Annotation and Visualisation of sequencing data in Bioconductor

Mark Dunning

Last modified: May 20, 2015

Contents

1 Introduction

This practical uses data from a Drosophilla dataset where the *pasilla* gene has been knocked-out. The dataset has been used many times to demonstrate RNA-seq workflows and the data are available and discussed in various packages in Bioconductor (*pasilla*, *pasillaBamSubset*, *DEseq*). Here we show the analysis of this dataset using edgeR and we will start this practical from the table of top tags. **You do**

not need to type this code, it is for your reference only, or if you want to run the practical outside of the course.

```
library(edgeR)

datafile = system.file( "extdata/pasilla_gene_counts.tsv", package="pasilla" )
datafile

pasillaCountTable = read.table( datafile, header=TRUE, row.names=1 )

head(pasillaCountTable)

pasillaDesign = data.frame(
 row.names = colnames( pasillaCountTable ),
 condition = c( "untreated", "untreated", "untreated",
 "untreated", "treated", "treated", "treated" ),
 libType = c( "single-end", "single-end", "paired-end",
 "paired-end", "single-end", "paired-end", "paired-end" ) )


pasillaDesign

pairedSamples = pasillaDesign$libType == "paired-end"
countTable = pasillaCountTable[ , pairedSamples ]
condition = pasillaDesign$condition[ pairedSamples ]

y <- DGEList(counts=countTable,group=condition)
y <- calcNormFactors(y)
y <- estimateCommonDisp(y)
y <- estimateTagwiseDisp(y)
et <- exactTest(y)
topTags(et)


pasillaRes <- topTags(et,n=nrow(countTable))$table
```

The `pasillaRes` object has been saved in the `exampleData` folder.

Use Case: Load the pasilla analysis and familiarise yourself with the contents. Create a data frame of the 100 most significant results.

```
load("exampleData//pasillaRes.rda")
topHits <- pasillaRes[order(pasillaRes$PValue, decreasing = F)[1:100],]
head(topHits)
```

The `topHits` data frame provides us with a list of genes that are found to be *statistically significant*. However, it is a separate investigation to determine if the genes are *Biologically significant*. We will

annotate our list of differentially expressed genes using the online resource biomart.

2 Annotation using an online resource

The *biomaRt* package provides an interface to the biomart website; allowing the data to be accessed in a consistent manner and without having write complex SQL queries or understand the underlying schema. The databases available through BioMart can be obtained with the `listMarts` function

2.1 General BiomaRt Usage

Use Case: Get a list of all databases available through the *biomaRt* package and connect to the Drosophilla one.

```
library(biomaRt)
listMarts()

ensembl <- useMart("ensembl", dataset ="dmelanogaster_gene_ensembl")
attr <- listAttributes(ensembl)
head(attr, 10)
```

warning: biomaRt will automatically connect to the latest annotation version. If you are unsure what this version is, you can use the following code to access this information

```
datasets <- listDatasets(ensembl)
datasets[which(datasets[,1] == "dmelanogaster_gene_ensembl"),]
```

The function `getBM` is used to build-up queries in biomart. We must specify one or more attributes that we want to retreive. If no values or filters are specified, then all values will be returned.

Use Case: Retrieve the names of all genes for Drosophilla. How many genes are there?

```
allGenes <- getBM(attributes = c("ensembl_gene_id"), mart = ensembl)
dim(allGenes)
```

When restricting the search to particular query items (e.g. genes) we need to specify both filters and values.

Use Case: Retrieve the ensembl, transcript and peptide ID for the first 10 genes

```
filt <- listFilters(ensembl)
head(filt, 10)
getBM(attributes = attr[1:3, 1], filters = "ensembl_gene_id",
      values = allGenes[1:10,1], mart = ensembl)
```

For situations when we want to specify multiple filters, we have to supply the values in list form.

Use Case: Retrieve all the genes between 1100000 and 1250000 on chromsome X.

```
getBM(attributes = "ensembl_gene_id", filters = c("chromosome_name",
"start", "end"), values = list("X", 1100000, 1250000), mart = ensembl)
```

2.2 Annotating the RNA-seq results

We can use *biomaRt* to annotate the results of our RNA-seq analysis. Firstly, we have to recognise that the row names of the `topHits` data frame are Ensembl IDs. Then it is a matter of deciding what attributes we want to retrieve from the database.

Use Case: Annotate the top hits from the DEseq analysis with their genomic locations and external names and create a merged table containing both DESeq results and the annotations.

```
myInfo <- getBM(attributes = c("flybase_gene_id", "chromosome_name",
"band", "start_position", "end_position", "external_gene_name"),
filters = "ensembl_gene_id", values = rownames(topHits), mart = ensembl)

myInfo

annotatedHits <- merge(topHits, myInfo, by.x = 0, by.y = 1)

head(annotatedHits)
```

The resulting table, `annotatedHits`, can now be used to ease the interpretation of the RNA-seq analysis and can be shared with collaborators.

3 Annotation using pre-built packages

As an alternative to *biomaRt*, we can use the variety of pre-built databases that are available as Bioconductor packages. The main advantage of this approach is that we do not require an internet connection to perform the annotation, once the package is installed. A full list of annotation packages is available on the Bioconductor website.

3.1 Genome sequences

For instance, we can obtain pre-built and easily accessible packages to interrogate genome sequences. Such packages take advantage of the *Biostrings* infrastructure that you will have already seen, and employ a database scheme under-the-hood. Thus, making queries is extremely efficient.

Use Case: Load the package that provides the latest genome representation for Drosophila ('dm6')

```
library(BSgenome)
available.genomes()
library(BSgenome.Dmelanogaster.UCSC.dm6)
```

```
Dmelanogaster
```

We can retrieve the sequence for a particular chromosome by subsetting using the [[operator and providing the name of a chromosome.

Use Case: Retrieve the sequence for chromosome X by subsetting. Then create a plot to show the lengths of all chromosomes. *comment: You can use the length once you have retrieved each chromosome sequence*.

```
names(Dmelanogaster)
Dmelanogaster[["chrX"]]
chrLen <- sapply(names(Dmelanogaster), function(x) length(Dmelanogaster[[x]]))
barplot(chrLen[1:8], horiz=TRUE,las=2)
```

Biostrings genomes have also been designed to inter-operate with the GenomicRanges functions and classes. A useful consequence is that you can access the genomic sequence for a particular gene by translating the coordinates of the gene into a GRanges object.

Use Case: Use *biomaRt* to get the coordinates of the fly gene FBgn0040357 on Chromosome X. Translate this into a GRanges object and retrieve the genomic sequence for this gene.

```
geneInfo <- getBM(attributes = c("chromosome_name", "start_position", "end_position"),
filters = "ensembl_gene_id", values = "FBgn0040357", ensembl)

gr <- GRanges("chrX", IRanges(geneInfo$start_position, geneInfo$end_position))
seq <- getSeq(Dmelanogaster , gr)
seq
```

We can, of course, specify ranges for mutliple genes in the same GRanges object. They can also be manipulated, expanded, collapsed, etc using any of the operations we saw yesterday.

Use Case: Create a GRanges object to represent the top genes from the RNA-seq analysis. Find the genomic sequences of each of these genes, and also the sequences 50 bases upstream.

```
gr2 <- GRanges(paste0("chr",annotatedHits$chromosome_name),
               IRanges(annotatedHits$start_position, annotatedHits$end_position))
flankGr <- flank(gr2, 50)
myseqs <- getSeq(Dmelanogaster,gr2)
myseqs

upSeqs <- getSeq(Dmelanogaster,flankGr)
upSeqs
```

comment: In the next section, we will see how to retrieve GRanges for genes so that we do not have to create these object manually.

Various Biostrings operations could now be performed on these sequences. Some suggestions are given below.

```
substr(myseqs , 1, 10)
translate(myseqs)
source("scripts/gcFunction.R")
gcFunction(myseqs)
```

Biostrings also allows us to export sequence data so that we can process them in another tool.

Use Case: Export the upstream sequences a *fasta* file.

```
names(upSeqs) <- annotatedHits[, 1]
writeXStringSet(upSeqs , file = "topGenes.fa")
```

3.2 Organism-level Annotation

So-called organism level packages provide an alternative to *biomaRt* and permit annotation queries offline. Each package is built around a central identifier (e.g. Entrez ID) and meta data surrounding the annotation sources used can be displayed once such an organism package has been installed and loaded.

The terminology of `columns` and `keys` is equivalent to `attributes` and `filters` used by *biomaRt*. The valid columns and keys can be interrogated using `columns` and `keytypes`.

Use Case: Load the Drosophilla annotation package and display the metadata describing how the package was created. Find out what information can be retrieved from the package and what types of key can be used.

comment: For covenience, we often assign a shorter name to the annotation package.

```
library(org.Dm.eg.db)
dm <- org.Dm.eg.db
columns(dm)
keytypes(dm)
```

The `select` function is used to make the query. We have to supply a set of valid keys and columns.

Use Case: Annotate your RNA-seq results using the organism package. Choose any columns that you feel might be interesting.

```
myInfo2 <- select(dm, keytype = "FLYBASE",
      columns =c("ENSEMBL", "MAP","CHR","GENENAME","SYMBOL") ,keys=rownames(topHits))

myInfo2
```

In the analysis of the pasilla dataset, we allocated reads to genes. However, in this section we will explore the ways of counting reads that align to other genomic regions of interest. In this case we will use *exons* and have to re-visit the orginal aligned reads. An example bam file is provided in the *pasillaBamSubset* package, but only reads on chromosome 4 are provided.

In order to use the example bam file we will have to get the IDs of genes that occur on chromosome 4. We illustrate this using the *org.Dm.eg.db* organism package.

Use Case: Use the appropriate organism package to retrieve the ensembl IDs of all genes located on chromosome 4 of Drosophilla.

```
chr4Genes <- select(dm, columns = "ENSEMBL", keytype = "CHR", keys = "4")
head(chr4Genes)
chr4ID <- chr4Genes[, 2]
```

warning: If you get the error message:
`unused arguments (columns = "ENSEMBL", keytype = "CHR", keys = "4")`
when trying to run the `select`, it is because another R package has defined another function called `select` and over-written the function from AnnotationDBI that we intended to use. If this occurs the statement
`select <- AnnotationDbi::select`
should allow you to run the line of code.

3.3 Transcript packages

Bioconductor provide a number of pre-built packages that have a database of all transcript information for a particular organism. The package we are going to use is *TxDb.Dmelanogaster.UCSC.dm3.ensGene*. A full list of available transcript packages is available on the Bioconductor website. Look for the packages that start with `TxDb.`.

comment: At present, there is no transcript database package relating to the dm6 genome, so we will use dm3 instead.

As its name suggests, *TxDb.Dmelanogaster.UCSC.dm3.ensGene* was built from the UCSC tables for the dm3 genome and uses Ensembl genes as an identifier. Convenient functions exist that can return the gene structure as a *GRanges* object. One such function is `exonsBy` which returns a list of `GRanges` objects, each item in the list pertaining to a particular gene. Thus, the output of `exonsBy` can be subset in the usual manner (`'[['`).

comment: transcripts and cds regions can be selected in a similar fashion. See the help pages of `intronsBy` and `cdsBy`

Use Case: Get all the exons for all genes on chromosome 4

```
library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
tx <- TxDb.Dmelanogaster.UCSC.dm3.ensGene

exo <- exonsBy(tx, "gene")
exo

length(exo)
exo4 <- exo[names(exo) %in% chr4ID]
```

```
length(exo4)
```

comment: If you cannot find a TranscriptDb package that provides the annotation you need, you can use the makeTxDbPackage function to create one from the UCSC browser, biomaRt or another source. See the help page for more details.

Use Case: Read the example untreated3_chr4 bam file from the *pasillaBamSubset* package. Verify that only chromosome 4 reads are included in the object

comment: untreated3_chr4() is a convenient function to give the absolute location of the example file in the pasillaBamSubset package.

```
library(pasillaBamSubset)
library(GenomicAlignments)
bam <- readGAlignments(untreated3_chr4(), use.name = TRUE)
table(seqnames(bam))
```

We are now in a position to begin overlapping reads with exons. You should have already seen the findOverlaps function that will report the amount of ovelap between two sets of ranges. We will start simple by just doing an overlap of a single gene to make sure that we understand the output

Use Case: Use the findOverlaps function to count the number of reads covering each exon of FBgn0002521.

```
olaps <- findOverlaps(exo4[["FBgn0002521"]], bam)
olaps
bam[subjectHits(olaps)]
```

summarizeOverlaps extends findOverlaps by providing options to resolve reads that overlap multiple features. Each read is counted a maximum of once. Different modes of counting are available. See the help page for summarizeOverlaps for more details.

Use Case: Repeat the counting for all genes on chromosome 4

```
olapGenes <- summarizeOverlaps(exo4, bam)
countGenes <- assays(olapGenes)$counts
countGenes
```

The count vector has entries for each gene on chromosome 4. If we wanted counts for all exons, then we would have to supply a slightly different argument to the function, by collapsing the list structure of the exons object.

Use Case: Obtain per-exon counts for each gene

```
 exonRanges <- unlist(exo4)
olapExon <- summarizeOverlaps(exonRanges , bam)
countExons <- assays(olapExon)$counts
countExons
```

The summarizeOverlaps function can also take the location of one or more bam files as an argument.

This means that the processing of reads and counting is handled in one step. However, we have to use the special `BamFileList` constructor when dealing with multiple files.

Use Case: Obtain counts of chromosome 4 genes for the bam files included with the *pasillaSubset* package.

```
fls <- c(untreated3_chr4(), untreated1_chr4())
names(fls) <- basename(fls)
multiOlap <- summarizeOverlaps(exo4, BamFileList(fls))
head(assays(multiOlap)$counts)
```

3.4 Importing and annotating using custom tracks

Genome browser or other tracks are often distributed as files in the formats gff, bed, wig etc. The package *rtracklayer* provides infrastrucutre to import files in these formats and create a *GRanges* representation.

Use Case: Read the gff file supplied with the pasilla package and use these intervals to produce a table of overlaps.

```
library(rtracklayer)
gffFile <- "Dmel.BDGP5.25.62.DEXSeq.chr.gff"
gff <- paste(system.file("extdata", package = "pasilla"), gffFile,sep="/")
read.table(gff, nrows = 10, sep = "\t")
gffRange <- import(gff)
gffRange
gffOverlap <- summarizeOverlaps(gffRange , bam)
```

4 Visualisation

4.1 Exporting tracks

It is also possible to save the results of a Bioconductor analysis in a browser to enable interactive analysis and integration with other data types, or sharing with collaborators. We shall use the bed format for illustration.

Use Case: Select all significant genes from the pasilla analysis (lets use a p-value of 0.1) and annotate their genome location. Create a *GRanges* representation of the locations.

```
sigResults <- pasillaRes[which(pasillaRes$PValue< 0.1), ]

myInfo <- getBM(attributes = c("flybase_gene_id", "chromosome_name",
"band", "start_position", "end_position", "external_gene_name"),
filters = "flybase_gene_id", values = rownames(sigResults), mart = ensembl)
```

```
myInfo <- merge(sigResults, myInfo, by.x = 0, by.y = 1)


sigRanges <- GRanges(paste("chr", myInfo$chromosome_name, sep = ""),
IRanges(start = myInfo$start_position , end = myInfo$end_position ,
names = myInfo[, 1]))
```

Rather than just representing the genomic locations, the .bed format is also able to colour each range according to some property of the analysis (e.g. direction and magnitude of change) to help highlight particular regions of interest. A score can also be displayed when a particular region is clicked-on. A useful propery of `GenomicRanges` is that we can attach metadata to each range using the `mcols` function.

Use Case: Store the adjusted p-values and log fold-change in the ranges object

```
mcols(sigRanges)$padj = myInfo$PValue
mcols(sigRanges)$logfc = myInfo$logFC
sigRanges
```

Use Case: Create a score from the p-values and colour scheme based on the fold-change. For convenience, restrict the fold changes to be within the region -3 to 3.

```
Score <- -log10(sigRanges$padj)
rbPal <-colorRampPalette(c("red", "blue"))

logfc <-  pmax(sigRanges$logfc, -3)
logfc <-  pmin(logfc , 3)

Col <- rbPal(10)[as.numeric(cut(logfc, breaks = 10))]
```

The colours and score can be saved in the *GRanges* object and `score` and `itemRgb` columns respectively and will be used to construct the track when the export is called.

Use Case: Export the signifcant results from the DE analysis as a *.bed* track. You can load the resulting file in IGV, if you wish.

```
mcols(sigRanges)$score <- Score

mcols(sigRanges)$itemRgb <- Col

export(sigRanges , con = "topHits.bed")
```

The export of other formats such as *gff*, *bedGraph*, *wig* and *bigwig* is also possible with *rtracklayer*. We can also call the UCSC browser directly to display the tracks. See the *rtracklayer* vignette for details.

4.2 ggbio

We will now take a brief look at one of the visualisation packages in Bioconductor that takes advantages of the GenomicRanges and GenomicFeatures object-types. The documentation for *ggbio* is very extensive and contains lots of examples.

`http://www.tengfei.name/ggbio/docs/`

Extra background reading can be found on the ggplot2 website; the package on which the principles of ggbio is based.
`http://ggplot2.org/`

The *Gviz* package is another Bioconductor package that specialising in genomic visualisations, but we will not explore this package in the course.

A useful function within *ggbio* is `autoplot`, which will construct an appropriate plot based on the object-type of the input. For example, if we pass a GRanges object to the function it will plot the genomic locations on a linear scale and label chromosome and positional information on the plot. The style of the plot can be changed by the layout argument.

Use Case: Plot the locations of the significant hits from the DE analysis. Try the default, karyogram and circle layouts

```
library(ggbio)
autoplot(sigRanges)
autoplot(sigRanges[seqnames(sigRanges) == "chrX"])
autoplot(sigRanges , layout = "karyogram")
```

The *Manhattan* plot is a common way of visualising genome-wide results, and this is implemented as the `plotGrandLinear` function. We have to supply a value to display on the y-axis. This is done by using the `aes` function, which is inherited from *ggplot2*.

Use Case: Plot the genomic locations of the significant genes on a linear scale. Modify the colour scheme to show whether each gene is up- or down-regulated

```
plotGrandLinear(sigRanges , aes(y = score))
mcols(sigRanges)$Up <- logfc > 0
plotGrandLinear(sigRanges, aes(y = score, col = Up))
```

ggbio is also able to plot the structure of genes according to a particular model represented by a `GenomicFeatures` object.

Use Case: Find the gene which has the most exon counts in the pasilla bam. Plot the struture of this gene.

```
myGene <- names(exonRanges)[which.max(countExons)]
autoplot(tx, which = exo4[[myGene]])
```

We can even plot the location of sequencing reads if they have been imported using `readGAlignments`

function (or similar).

Use Case: Plot the number of reads in this region surrounding the gene.

```
myreg <- flank(reduce(exo4[[myGene]]), 500, both = T)
bamSubset <- bam[bam %over% myreg]
autoplot(bamSubset)
```

Use Case: Repeat the plot, but with a smoothed coverage

```
autoplot(bamSubset , stat = "coverage")
```

The plots produced by *ggbio* (and indeed *ggplot2* on which the package is based) are not in standard R graphics format, so techniques like `par(mfrow)` for arranging plots on the same page are not applicable. However, *ggbio* allows the plots to be saved as objects, which can later be arranged by specialist functions such as `tracks`. This function automatically aligns the x-axis of each plot to be on the same scale.

Use Case: Make a combined plot of the aligned reads and transcripts for the selected gene.

```
p1 <- autoplot(tx, which = myreg)
p2 <- autoplot(bamSubset, stat = "coverage")
tracks(p1, p2)
```

5 Annotating a set of genomic variants

If given a set of genomic variants (i.e. single-nucleotide variants or indels), it is a common task to see where each variant is located with respect to a gene and what effect, if any, they have on coding.

Most variant-calling tools produce output in *vcf* format, which emerged as a standard file format from the 1000 genomes project. The *VariantAnnotation* package provides infrastrucure for reading and manipulating variant calls in this format.

As an example, we will use variants that were called on chromosome 22 using 'samtools' for three Hapmap individuals. The resulting vcf file can be found in the `exampleData` folder.

Use Case: Read the example `vcf` file using *VariantAnnotation*

```
library(VariantAnnotation)
myvcf <- readVcf("exampleData//chr22.flt.vcf","hg19")
myvcf
```

comment: We have to specify a 'genome' version when we read the file in. As yet this doesn't have any impact on the analysis, so we can really specify any value we like

5.1 Exploring the vcf data

The structure of the `myvcf` object is complex, but you should be able to determine the number of SNPs and samples. We can extract data using a series of extractor functions.

'info' contains data that are specific to each SNP and can be retrived using the `info` function. The fields to be found here are often useful for quality control and filtering. For instance, we would expect bases with more reads covering them to yield more-confident SNPs. How to define such filters is beyond the scope of what we cover here, and such filters may be application or tool-specific. However, as a basic rule-of-thumb we want to exclude calls at low depth.

Use Case: Get descriptions for the info values and plot histograms of the depth and mapping quality of each SNP. Create a new object to contain only variants that are covered at depth more than 10. How many variants remain?

```
info(header(myvcf))
hist(info(myvcf)[["DP"]])
hist(info(myvcf)[["MQ"]])
lowDepthVars <- which(info(myvcf)[["DP"]] < 10)

myvcf <- myvcf[-lowDepthVars]
myvcf
```

Information that is sample-specific is retrieved using the `geno` function. Here we get a data frame returned, with one row for each variant, and one column for each sample.

Use Case: What per-sample information is available? How many calls are shared between the three samples?

```
geno(header(myvcf))

gt <- geno(myvcf)[["GT"]]
sum(gt[,1] == gt[,2])
sum(gt[,2] == gt[,3])
sum(gt[,1] == gt[,3])
```

We are still left with a long list of variants. In some studies, we might want to refine our list so that it contains only 'novel' findings that we can follow-up. One resource useful for this task is the various versions of *dbSNP*

5.2 Checking for novelty

Use Case: Load the *SNPlocs.Hsapiens.dbSNP.20101109* package and retrieve the locations of common snps on chromosome 22. Find which of variants overlap with these SNPs, and remove them.

```
library(SNPlocs.Hsapiens.dbSNP.20101109)
 rd <- rowData(myvcf)
```

```
ch22snps <- getSNPlocs("ch22",as.GRanges = TRUE)
ch22snps <- renameSeqlevels(ch22snps, c("ch22" = "22"))

olapPos = findOverlaps(rowData(myvcf),ch22snps)

knownStatus <- rep("novel", length(rd))
knownStatus[queryHits(olapPos)] <- "known"
table(knownStatus)

novelHits <- myvcf[knownStatus == "novel"]
```

comment: Notice that we have to rename the chromosomes of the dbSNP locations our variants so that they are consistent with our set of variants.

5.3 Annotating to genes

Since the variant locations can be translated to GenomicRanges using the `rowData` function, it is relatively trivial to determine which ones are locating within genes, and thus have the potential to disrupt protein-coding. We have already seen how to use transcript database packages to retrieve coding sequences. However, since it is such a common use-case, a useful `codingVariants` function has been provided in *VariantAnnotation* to assist in the task

Use Case: Use `locateVariants` to determine which of the novel variants are within coding regions. Again, make sure that the chromosome names are consistent between the two sets of ranges.

```
 library(TxDb.Hsapiens.UCSC.hg19.knownGene)
txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene

novelHits <- renameSeqlevels(novelHits, c("22" = "chr22"))
rd <- rowData(novelHits)


 loc <- locateVariants(rd, txdb, CodingVariants())
 head(loc, 3)
```

comment: Other types of variant overlap are supported by `locateVariants`. For example, we can see which of our variants fall within introns, splice sites or promoters. As usual, we can read about them on the help page for the function.

5.4 Predicting consequence

Given the variant locations and genome sequence, we can quantify what effect each will have on coding.

Use Case: Determine the amino acids changes that arise from the variants we have detected.

```
library(BSgenome.Hsapiens.UCSC.hg19)
 coding <- predictCoding(novelHits, txdb, seqSource=Hsapiens)

coding[1:10]
table(mcols(coding)$CONSEQUENCE)
```

comment: A useful column here is QueryID, which can be used to select rows (variants) from the vcf file

The structure used to store variants is recognised by *ggbio*, which allows variant calls to be displayed alongside transcript and other genomic data. Here is example code to plot the location of our coding variants with respect to the relevant gene. Firstly, we create a set of exon locations from the human genome transcript database and a reduced vcf file containing just the coding variants. It is then a matter of creating a `GRanges` object corresponding to the gene of interest. We can supply these locations to `autoplot`, and *ggbio* will work out how to do the plot.

```
hgExons <- exonsBy(txdb, "gene")

codingvars <- novelHits[coding$QUERYID]

affectedGenes <- na.omit(unique(coding$GENEID))


gr <- hgExons[[affectedGenes[1]]]

t1 <- autoplot(txdb, which = gr)
t2<- autoplot(codingvars, which = gr,geom="rect")
tracks(t1, t2)
```

If we wish to view the variants in an external tool, we can export the locations as a bed file using functionality from the *rtracklayer* package. *VariantAnnotation* is also able to write a vcf file.

Use Case:

Export the locations of coding variants as a *.bed* file and export the vcf data relating to these variants as a new vcf file.

```
export(rowData(codingvars),con="codingVariants.bed")
writeVcf(codingvars,filename = "codingVariants.vcf")
```