CrispRVariants User Guide

Helen Lindsay, Mark Robinson

27 October 2020

Package

CrispRVariants 1.18.0

Contents

1 Introduction

The CRISPR-Cas9 system is an efficient method of introducing mutations into genomic DNA. A guide RNA directs nuclease activity to an approximately 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.

2 Quickstart

This section is intended for people familiar with mapping reads and working with core Bioconductor classes. See the case study in the next section for a complete step-by-step analysis.

The `CrisprSet` class stores aligned reads which have been trimmed to a target region along with annotations of where insertion and deletions are located with respect to a specified location. `CrisprSet` objects are created using the functions `readsToTarget` (for a single target region) and `readsToTargets` (for multiple target locations). The following objects are needed to create a `CrisprSet` for a single guide sequence. For multiple guides, the equivalent parameters to **target** and **reference** are named **targets** and **references** respectively.

- *reads* - may be a vector of bam filenames, a `GAlignments` object or a `GAlignmentsList` object. Bam files are assumed to represent individual experimental samples (possibly containing reads from more than one guide). If the bam files contain multiplexed reads that should be separated into groups, first read the alignments into R, separate as required and then provide the separated alignments to `readsToTarget`. (See package GenomicAlignments for more details about `GAlignments` and `GAlignmentsList` objects).

- *target* - A `GRanges` object indicating the genomic range to analyse. The sequence name and coordinates should match regions found in the bam file(s). For `readsToTarget`, the `target` must contain a single range. The target range can be found by searching BLAST or BLAT for the guide sequence and extending the found range if desired - be careful that the genome used by BLAST matches the genome used for mapping! Alternatively, the target region can be found by mapping the guide sequence to the genome or amplicon reference with a short read aligner (we typically use **bwa fastmap**); or if reference sequence is not too large by reading the reference sequence into R and using `gregexpr` to search the reference for the guide sequence. (See package GenomicRanges for more details about `GRanges` objects).

- *reference* - A `DNAString` object, containing the reference sequence at the target location. This can be fetched from the genome using the command line tool `samtools faidx` or from within R using the Bioconductor package `RSamtools`; fetched from a `BSgenome` object using `Biostrings::getSeq`; or reconstructed from a bam file using the CrispRVariants function `refFromAlns` providing the bam file has an MD tag. (See package Biostrings for more details about `DNAString` objects).
- *target.loc* This is the base that should be considered base zero, with respect to the target. For example, if using a 23 nucleotide guide + PAM sequence as the reference, the target.loc would be 17, meaning that bases 17 and 18 are numbered -1 and 1 respectively. If considering the 23bp guide region plus 10 bases on both sides, the target.loc would be 27.

Other important `readsToTarget` parameters:

- If the bam file contains **paired end** data, set `collapse.pairs = TRUE` so that each read pair is only counted once.
- By default CrispRVariants is conservative when dealing with **large gaps**, and requires one mapped endpoint to be within 5 bases of the target.loc. To change this, increase the value of the parameter chimera.to.target, e.g. `chimera.to.target = 200`
- If the parameter **names** is set, the sample names will be used when plotting.
- If using *readsToTargets*, providing *primer.ranges* (amplicon ranges) as well as *targets* will help CrispRVariants disambiguate reads from nearby guide sequences.
- By default, CrispRVariants displays the variant alleles with respect to the **strand** of the target. To change this behaviour, set `orientation = "positive"` to always display with respect to the plus strand or `orientation = "opposite"` to display the opposite strand to the target.
- Long gaps are typically mapped in separate segments. These are called **chimeric** mappings. In this release CrispRVariants introduces an experimental parameter `chimeras = "merge"`, which reconstructs a linear alignment for the simplest case of two aligned regions separated by a large gap with at most 10 bases multi-mapped to both segments. Setting `chimeras = "merge"` means that these simple, "long-gap" chimeras can be counted and displayed separately from more complex chimeric reads. However, be aware that the exact endpoints of the gaps may be ambiguous and we do not at present have a method for indicating ambiguous mapping.
- If considering a **large region**, e.g. the entire amplicon, use the minoverlap parameter to set the minimum number of aligned positions overlapping the target required for a read to be counted. For example, `minoverlap = 10` means that reads with an aligned range spanning at least 10 bases of the target will be counted.

Assuming the above parameters are defined, the following code will set up a CrisprSet object and plot the variants:

```
crispr_set <- readsToTarget(reads, target = target, reference = reference,
                            target.loc = target.loc)
plotVariants(crispr_set)
# or use plotVariants(crispr_set, txdb) to additionally show the target
# location with respect to the transcripts if a Transcript Database
# txdb is available
```

3 Case study: Analysis of ptena mutant spectrum in zebrafish

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

3.1 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 the "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 example, there is only 1 sequence, which will be output to 1 file:

```
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).

3.2 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.

3.3 List the BAM files

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. Here we read the bam filenames from a metadata table which also contains sample names and experimental grouping information. Note that we could also have used the bam filenames listed above.

```
# 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
```

3.4 Create the target location and reference sequence

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.

The coordinates of the region of interest can be obtained by running BLAST or BLAT on the guide sequence or by mapping the guide sequence to the reference sequence. The coordinates, or "target" should be represented as a **GenomicRanges::GRanges** object. This can be created directly, but here we will import the coordinates of the guide sequence from a BED file using the **rtracklayer** package. The import() commmand below 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
```

The 23bp (including PAM) *ptena* guide sequence used in this experiment is located on chromosome chr17 from 23648474-23648496. We prefer to analyse a slightly larger region. Below, we'll extend the guide region by 5 bases on each side when counting variants. Note that the expected cut site (used later for labeling variants), after extension, isat base 22 with respect to the start of the new target sequence.

```
gdl <- GenomicRanges::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 (for an alternative method, see the note for Windows and Galaxy users below). 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 object
## seq: GCCATGGGCTTTCCAGCCGAACGATTGGAAGGT
```

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

3.4.1 Note for Windows and Galaxy Users

If you do not have a copy of the genome you used for mapping on the computer you are using to analyse your data, or you cannot install *samtools* because you are working on Windows, *CrispRVariants* provides an alternative, albeit slower, method for fetching the reference sequence:

```
# First read the alignments into R.  The alignments must include
# the read sequences and the MD tag
alns <- GenomicAlignments::readGAlignments(bam_fnames[[1]],
          param = Rsamtools::ScanBamParam(tag = "MD", what = c("seq", "flag")),
          use.names = TRUE)

# Then reconstruct the reference for the target region.
# If no target region is given, this function will reconstruct
# the complete reference sequence for all reads.
rfa <- refFromAlns(alns, gdl)

# The reconstructed reference sequence is identical to the sequence
# extracted from the reference above
print(rfa == reference)
## [1] TRUE
```

Note that the object `alns` created above can be directly passed to the function `readsToTarget` (see below) instead of the bam filenames. If there is more than one bam file, `readsToTarget` can also accept a `GAlignmentsList` object (see the GenomicAlignments package) for more details).

3.5 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" "array"
```

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.

3.6 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)

# 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")))
## Warning in min(xranges): no non-missing arguments to min; returning Inf
## Warning in max(xranges): no non-missing arguments to max; returning -Inf
## Warning in max(yranges): no non-missing arguments to max; returning -Inf
## Warning: Vectorized input to `element_text()` is not officially supported.
## Results may be unexpected or may change in future versions of ggplot2.
```

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).

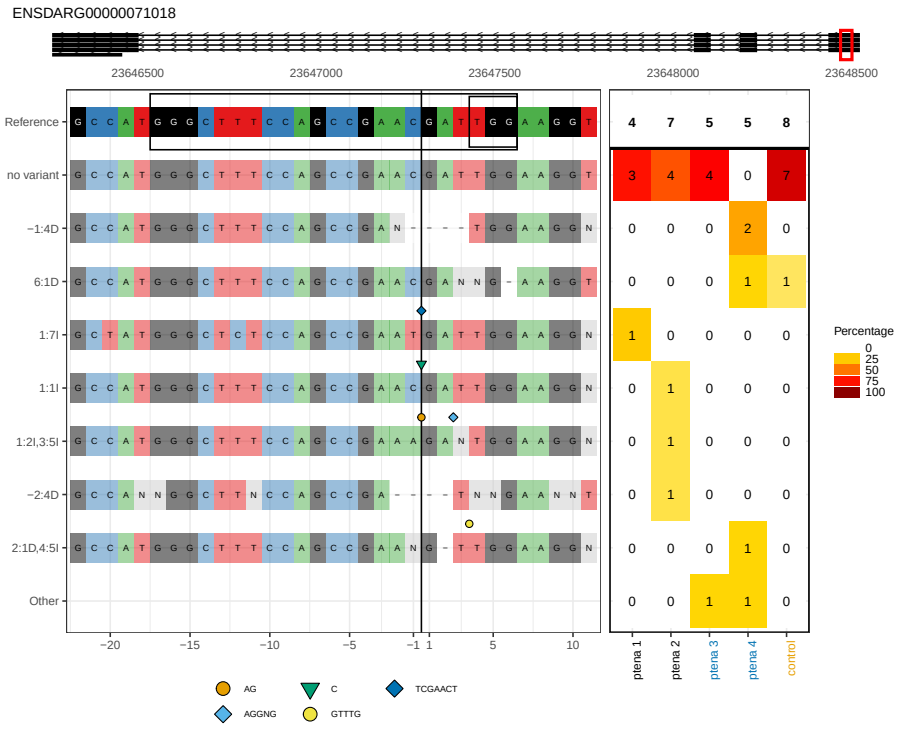


Figure 1: (Top) schematic of gene structure showing guide location (left) consensus sequences for variants (right) variant counts in each embryo

3.7 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.

3.8 Get consensus alleles

The consensus sequences for variant alleles can be accessed using **consensusSeqs**. This function allows filtering by variant frequency or read count, as for **plotAlignments** and **plotFreqHeatmap**. Consensus alleles are returned with respect to the positive strand.

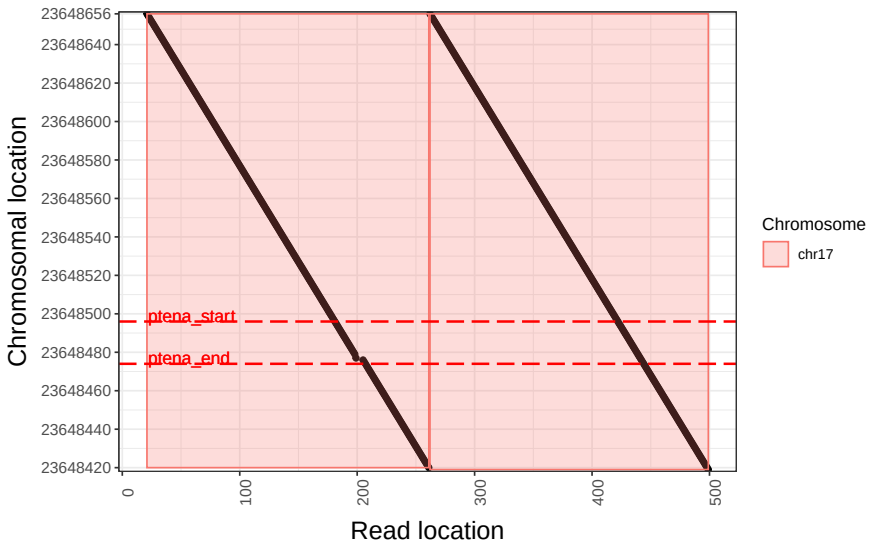
```
sqs <- consensusSeqs(crispr_set)
sqs
## DNAStringSet object of length 8:
##     width seq                                                 names
## [1]    33 ACCTTCCAATCGTTCGGCTGGAAAGCCCATGGC                  no variant
## [2]    29 NCCTTCCANTCGGCTGGAAAGCCCATGGC                      -1:4D
## [3]    32 ACCTTCNNTCGTTCGGCTGGAAAGCCCATGGC                   6:1D
## [4]    40 NCCTTCCAATCAGTTCGAATTCGGCTGGAGAGCCCATAGC           1:7I
## [5]    34 NCCTTCCAATCGGTTCGGCTGGAAAGCCCATGGC                 1:1I
## [6]    40 NCCTTCCANCNCCTTCCTTTTCGGCTGGAAAGCCCATGGC           1:2I,3:5I
## [7]    29 ANNTTCNNATCGGCTGGNAAGCCNNTGGC                      -2:4D
## [8]    37 NCCTTCCACAAACACNTTCGGCTGGAAAGCCCATGGC              2:1D,4:5I

# The ptena guide is on the negative strand.
# Confirm that the reverse complement of the "no variant" allele
# matches the reference sequence:
Biostrings::reverseComplement(sqs[["no variant"]]) == reference
## [1] TRUE
```

3.9 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
```

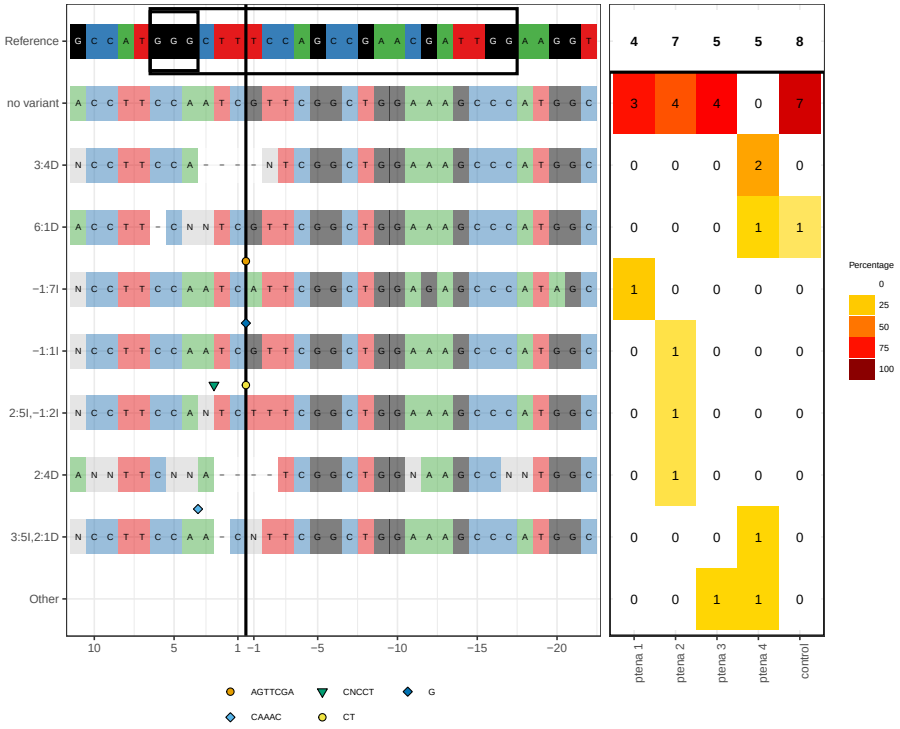
We see that the mutation effiency for "ptena 4" is now 75%, i.e. 3 genuine variant sequences, 1 sequence counted as "non-variant" because it occurs in the control, and the chimeric read excluded completely

4 Choosing the strand for display

CrispRVariants is capable of tabulating variants with respect to either strand. By default, variant alleles are displayed with respect to the target strand, i.e. sequences for a guide on the negative strand are reverse complemented for display. For some applications it may be preferable to display the variants on the opposite strand, for example if a guide on the negative strand is used to target a gene on the positive strand. The display strand is controlled using the **orientation** parameter in **readsToTarget(s)** during initialization.

To illustrate, we will plot the variants for *ptena* on the positive strand. Note that the only changes to the initialization code is the *orientation* parameter. In particular, the *target.loc* is still specified with respect to the guide sequence and the reference is still the guide sequence, not its reverse complement.

```
crispr_set_rev <- readsToTarget(bam_fnames, target = gdl, reference = reference,
                                names = md$Short.name, target.loc = 22,
                                orientation = "opposite")
plotVariants(crispr_set_rev)
```




```
## TableGrob (2 x 1) "arrange": 2 grobs
##   z     cells    name                grob
## 1 1 (1-1,1-1) arrange rect[GRID.rect.246]
## 2 2 (2-2,1-1) arrange     gtable[arrange]
```

Note that variants are labelled with respect to their leftmost coordinate, so the labelled variant location changes when plotting on the opposite strand.

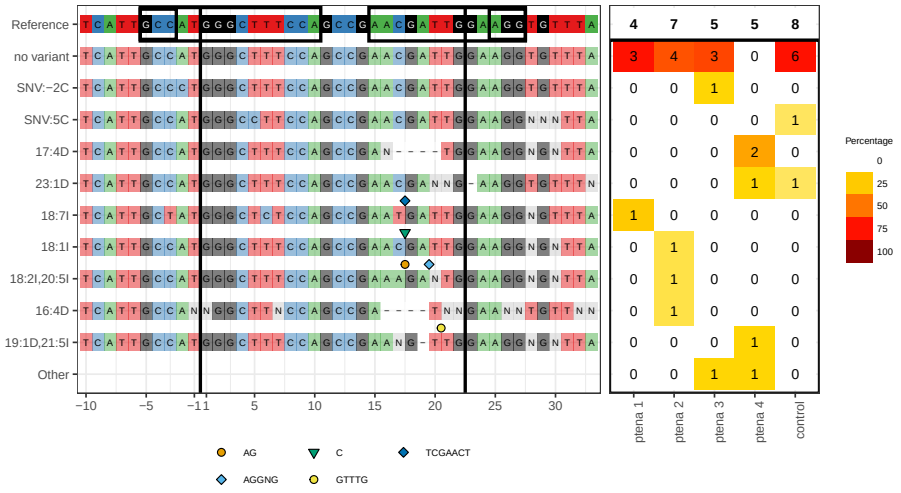
5 Multiple guides

`CrispRVariants` accepts an arbitrarily long reference sequence and target region. By default, reads must span the target region to be counted. Since v1.3.6, a new argument `minoverlap` to `readsToTarget` is available, which allows reads which do not span the target region to be counted, provided they have at least `minoverlap` aligned bases overlapping the target. This is particularly important when the target region is close to or greater than the sequencing read length.

When using CrispRVariants with multiple guides, initialisation of a `CrisprSet` object is done as for a single guide, but with the `reference` and `target` parameters corresponding to a region spanning the guides of interest. One important parameter is the target location, or `target.loc` which determines how the variant alleles are numbered. For a single guide, position zero would typically be the cut site. With multiple guides, possible zero points might include the cut site of the leftmost guide or the first base of the amplified sequence. Multiple guide sequences are first indicated at the stage of plotting.

In the example below, we reuse the *ptena* data used in the case study. In this experiment, a single guide was injected. However, for illustrative purposes we will treat the data as if it was a paired injection of two nearby guides.

```
# We create a longer region to use as the "target"
# and the corresponding reference sequence
gdl <- GenomicRanges::resize(gd, width(gd) + 20, fix = "center")
reference <- Biostrings::DNAString("TCATTGCCATGGGCTTTCCAGCCGAACGATTGGAAGGTGTTTA")

# At this stage, target should be the entire region to display and target.loc should
# be the zero point with respect to this region
crispr_set <- readsToTarget(bam_fnames, target = gdl, reference = reference,
                            names = md$Short.name, target.loc = 10,
                            verbose = FALSE)

# Multiple guides are added at the stage of plotting
# The boundaries of the guide regions must be specified with respect to the
# given target region
p <- plotVariants(crispr_set,
       plotAlignments.args = list(pam.start = c(6,35),
                                  target.loc = c(10, 32),
                                  guide.loc = IRanges::IRanges(c(6, 25),c(20, 37))))
```




```
p
## TableGrob (2 x 1) "arrange": 2 grobs
##   z     cells    name                grob
```

```
## 1 1 (1-1,1-1) arrange rect[GRID.rect.390]
## 2 2 (2-2,1-1) arrange     gtable[arrange]
```

In the above call to `plotAlignments`, `pam.start` and `pam.end` control where the box around the PAM sequence is drawn, `target.loc` controls where vertical lines are drawn (note this does not have to match the `target.loc` passed to `readsToTarget`), and `guide.loc` controls where the box around the guide is drawn.

6 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`.

6.1 Filtering data in plotVariants

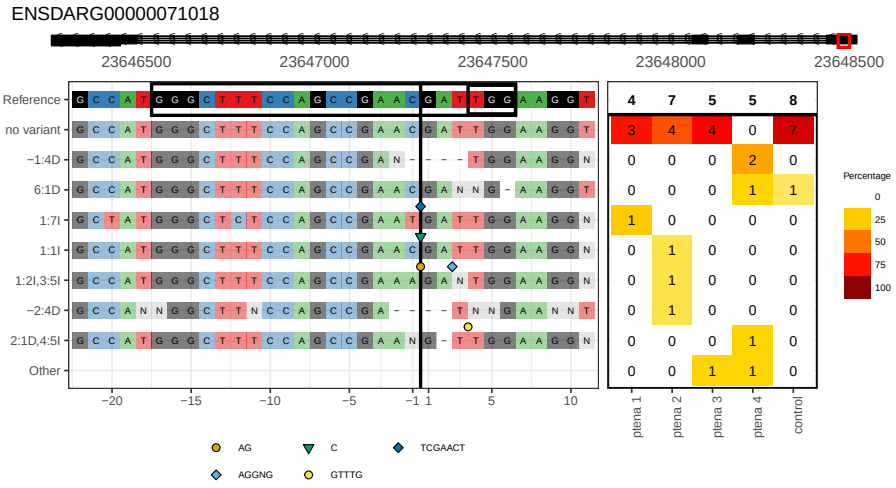
For the following examples, we will use the *ptena* data set. We must first load the data and create a `CrispRVariants::CrisprSet` object.

```
# 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)
```

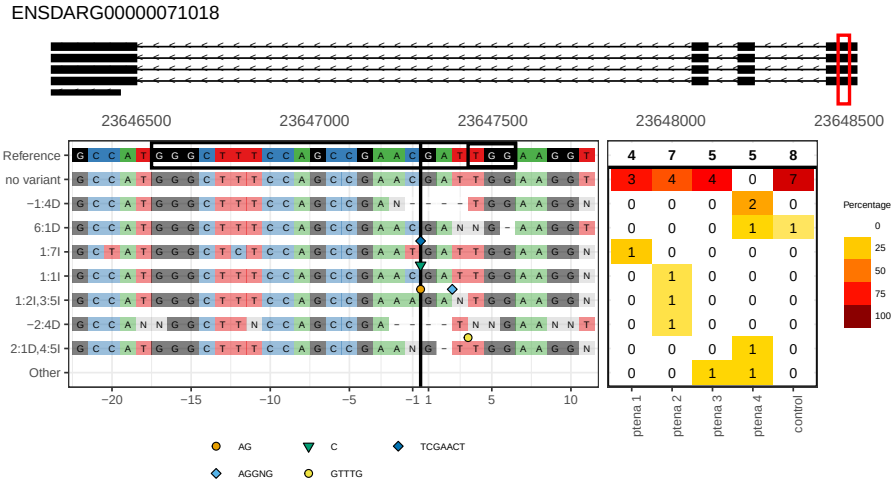
Here is the ptena data set plotted with default options:

```
p <- plotVariants(crispr_set, txdb = txdb)
## 'select()' returned 1:many mapping between keys and columns
## 'select()' returned 1:many mapping between keys and columns
```



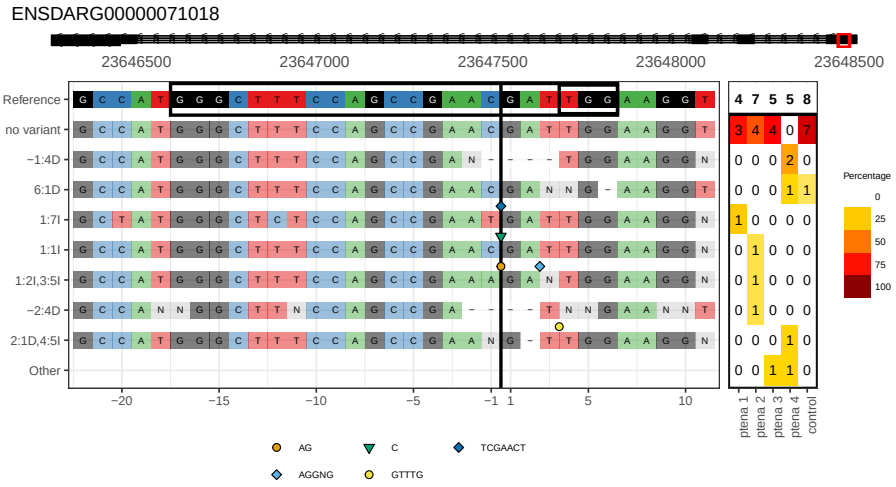
The layout of this plot is controlled mainly by two parameters: `row.ht.ratio` and `col.wdth.ratio`. `row.ht.ratio` (default `c(1,6)`) controls the relative sizes of the transcript plot and the other plots. Below we show how to change the ratio so that the transcript plot is relatively larger:

```
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))
```

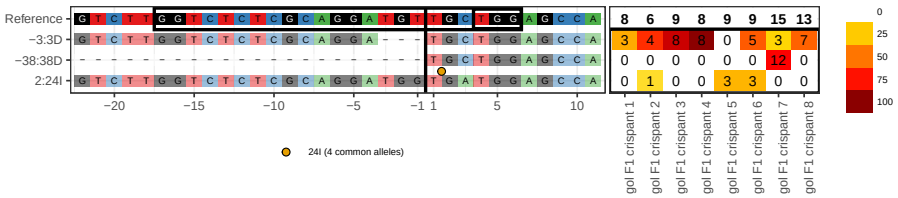


The remaining examples in this section use the *gol* data set.

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

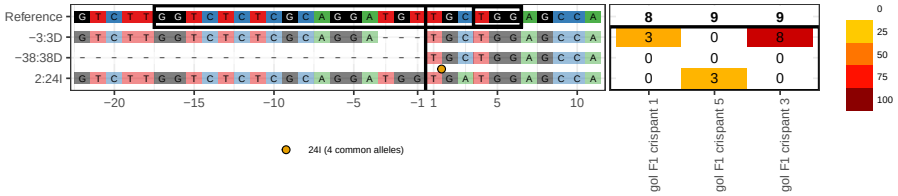
The data used in `plotAlignments` and `plotFreqHeatmap` can be filtered by either frequency via `min.freq`, count via `min.count`, or to show a set number of alleles sorted by frequency, via `top.n`. Within `plotVariants`, these filtering options need to be set for both `plotAlignments` and `plotFreqHeatmap`. 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"))
```



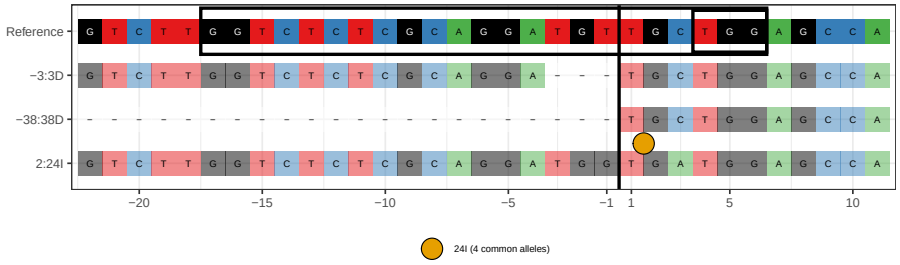

```
## TableGrob (2 x 1) "arrange": 2 grobs
##   z     cells    name                grob
## 1 1 (1-1,1-1) arrange rect[GRID.rect.1174]
## 2 2 (2-2,1-1) arrange      gtable[arrange]
```

6.2 plotAlignments

6.2.1 Insertion symbols

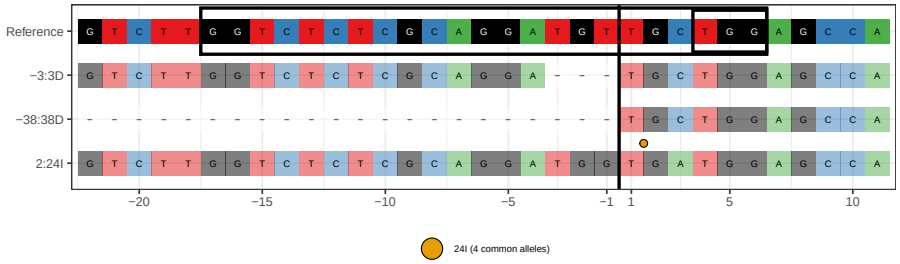
The symbols indicating insertions are controlled by four parameters. `ins.size` (default 3) controls the size of the symbols within the plot area.

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



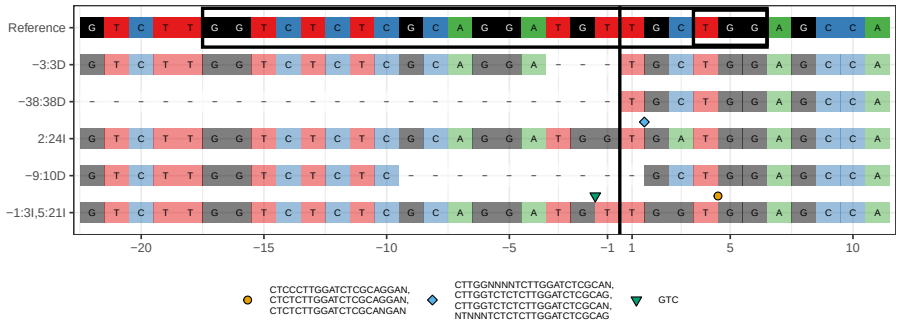
By default the symbols in the legend are the same size as those in the plot, but this can be controlled separately with `legend.symbol.size`.

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



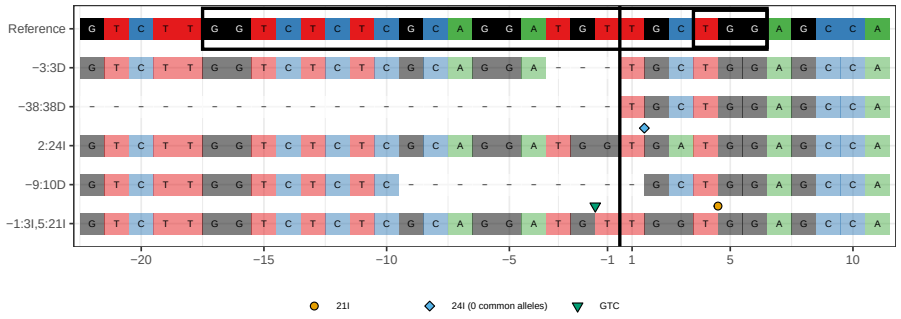
As long sequences can make the plot difficult to read, by default only the length of insertions greater then 20bp is shown. This can be changed with the `max.insertion.size` parameter. If there is more than one allele, the number of (frequent) alleles is indicated.

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



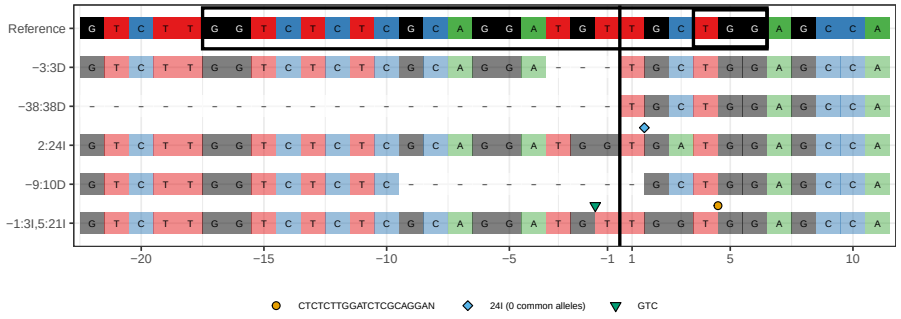
Finally, the parameter `min.insertion.freq` (default 5%) controls how many alleles are displayed at each insertion locus. In large data sets, there will be a substantial proportion of reads with sequencing errors, and we may only wish to display the most common sequences.

```
# 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.

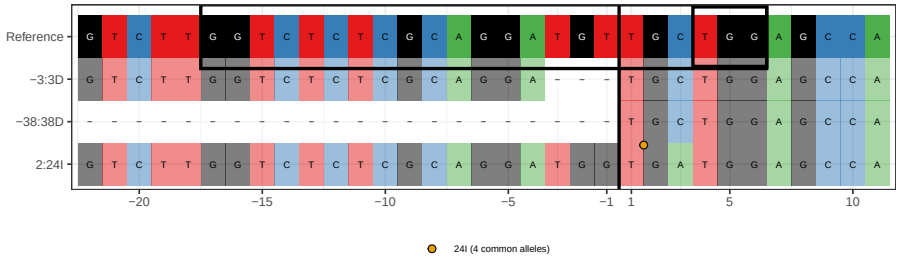
```
plotAlignments(gol, top.n = 5, max.insertion.size = 25, min.insertion.freq = 50)
```



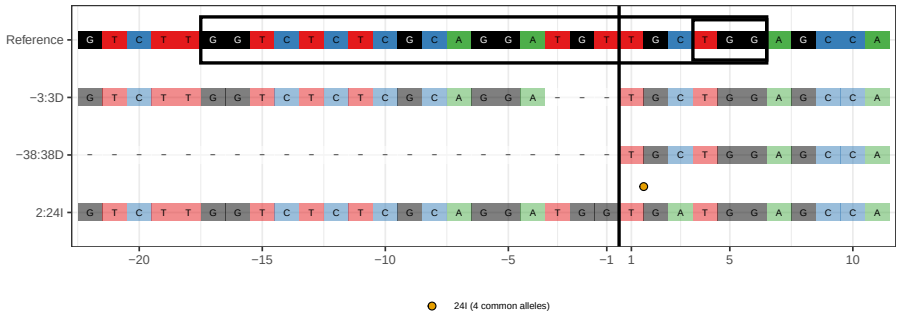
6.2.2 Whitespace between rows

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)
```



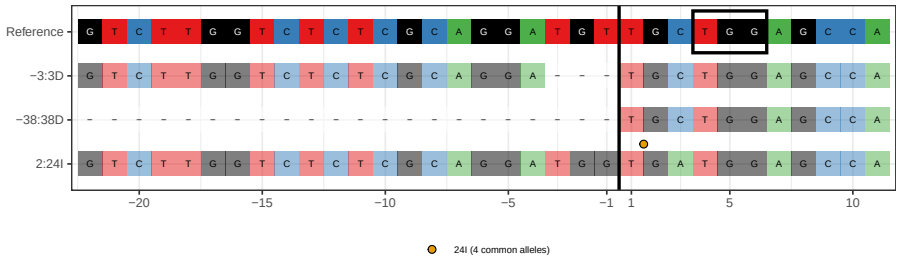

```
# More white space between rows
plotAlignments(gol, top.n = 3, tile.height = 0.3)
```



6.2.3 Box around guide

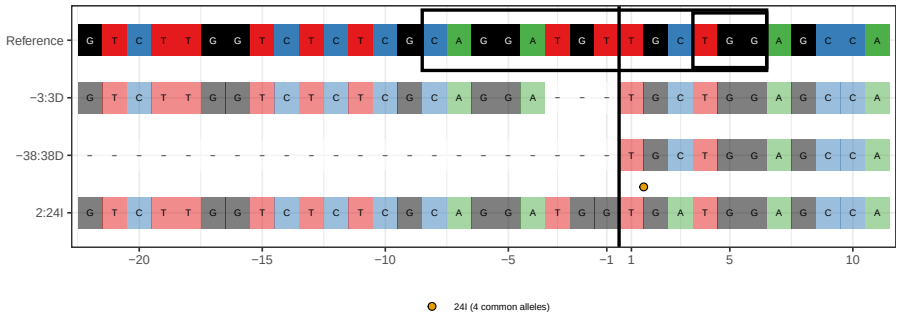
The black box around the guide sequence can be removed by setting `highlight.guide = FALSE`.

```
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.

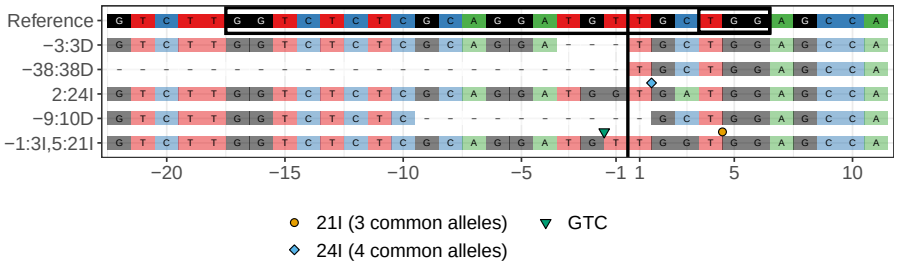
```
library(IRanges)
guide <- IRanges::IRanges(15,28)
plotAlignments(gol, top.n = 3, guide.loc = guide)
```



6.2.4 Text sizes

The text showing bases within the alignment plot is controlled by `plot.text.size` (default 2), 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).

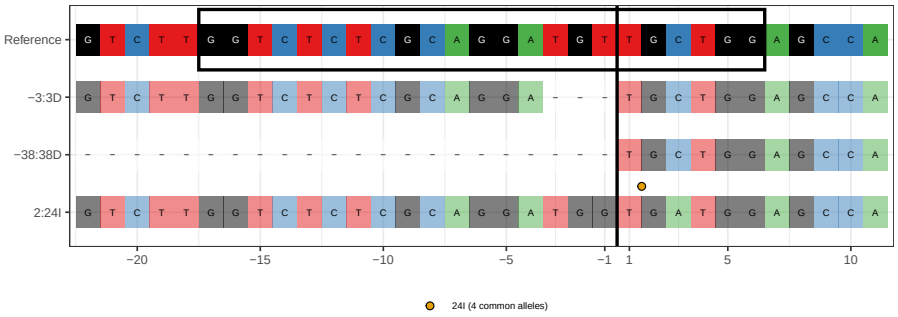
```
# Here we increase the size of the axis labels and make
# two columns for the legend
plotAlignments(gol, top.n = 5, axis.text.size = 12,
               legend.text.size = 12, legend.cols = 2)
```



6.2.5 Box around PAM

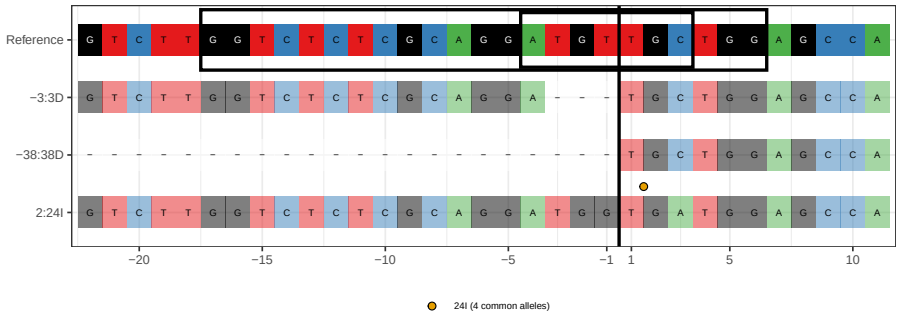
The argument `highlight.pam` determines whether a box around the PAM should be drawn.

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



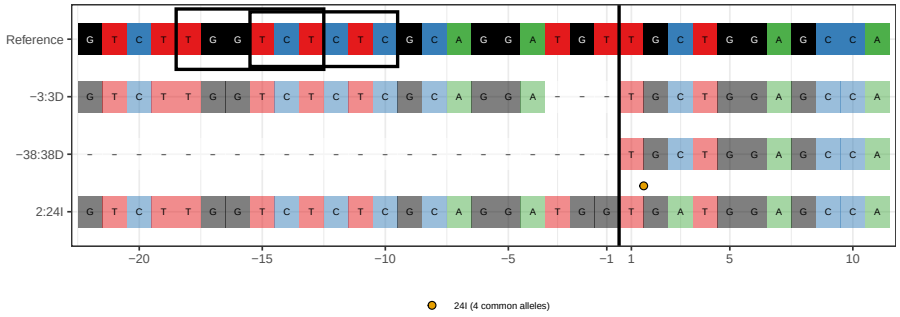
By default this box is drawn 3 nucleotides downstream of the `target.loc`. Other applications might require a different region highlighted. This can be achieved by explicitly setting the start and end positions of the box, with respect to the reference sequence.

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



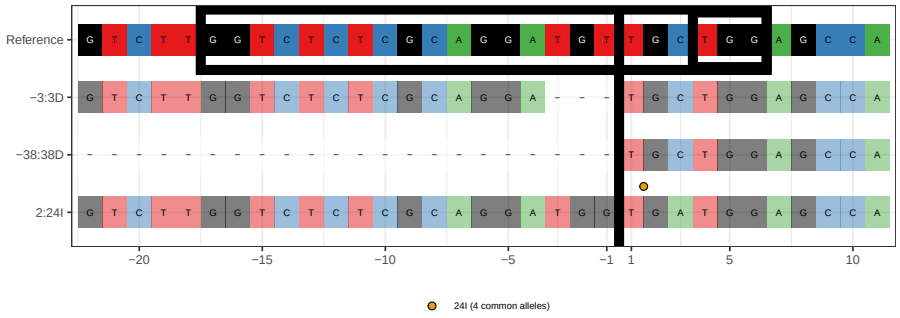
The boxes around the guide and the PAM can both be changed to arbitrary locations, however note that the guide box is specified by a ranges object whilst the PAM box is specified by start and end coordinates. Both coordinates are with respect to the start of the reference sequence. The box around the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

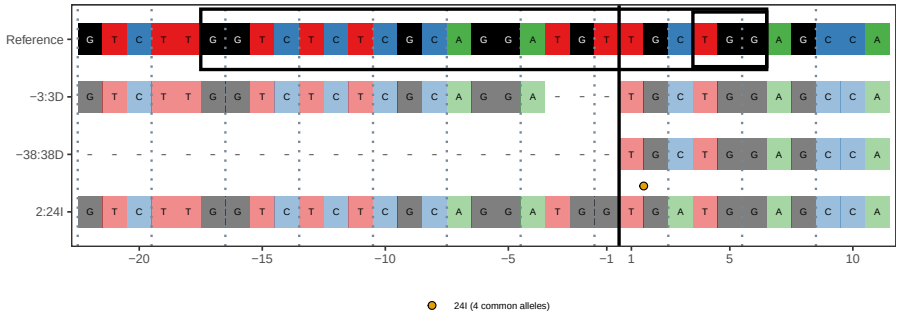
```
plotAlignments(gol, top.n = 3, line.weight = 3)
```



6.2.6 Add a codon frame

If the codon frame with respect to the first base of the target region is known, it can be added to `plot.alignments` using the argument `codon.frame`

```
plotAlignments(gol, top.n = 3, codon.frame = 1)
```



6.2.7 Other modifications

To retreive the information used in `plotAlignments` when starting from a `CrisprSet` object, use the argument `create.plot = FALSE`.

```
plot_data <- plotAlignments(gol, top.n = 3, create.plot = FALSE)
names(plot_data)
# This data can be modified as required, then replotted using:
do.call(plotAlignments, plot_data)
```

6.3 plotFreqHeatmap

`plotFreqHeatmap` produces a heatmap of the counts or proportions of the variant alleles. Typically, `plotFreqHeatmap` is passed a `CrisprSet` object, but it can also accept a `matrix` if greater flexibility is required (see below). As `plotFreqHeatmap` returns a `ggplot` object, it can also be modified using standard `ggplot2` syntax. For example, we add a title to the plot below.

By default, when given an object of class `CrisprSet`, `plotFreqHeatmap` shows the allele counts and the header shows the total number of on-target reads in each sample. For example, the following code shows the three most common variant alleles in the gol dataset. The header here does not equal the sum of the columns as not all variants are shown in the plot.

```
# Save the plot to a variable then add a title using ggplot2 syntax.
# If the plot is not saved to a variable the unmodified plot is displayed.
p <- plotFreqHeatmap(gol, top.n = 3)
## Warning in min(xranges): no non-missing arguments to min; returning Inf
## Warning in max(xranges): no non-missing arguments to max; returning -Inf
## Warning in max(yranges): no non-missing arguments to max; returning -Inf
p + labs(title = "A. plotFreqHeatmap with default options")
```

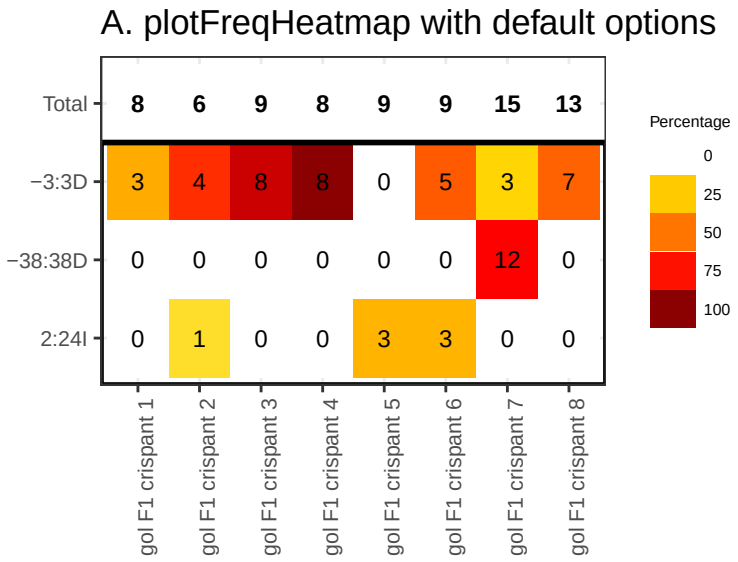


Figure 2: plotFreqHeatmap with default options

6.3.1 Plotting allele proportions

When calling `plotFreqHeatmap` with a `CrisprSet` object the `type` argument controls the information shown in text in the heatmap. Setting `type = "counts"` (the default) shows allele counts, setting `type = "proportions"` shows allele proportions. This also affects the default header values. When `type = "proportions"` the header shows the column sums, i.e. the percentage of the total number of reads shown in the plot.

```
plotFreqHeatmap(gol, top.n = 3, type = "proportions")
## Warning in min(xranges): no non-missing arguments to min; returning Inf
## Warning in max(xranges): no non-missing arguments to max; returning -Inf
## Warning in max(yranges): no non-missing arguments to max; returning -Inf
```

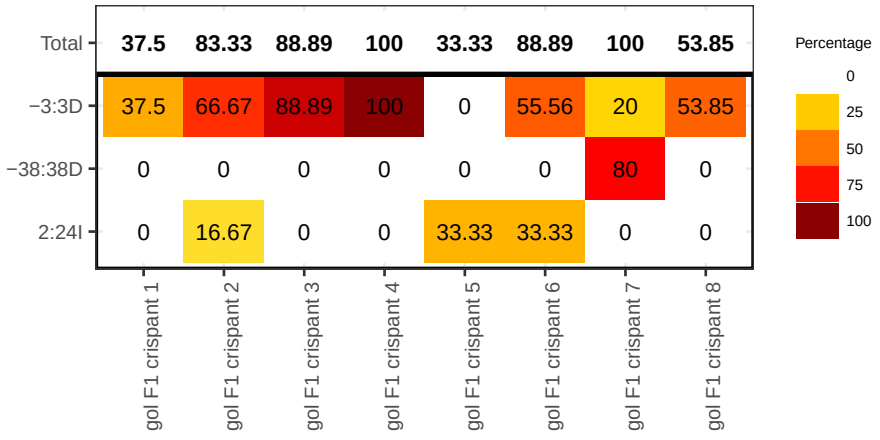


Figure 3: plotFreqHeatmap showing allele proportions

6.3.2 Changing the header

There are three standard options for the header when calling `plotFreqHeatmap` with an object of class `CrisprSet`:

- `header = "default"` shows total read counts when `type = "counts"` or column sums when `type = "proportions"`. See the examples above.
- `header = "counts"` shows total read counts. If variants are excluded, the values in the header do necessarily equal the column totals. For example, see plot "C. Modified plotFreqHeatmap" below.
- `header = "efficiency"` shows the mutation efficiency, i.e. the percentage of reads that have an insertion or deletion variant. The mutation efficiency is calculated using the default options of the function `mutationEfficiency`.

6.3.3 Heatmap colours

The tiles may be coloured by either the percentage of the column totals (default), or by the counts, by setting `as.percent = FALSE`. For example, see plot "B. coloured X labels with tiles coloured by count" below and contrast with plot "A. plotFreqHeatmap with default options" above.

6.3.4 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)
p <- plotFreqHeatmap(gol, top.n = 3, group = grp, as.percent = FALSE)
## Warning in min(xranges): no non-missing arguments to min; returning Inf
## Warning in max(xranges): no non-missing arguments to max; returning -Inf
## Warning in max(yranges): no non-missing arguments to max; returning -Inf
## Warning: Vectorized input to `element_text()` is not officially supported.
## Results may be unexpected or may change in future versions of ggplot2.
p + labs(title = "B. coloured X labels with tiles coloured by count")
```

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")
p <- plotFreqHeatmap(gol, top.n = 3, group = grp, group.colours = grp_clrs,
                type = "proportions", header = "counts",
                legend.position = "bottom")
## Warning in min(xranges): no non-missing arguments to min; returning Inf
## Warning in max(xranges): no non-missing arguments to max; returning -Inf
## Warning in max(yranges): no non-missing arguments to max; returning -Inf
## Warning: Vectorized input to `element_text()` is not officially supported.
```

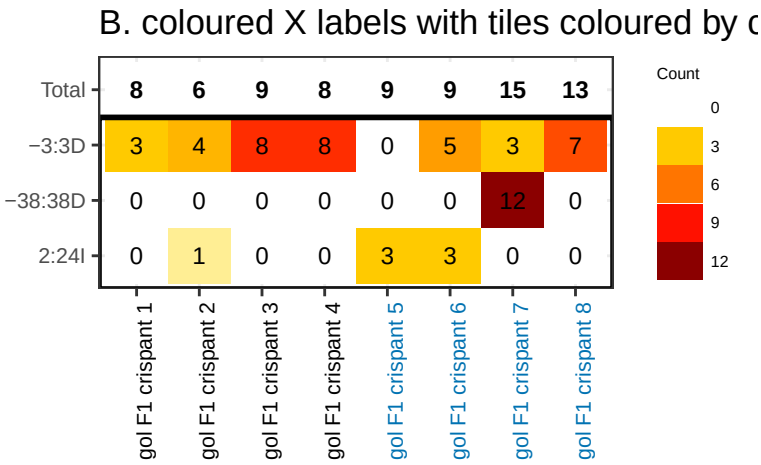


Figure 4: **plotFreqHeatmap with X-axis labels coloured by experimental group and tiles coloured by count instead of proportion**

```
## Results may be unexpected or may change in future versions of ggplot2.
p <- p + labs(title = "C. Modified plotFreqHeatmap")
p
```

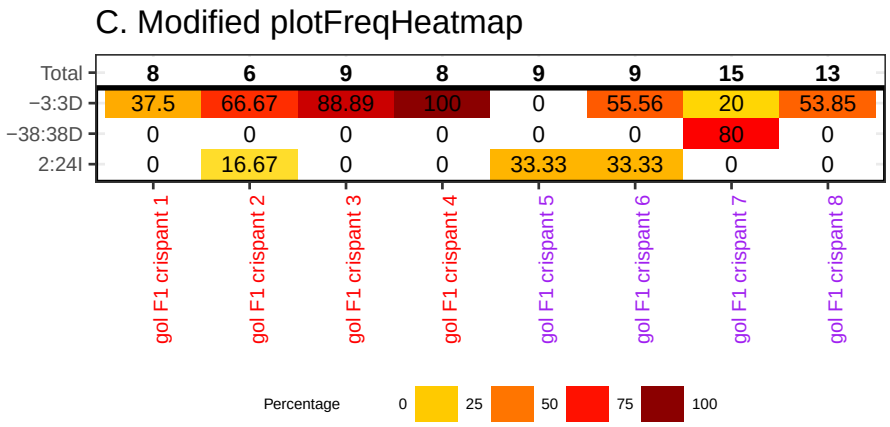


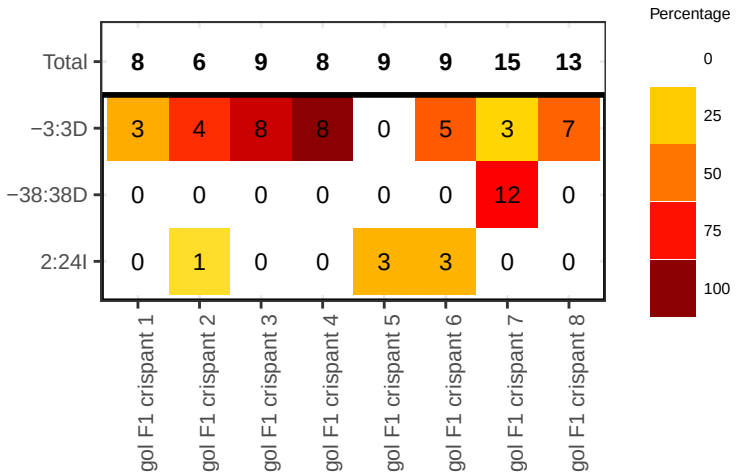
Figure 5: **plotFreqHeatmap with labels showing allele proportions, header showing counts per sample and modified legend position**

6.3.5 Controlling the appearance of the legend

The legend position is controlled via the `plotFreqHeatmap` argument `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.key.height = ggplot2::unit(1.5, "lines"))
## Warning in min(xranges): no non-missing arguments to min; returning Inf
```

```
## Warning in max(xranges): no non-missing arguments to max; returning -Inf
## Warning in max(yranges): no non-missing arguments to max; returning -Inf
```



An additional example where the legend is placed at the bottom is shown above in plot C named "Modified plotFreqHeatmap" above.

6.3.6 Further customisation

The function `variantCounts` returns a matrix of allele counts or proportions which can be passed to `plotFreqHeatmap`. `variantCounts` allows filtering by number of alleles or allele frequency. When passing `plotFreqHeatmap` a `matrix` instead of a `CrisprSet`, a header vector can also be supplied. If no header is supplied, the header is the column sums.

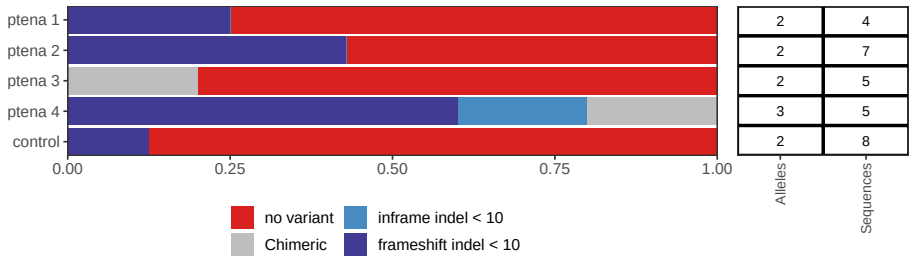
```
var_counts <- variantCounts(gol, top.n = 3)
# (additional modifications to var_counts can be added here)
plotFreqHeatmap(var_counts)
```

6.4 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.

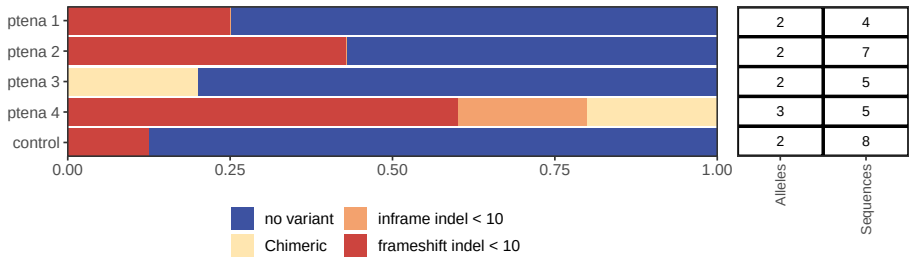
Here `barplotAlleleFreqs` is run with the default parameters:

```
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
## Warning in dispatchDots(.self$.getFilteredCigarTable, ...): dispatchDots may not
## work as expected with S4 functions
```



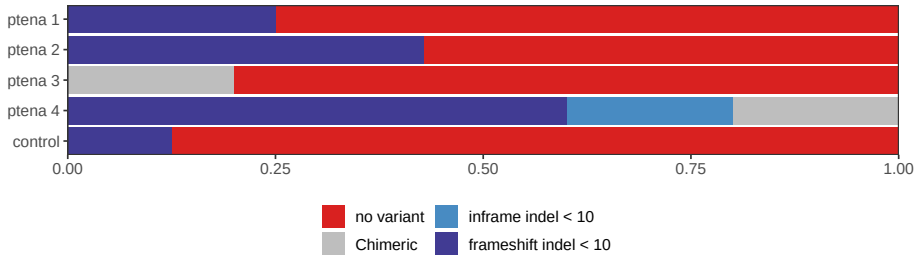
In this case `barplotAlleleFreqs` is run with the alternative palette.

```
barplotAlleleFreqs(crispr_set, txdb = txdb, palette = "bluered")
## Warning in dispatchDots(.self$.getFilteredCigarTable, ...): dispatchDots may not
## work as expected with S4 functions
```



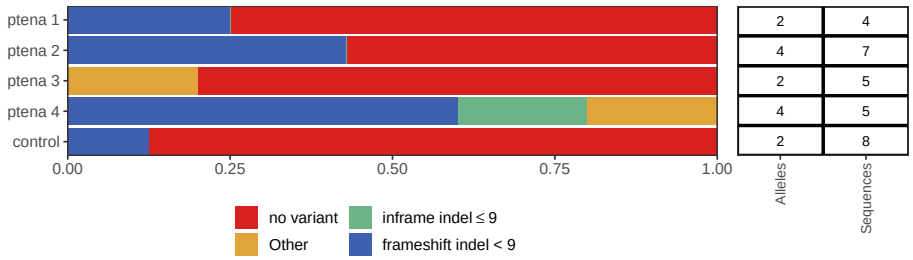
By default, a table of the number of sequences and alleles is plotted next to the barplot. This can be switched off. In this case, `barplotAlleleFreqs` will return an `ggplot` object, allowing further alteration of the appearance through the usual `ggplot2::theme` settings.

```
barplotAlleleFreqs(crispr_set, txdb = txdb, include.table = FALSE)
## Warning in dispatchDots(.self$.getFilteredCigarTable, ...): dispatchDots may not
## work as expected with S4 functions
```



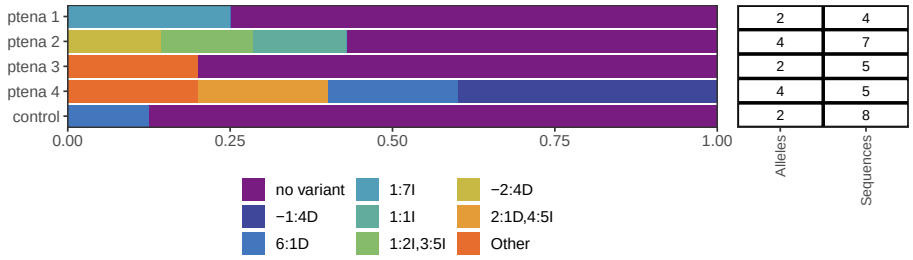
`barplotAlleleFreqs.CrisprSet` uses `VariantAnnotation::locateVariants` to look up the variant locations with respect to a transcript database. The default behaviour of `barplotAlleleFreqs.matrix` is to perform a naive classification of the variants as frameshift or non-frameshift by size. This approach 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.

```
rainbowPal10 <- c("#781C81","#3F479B",
                  "#4277BD","#529DB7",
                  "#62AC9B","#86BB6A",
                  "#C7B944","#E39C37",
                  "#E76D2E","#D92120")

barplotAlleleFreqs(var_counts, classify = FALSE, bar.colours = rainbowPal10)
```

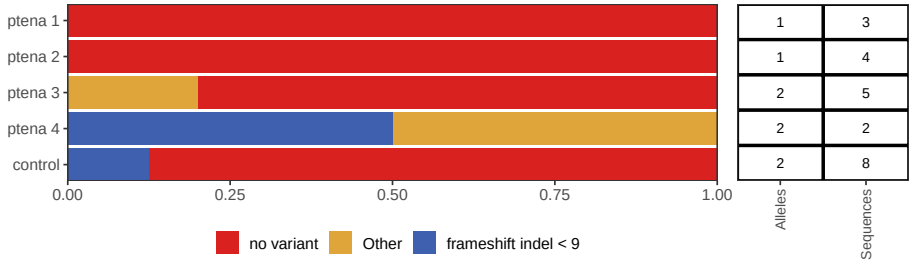


An arbitrary classification can also be used. `CrispRVariants` provides some utility functions to assist in classifying variants. Note that methods of the `CrisprSet` class are accessed with `crisprSet$function()` rather than `function(crisprSet)`.

Here are some examples of variant classification:

```
# Classify variants as insertion/deletion/mixed
byType <- crispr_set$classifyVariantsByType()
## Warning in dispatchDots(.self$.getFilteredCigarTable, ...): dispatchDots may not
## work as expected with S4 functions
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
```

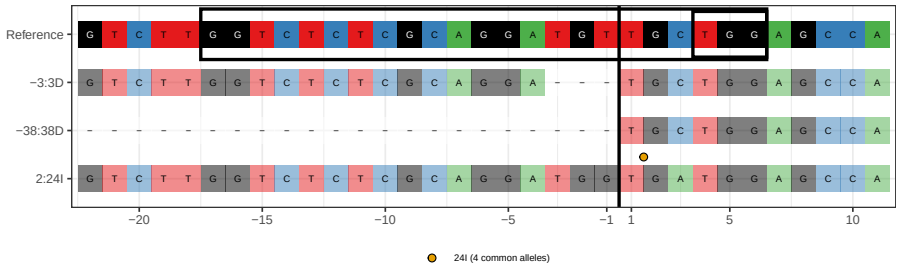
```
## Classifying variants
## Warning in dispatchDots(.self$.getFilteredCigarTable, ...): dispatchDots may not
## work as expected with S4 functions
byLoc
##   no variant        -1:4D         6:1D         1:7I         1:1I    1:2I,3:5I
## "no variant"     "coding"     "coding"     "coding"     "coding"     "coding"
##        -2:4D    2:1D,4:5I        Other
##     "coding"     "coding"      "Other"
# Coding variants can then be classified by setting a size cutoff
byLoc <- crispr_set$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])
```



6.4.1 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"))
## Warning: `legend.margin` must be specified using `margin()`. For the old
## behavior use legend.spacing
```



7 Using CrispRVariants plotting functions independently

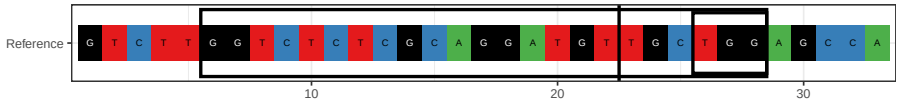
The CrispRVariants plotting functions are intended to be used within a typical CrispRVariants pipeline, where the correct arguments are extracted from a CrisprSet object. However, with some data formatting, it is also possible to use these functions with standard R objects.

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

7.1 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())
```



8 Note about handling of large deletions

`BWA` reports deletions above a threshold length as "chimeric" reads, with separate entries in the bam file for each mapped segment. By default, `CrispRVariants` only counts chimeric reads where one mapped endpoint is near the cut site. This setting was chosen as we observed long chimeric deletions in both on- and off-target CRISPR amplicon sequencing experiments in several independent data sets. The mapped endpoints were more likely to be in the vicinity of the cut site in on-target experiments. The off-target experiments did not have the other mutant alleles we expect to see if the long deletions are genuine CRISPR-induced variants. Some of the chimeric reads we observed appeared to be primer dimers. See the supplementary material of the CrispRVariants paper for more details:

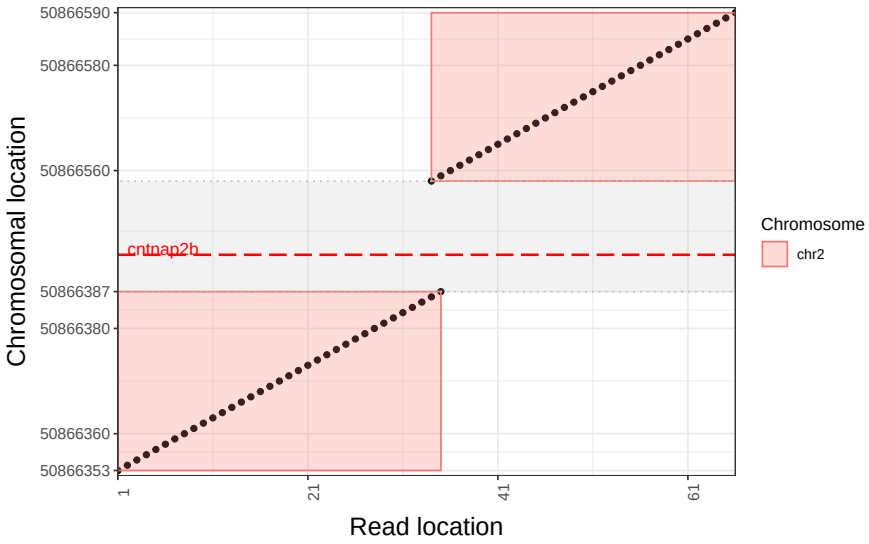
```
Lindsay H, Burger A, Biyong B, Felker A, Hess C, Zaugg J, Chiavacci E, Anders C, Jinek M,
Mosimann C and Robinson MD (2016). "CrispRVariants charts the mutation spectrum of genome
engineering experiments." Nature Biotechnology, 34, pp. 701-702. doi: 10.1038/nbt.3628.
```

The default chimera setting prioritises avoiding false positives such as primer dimers at the expense of potentially missing some genuine variants. This can be changed during initialisation by setting the `chimera.to.target` parameter to a large value.

```
library(Biostrings)
library(CrispRVariants)
library(rtracklayer)
```

```
# This is a small, manually generated data set with a variety of different mutations
bam_fname <- system.file("extdata", "cntnap2b_test_data_s.bam",
                         package = "CrispRVariants")
guide_fname <- system.file("extdata", "cntnap2b_test_data_guide.bed",
                           package = "CrispRVariants")
guide <- rtracklayer::import(guide_fname)
guide <- guide + 5
reference <- Biostrings::DNAString("TAGGCGAATGAAGTCGGGGTTGCCCAGGTTCTC")

cset <- readsToTarget(bam_fname, guide, reference = reference, verbose = FALSE,
                      name = "Default")
cset2 <- readsToTarget(bam_fname, guide, reference = reference, verbose = FALSE,
                       chimera.to.target = 100, name = "Including long dels")

default_var_counts <- variantCounts(cset)
print(default_var_counts)
##            cntnap2b_test_data_s.bam
## no variant                        6
## SNV:6G                            1
## 3:10I                             2
## 6:3D                              1
## 4:3D                              1
## -9:2D                             1
## -25:58D                           1
## 4:26D                             1
## 6:7D                              1
## 6:3I                              1
```

```
## 4:3I                              1
## 5:20I                             1
## 4:2D                              1
print(c("Total number of reads: ", colSums(default_var_counts)))
##                            cntnap2b_test_data_s.bam
## "Total number of reads: "                      "19"

# With chimera.to.target = 100, an additional read representing a large deletion is
# reported in the "Other" category.
var_counts_inc_long_dels <- variantCounts(cset2)
print(var_counts_inc_long_dels)
##            cntnap2b_test_data_s.bam
## no variant                        6
## SNV:6G                            1
## 3:10I                             2
## 6:3D                              1
## 4:3D                              1
## -9:2D                             1
## -25:58D                           1
## 4:26D                             1
## 6:7D                              1
## 6:3I                              1
## 4:3I                              1
## 5:20I                             1
## 4:2D                              1
## Other                             1
print(c("Total number of reads: ", colSums(var_counts_inc_long_dels)))
##                            cntnap2b_test_data_s.bam
## "Total number of reads: "                      "20"

# This alignment can be viewed using `plotChimeras`
ch <- getChimeras(cset2, sample = 1)
plotChimeras(ch, annotations = cset2$target)
```


50866590
50866580
50866560
cntnap2b
50866387
50866380
50866360
50866353
Chromosomal location
1
21
41
61
Read location
Chromosome
chr2