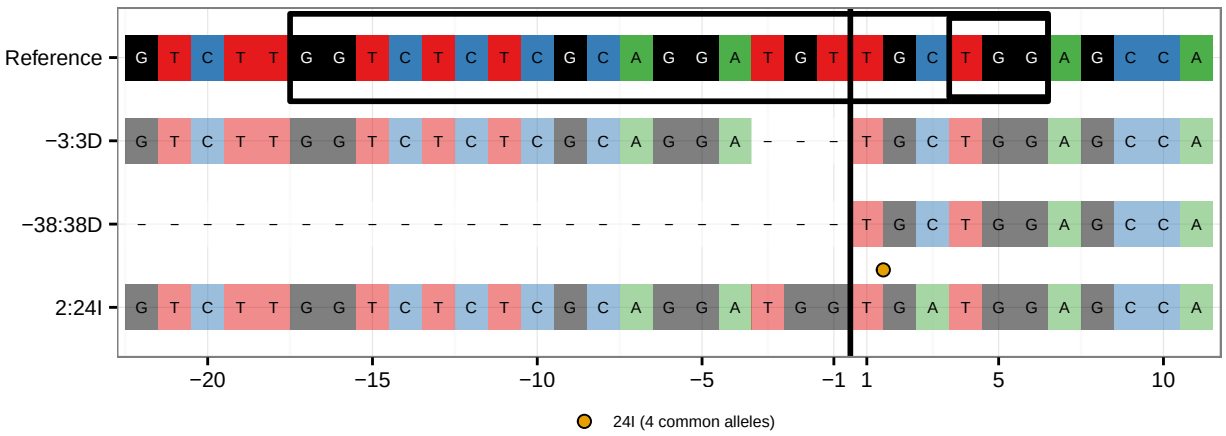
```
# Use this classification and the selected variants
barplotAlleleFreqs(vc[keep,], category.labels = byLoc[keep])
```



Other modifications

`plotAlignments` and `plotFreqHeatmap` both return `ggplot` objects, which can be adjusted via `theme()`. For example, to decrease the space between the legend and the plot:

```
p <- plotAlignments(gol, top.n = 3)
p + theme(legend.margin = ggplot2::unit(0, "cm"))
```



Using CrispRVariants plotting functions independently

The CrispRVariants plotting functions are intended to be used within a typical CrispRVariants pipeline, where the correct arguments are extracted from a CrisprSet object. However, with some data formatting, it is also possible to use these functions with standard R objects.

An example adapting `CrispRVariants::plotVariants` to display pairwise alignment can be found in the code accompanying the CrispRVariants paper: https://github.com/markrobinsonuzh/CrispRvariants_manuscript

Plot the reference sequence

Processing large data with CrispRVariants requires some time. It can be useful to first plot the reference sequence to check that the intended target location is specified. Here we use the reference sequence from the *gol* data set included in CrispRVariants. Any Biostrings::DNAString can be used. Note that CrispRVariants::plotAlignments accepts elliptical arguments in its signature, so non-signature arguments must be supplied by name. The code below shows the minimum arguments required for running CrispRVariants::plotAlignments.

```
# Get a reference sequence
library("CrispRVariants")
data(gol_clutch1)
ref <- gol$ref

#Then to make the plot:
plotAlignments(ref, alns = NULL, target.loc = 22, ins.sites = data.frame())
```