How-to guide to Rgb

Sylvain Mareschal

January 14, 2016

About

This document describes how to use the R Genome Browser, from general concepts to practical user cases. Questions and feedback may be sent to maressyl@gmail.com, news and updates will be made available on the package web page.

Reference

Mareschal S, Dubois S, Lecroq T, Jardin F.
Rgb: a scriptable genome browser for R.
Bioinformatics. 2014 Aug 1;30(15):2204-5.
doi: 10.1093/bioinformatics/btu185

Acknowledgements

Many thanks to Sydney Dubois for her consciencious copy-editing, to Christian Bastard for his implication in Rgb development, and to Bioinformatics' peer reviewers for their useful suggestions on the current manual.

Contents

1 Quick start

1.1 Purpose

Rgb, the "R genome browser", aims to provide native **genome browsing** solutions for the R language. Genomes are usually organized in independent chromosomes, each of them consisting in a long sequence of nucleotides in which strips of biological interest (genes, introns, exons ...) can be localized using integer coordinates. Working with this specific paradigm requires dedicated tools, usually called "Genome browsers".

As a member of this family, Rgb provides classes to store such genomically located data, and methods to perform common tasks on them (subset, edit, draw ...). These classes transparently integrate a generic drawing system able to produce publication-grade representations of a locus of interest, which can be controlled either by direct R commands or by a Graphical User Interface that requires no specific R knowledge to be operated.

In most genome browsers (including Rgb), data is handled as **tracks**, which consist of collections of genomic features of the same kind (genes, transcripts ...) and from the same organism. Rgb provides R classes to handle them, and a specific file format (.rdt) to store them from one session to another. Numerous parameters can be customised in such tracks to control the way the data will be represented, and great care has been taken to allow R developers to implement entirely new representation systems in Rgb.

1.2 Track building

Full documentation on track object building can be found in the R manuals:

```
>   help("Annotation")
```

1.2.1 Usual annotation tracks

A few functions are provided to build annotation tracks from data available in remote repositories (NCBI, UCSC ...). First download the appropriate file, call the parsing function to produce the object and export it as a .rdt file:

```
>   download.file(
+     "http://hgdownload.cse.ucsc.edu/goldenPath/hg19/database/cytoBand.txt.gz",
+     destfile = "cytoBand.txt.gz"
+   )
>   track <- track.bands.UCSC(
+     file = "cytoBand.txt.gz",
+     .organism = "Human",
+     .assembly = "hg19"
+   )
>   saveRDT(track, file="cytoBands.rdt")
```

The full list of handled datasets and links to download the corresponding files are described in the Annotation help page:

```
>   help("Annotation")
```

1.2.2 Custom annotation tracks

Annotation data can be downloaded from the UCSC Table Browser in the "Gene Transfert Format" (GTF), parsed, and saved as a RDT track. The following example is derived from the user case 3.2.4, in which we were interested in the COSMIC mutations in the ATM gene region:

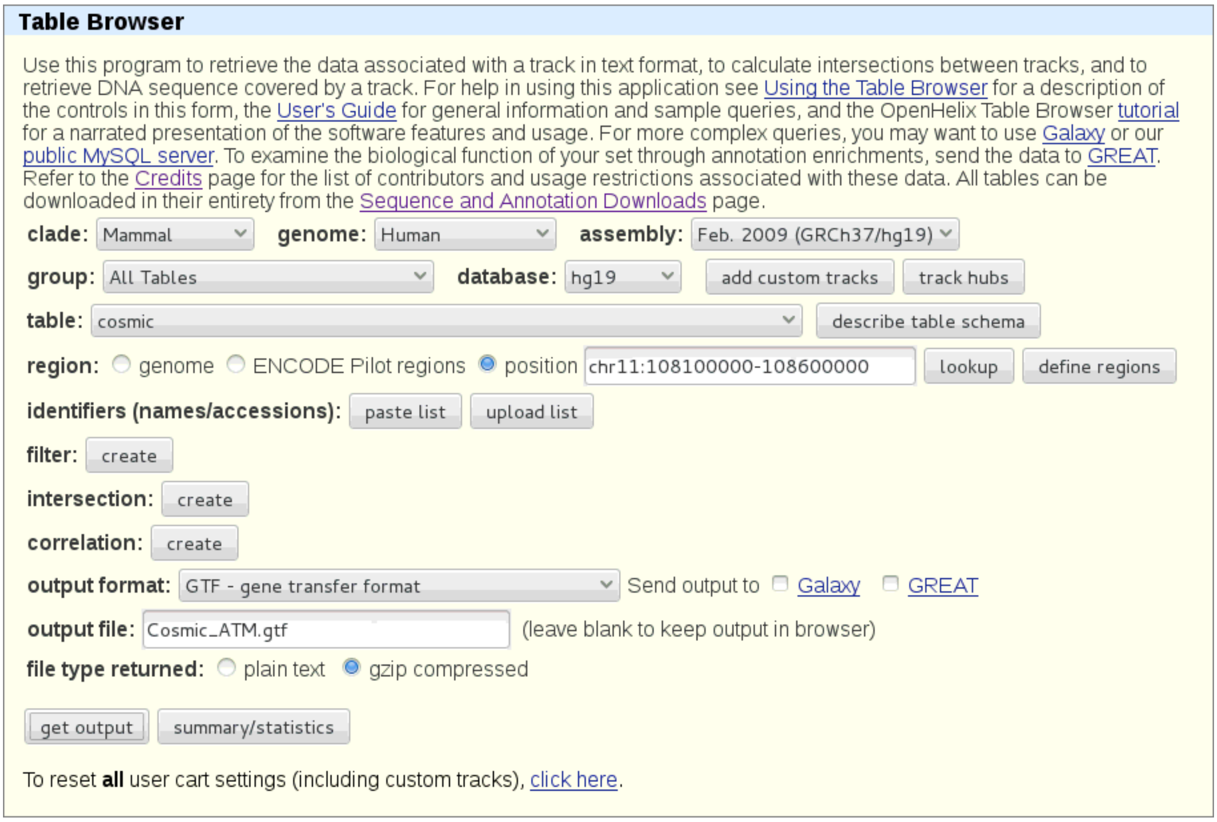


Table Browser

Use this program to retrieve the data associated with a track in text format, to calculate intersections between tracks, and to retrieve DNA sequence covered by a track. For help in using this application see Using the Table Browser for a description of the controls in this form, the User's Guide for general information and sample queries, and the OpenHelix Table Browser tutorial for a narrated presentation of the software features and usage. For more complex queries, you may want to use Galaxy or our public MySQL server. To examine the biological function of your set through annotation enrichments, send the data to GREAT. Refer to the Credits page for the list of contributors and usage restrictions associated with these data. All tables can be downloaded in their entirety from the Sequence and Annotation Downloads page.

clade: Mammal genome: Human assembly: Feb. 2009 (GRCh37/hg19)

group: All Tables database: hg19

table: cosmic

region: ○ genome ○ ENCODE Pilot regions ● position chr11:108100000-108600000

identifiers (names/accessions):

filter:

intersection:

correlation:

output format: GTF - gene transfer format Send output to □ Galaxy □ GREAT

output file: Cosmic_ATM.gtf (leave blank to keep output in browser)

file type returned: ○ plain text ● gzip compressed

To reset **all** user cart settings (including custom tracks), click here.

The downloaded GTF file can be parsed using the `track.table.GTF` function. Notice this function handles gzipped GTF files, so it is preferable to download them in such a format in order to minimize download time and disk usage. To make the example run, the following commands rely on the same file which is included in the Rgb package for testing purposes.

```
>   file <- system.file("extdata/Cosmic_ATM.gtf.gz", package="Rgb")
>   tt <- track.table.GTF(file)
>   saveRDT(tt, file="custom.rdt")
```

As the tables supplied by the Table Browser may need some reshaping, such files can also by parsed into data.frames by the `read.gtf` function, updated manually according to your needs, and exported as RDT tracks (see 1.2.3 for the generic method, and 3.2.4 for the more specific user case).

1.2.3 Custom R data tracks

Custom datasets can easily be converted from `data.frame` to `track.table` objects, then exported as .rdt files for Rgb. The table can hold as many columns as you want, as long as the few required columns are provided: chrom (factor), strand ("+" or "-"), start (integer), end (integer) and name (character). For more information on the `track.table` class and constructor, see 2.4.

```
>   data(hsFeatures)
>   class(hsGenes)

[1] "data.frame"

>   head(hsGenes)

     name chrom start   end strand
1  WASH7P     1 14362 29370      -
2 FAM110C     2 38814 46588      -
3  ZNF595     4 53227 88099      +
4 OR4G11P     1 63016 63885      +
5 DEFB125    20 68351 77296      +
6   TUBB8    10 92828 95178      -

>   tt <- track.table(hsGenes)
>   saveRDT(tt, file="custom.rdt")
```

1.2.4 Next Generation Sequencing tracks

BAM files are currently supported as an experimental feature, by the `track.bam` function which relies on a de novo R implementation of BAM querying based on SAM 1.4 specification sheet. Due to the size of the datasets handled in such formats, the track produced does not contain the dataset but rather links it to R.

```
>   track <- track.bam(
+     bamPath = system.file("extdata/ATM.bam", package="Rgb"),
+     .organism = "Human",
+     .assembly = "hg19"
+   )
>   saveRDS(track, file="sequencing.rds")
```

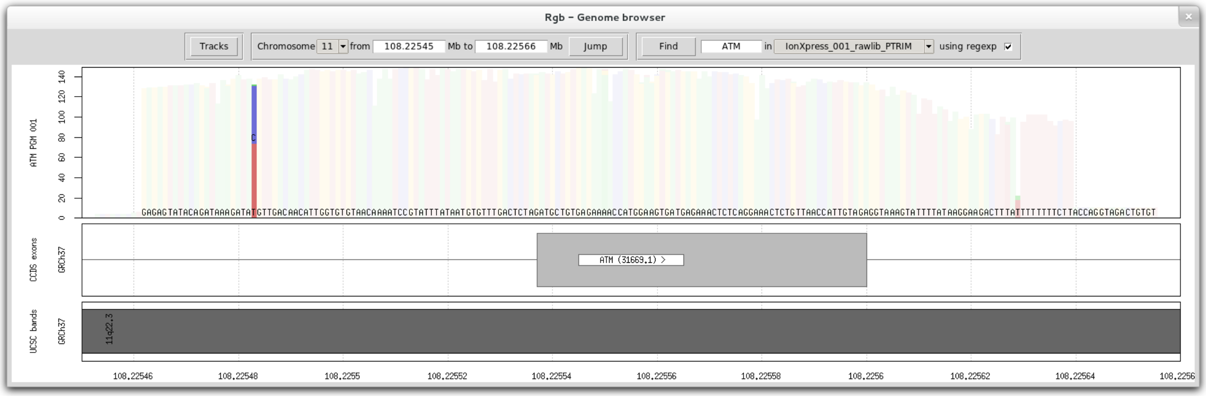
Notice the file format for such a track is slightly different: we use the RDS file format provided by R here, which is also plainly supported by Rgb.

1.3 Genome browsing

1.3.1 Interactive genome browsing

The genome browser can be summoned in an interactive session by a single call to `tk.browse`. It requires the `tcltk` and `tkrplot` packages (both available at the CRAN for most operating systems).

```
>   tk.browse()
```



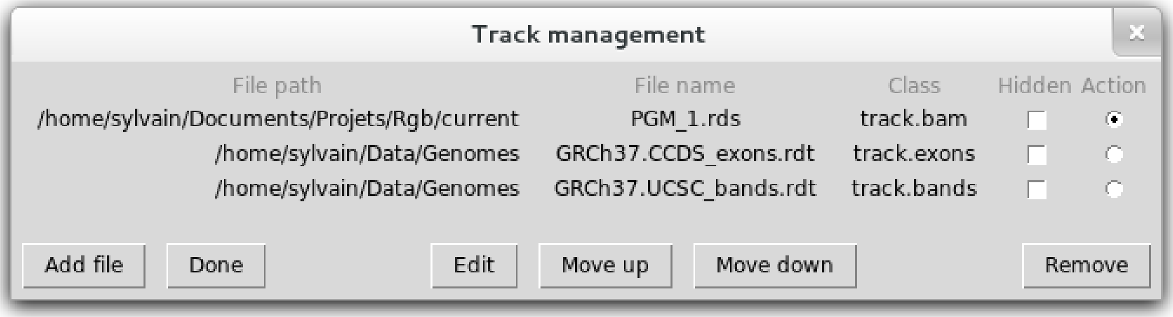
Rgb – Genome browser

Chromosome 11 from 108.22545 Mb to 108.22566 Mb

Find ATM in IonXpress_001_rawlib_PTRIM using regexp ☑

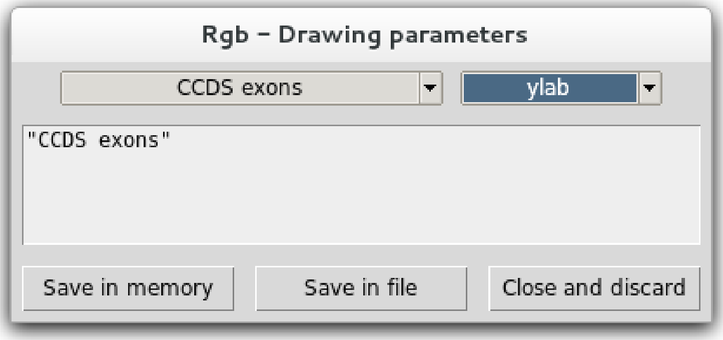
Tracks
Jump
Find
ATM PGM 001
0
20
40
60
80
100
120
140
C
GAGAGTATACAGATAAAGATATGTTGACAACATTGGTGTGTAACAAAATCCGTATTTATAATGTGTTTGACTCTAGATGCTGTGAGAAAACCATGGAAGTGATGAGAAACTCTCAGGAAACTCTGTTAACCATTGTAGAGGTAAAGTATTTTATAAGGAAGACTTTATTTTTTTTCTTACCAGGTAGACTGTGT
CCDS exons
GRCh37
ATM (31669.1) >
UCSC bands
GRCh37
11q22.3
108.22546
108.22548
108.2255
108.22552
108.22554
108.22556
108.22558
108.2256
108.22562
108.22564
108.2256

Track files can be selected by clicking the ”Tracks” button in the top left hand corner, via the track management interface presented below. Tracks can be hidden by ticking the corresponding ”Hidden” box, edited or reordered. Drawing parameters of all loaded tracks can be edited, replacing their values by valid R code (most parameters are single values). For more information on the drawing parameters, please refer to the appropriate drawing function manual (summoned by the ”drawFun” parameter).



Track management

File path	File name	Class	Hidden	Action
/home/sylvain/Documents/Projets/Rgb/current	PGM_1.rds	track.bam	☐	●
/home/sylvain/Data/Genomes	GRCh37.CCDS_exons.rdt	track.exons	☐	○
/home/sylvain/Data/Genomes	GRCh37.UCSC_bands.rdt	track.bands	☐	○



Rgb – Drawing parameters

CCDS exons ▾ ylab ▾

```
"CCDS exons"
```

The plotted window can be moved by entering genomic coordinates, or via keyboard shortcuts: use the **left and right arrows** to move along the chromosome and the **up and down arrows** to zoom in and out (or the mouse wheel if available). Zooming in can also be achieved with the mouse, **holding a left click** from the desired left border to the desired right border. Finally the **”R” key** can prove useful to force a plot refresh.

Centering the view on track features searched by ”name” can also be achieved by using the top right hand fields, selecting the track to be searched and entering the searched pattern aside. The default behavior is to return only exact matches, but regular expressions can be used by checking the ”using regexp” box. If multiple hits are found, a message window pops up and reminds the user which match is currently viewed, switching each time the ”Find” button is hit.

1.3.2 Scripted genome browsing

Genome browser representations can also be produced by scripts, calling the **browsePlot** on which **tk.browse** relies for its renderings. This function needs the track list to be passed as a **drawable.list** object:

```
>   data(hsFeatures)
>   dl <- drawable.list()
>   dl$add(file=NA, track=hsBands)
>   dl$add(file=NA, track=track.table(hsGenes))
>   browsePlot(dl, chrom="1", start=0, end=10e6)
```

Drawable lists provide a common interface for scripts and GUI, allowing users to edit their content interactively even while scripting. Notice tracks should be passed with their paths to allow updates to be stored, in the examples presented here updates can only be stored in memory:

```
>   dl$fix.files()
>   dl$fix.param()
```

As it relies on R plot, it can be redirected to a file (PNG, PDF ...):

```
>   pdf("Rgb_tests.pdf")
>   browsePlot(dl, chrom="1", start=0, end=10e6)
>   browsePlot(dl, chrom="8", start=50e6, end=60e6)
>   dev.off()
```

```
pdf
  2
```

1.4 Working with genomic data

The **track.table** class developed for Rgb can prove very useful when computationally intensive operations are to be performed on genomic data.

1.4.1 slice: subset by coordinates

Aside from the **draw** method extensively used by the genome browser, a fast and memory efficient **slice** method is proposed for the most common task in genomic data manipulation: extracting rows in a given genomic window.

```
>   data(hsFeatures)
>   genes <- track.table(hsGenes)
>   genes$slice(chrom="12", start=45e6, end=48e6)
```

```
       name chrom    start      end strand
1      DBX2    12 45408539 45444882      -
2  RACGAP1P    12 45456401 45459194      -
3 RPL13AP21    12 46397809 46398468      -
4   OR7A19P    12 46986357 46987089      +
5   SLC38A4    12 47158544 47219780      -
6    AMIGO2    12 47469490 47473734      -
```

```
>   system.time(
+     for(i in 1:10000) genes$slice("12", 25e6, 118e6)
+   )
```

```
   user  system elapsed
  0.889   0.003   0.896
```

1.4.2 cross: intersect two tables

The **cross** method, which makes direct use of the **slice** one presented above, can also prove particularly useful to annotate genomic tables or count overlaps between tables:

```
>   data(hsFeatures)
>   print(hsBands)
```

```
  "track.table" reference class object
  organism   : Human
  assembly   : GRCh37

  Extends "drawable"
  name       : UCSC bands

  Extends "refTable"

         name chrom strand    start      end  stain
1    1p36.33     1      +        1  2300000   gneg
2    1p36.32     1      +  2300000  5400000 gpos25
3    1p36.31     1      +  5400000  7200000   gneg
...      ...   ...    ...      ...      ...    ...
860 Yq11.223     Y      + 22100000 26200000 gpos50
861  Yq11.23     Y      + 26200000 28800000   gneg
862     Yq12     Y      + 28800000 59373566   gvar
```

```
>   genes <- track.table(hsGenes)
>   hsBands$cross(genes, type="count")[1:5]
```

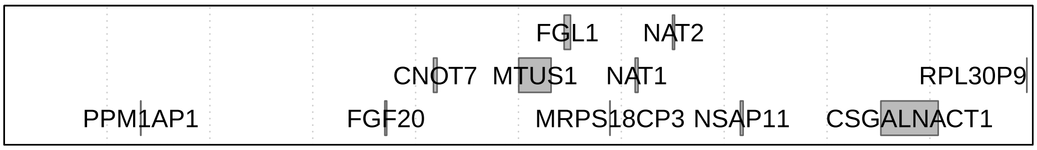
```
[1] 17  4  8  6 13

>   hsBands$cross(genes, colname="genes", type="name", maxElements=5)
>   print(hsBands)

  "track.table" reference class object
  organism   : Human
  assembly   : GRCh37

  Extends "drawable"
  name       : UCSC bands

  Extends "refTable"

        name chrom strand    start      end  stain
1    1p36.33     1      +        1  2300000   gneg
2    1p36.32     1      +  2300000  5400000 gpos25
3    1p36.31     1      +  5400000  7200000   gneg
...      ...   ...    ...      ...      ...    ...
860 Yq11.223     Y      + 22100000 26200000 gpos50
861  Yq11.23     Y      + 26200000 28800000   gneg
862     Yq12     Y      + 28800000 59373566   gvar

                                genes
1                       (17 elements)
2   WRAP73, TP73-AS1, C1orf174, AJAP1
3                        (8 elements)
...                               ...
860                     (24 elements)
861                     (17 elements)
862                    TCEB1P24, IL9R
```

For further details on the **track.table** class, please refer to the next chapter.

1.4.3 draw: plot a single table

The multi-track genome browsing system described above may be quite heavy to visualize a single track. In this case, one should consider using the `drawable` features directly:

```
>   # Drawable data format
>   data(hsFeatures)
>   genes <- track.table(hsGenes)
>   # Draw
>   genes$draw(chrom="8", start=15e6, end=20e6)
```

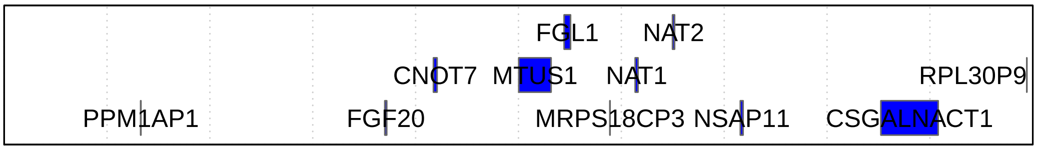


Drawing parameters may be changed for a single call:

```
>   print(genes$defaultParams()[1:5])

$height
[1] 1

$mar
[1] 0.2 5.0 0.2 1.0
```

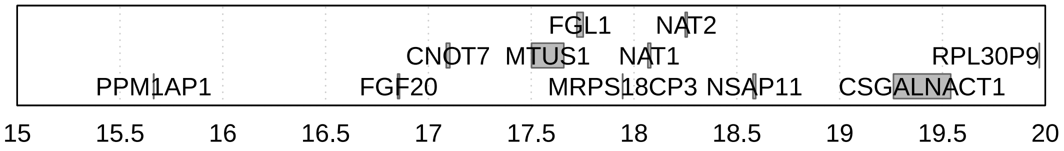
```
$new
[1] FALSE

$drawFun
[1] "draw.boxes"

$ylab
[1] NA
```

```
>   genes$draw(chrom="8", start=15e6, end=20e6, colorVal="blue")
```



Or in a more persistent way:

```
>   # Session persistent
>   print(genes$defaultParams()[["mar"]])

[1] 0.2 5.0 0.2 1.0

>   genes$setParam("mar", c(1.5, 5.0, 0.2, 1.0))
>   genes$draw(chrom="8", start=15e6, end=20e6)
>   # Save to file with custom parameters
>   saveRDT(genes, file="genes.rdt")
```



2 Manipulating the objects

2.1 Reference classes reminder

All the classes used by Rgb are defined as ”reference classes” (not to be confused with ”S3” and ”S4” systems). Complete documentation on this system can be found in the R manuals, however here is a quick reminder on this system, more similar to Java’s Object-Oriented paradigm than R’s ”Function-Oriented” one.

```
>   help("setRefClass")
```

2.1.1 Methods are called from objects, using the $ sign

```
>   data(hsFeatures)
>   hsBands$fill(1:5, "stain", LETTERS[1:5])
>   hsBands$getColNames()

[1] "name"   "chrom"  "strand" "start"  "end"    "stain"
```

2.1.2 Objects are only copied on explicit demand

```
>   data(hsFeatures)
>   a <- hsBands
>   a$getColNames()

[1] "name"   "chrom"  "strand" "start"  "end"    "stain"

>   a$delColumns("stain")
>   hsBands$getColNames()

[1] "name"   "chrom"  "strand" "start"  "end"

>   hsCopy <- hsBands$copy()
```

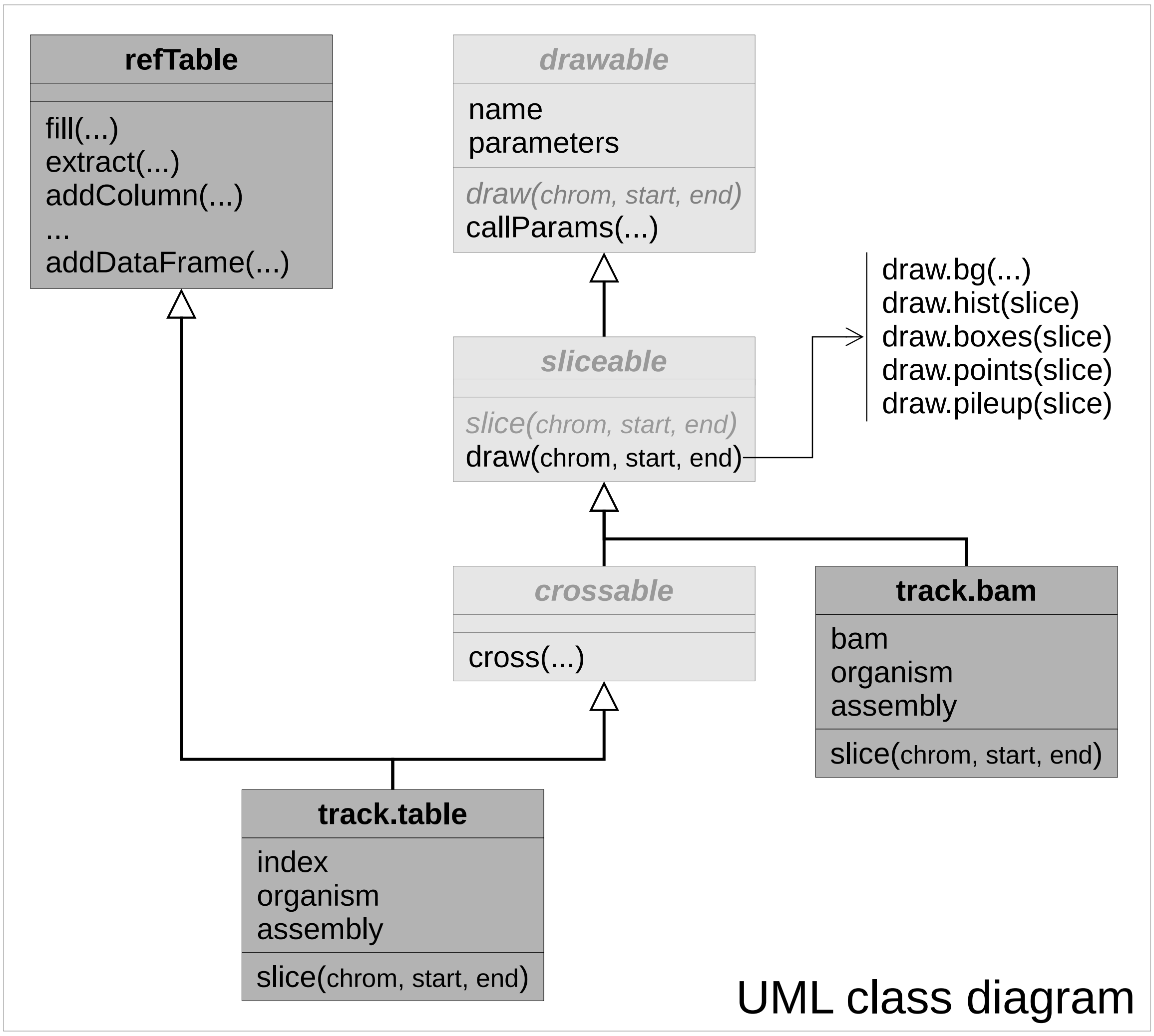
2.1.3 Classes are self-documented objects

```
>   classDefinition <- getRefClass("sliceable")
>   classDefinition$methods()

 [1] "callParams"              "callSuper"               "check"
 [4] "chromosomes"             "copy"                    "defaultParams"
 [7] "defaultParams#drawable"  "draw"                    "draw#drawable"
[10] "export"                  "field"                   "fix.param"
[13] "getChromEnd"             "getChromEnd#drawable"    "getClass"
[16] "getName"                 "getParam"                "getRefClass"
[19] "import"                  "initFields"              "initialize"
[22] "setName"                 "setParam"                "show"
[25] "show#drawable"           "show#envRefClass"        "slice"
[28] "trace"                   "untrace"                 "usingMethods"

>   classDefinition$help("draw")

Call:
$draw(chrom, start = , end = , ...)


Draws the object content corresponding to the defined genomic window, usually in a single plot area w
- chrom   : single integer, numeric or character value, the chromosomal location. NA is not required
- start   : single integer or numeric value, inferior boundary of the window. NA is not required to b
- end     : single integer or numeric value, superior boundary of the window. NA is not required to b
- ...     : additionnal drawing parameters (precede but do not overwrite parameters stored in the obj
```

2.1.4 Classes inherit methods and parameters

```
>   # All "track.table" objects are "drawable" objects
>   class(hsBands)

[1] "track.table"
attr(,"package")
[1] "Rgb"

>   is(hsBands, "drawable")

[1] TRUE

>   # Many "track.table" methods are defined by "drawable" class
>   dw <- getRefClass("drawable")
>   tl <- getRefClass("track.table")
>   intersect(dw$methods(), tl$methods())

 [1] "callParams"        "callSuper"         "check"             "chromosomes"
 [5] "copy"              "defaultParams"     "draw"              "export"
 [9] "field"             "fix.param"         "getChromEnd"       "getClass"
[13] "getName"           "getParam"          "getRefClass"       "import"
[17] "initFields"        "initialize"        "setName"           "setParam"
[21] "show"              "show#envRefClass" "trace"             "untrace"
[25] "usingMethods"
```

2.2 Rgb class hierarchy



Rgb heavily relies on class inheritance to avoid code duplication. As can be seen on the UML diagram above, objects from the `track.table` class (most of the objects manipulated in Rgb) are a convergence between the `refTable` class (that stores tabular data in an efficient way) and `drawable`-inheriting classes (managing all the drawing process, independently from the data type).

The `sliceable` class in the `drawable` inheritance tree only implements the straight-forward way to plot genomic data: in most cases, the dataset must first be ”sliced” (only data located in the queried genomic window are extracted), then the slice must be provided to the suitable drawing function. As a result, all classes inheriting from `sliceable` only need to implement an appropriate `slice` method to make use of the existing drawing engine (`track.table` and `track.bam` are two examples of very dissimilar data types that benefit from the same drawing system).

The `crossable` class in the `drawable` inheritance tree offers a `cross` method to classes implementing a `slice` method, allowing one track to be annotated according to its overlaps with a second one.

2.3 refTable: tabular data storage

Rgb defines a new class to store tabular data: the `refTable` class. Such objects are very similar to R classical `data.frames`, and differ mainly in the way data is stored and extracted. The purpose of this class is to handle such data in a more efficient way, avoiding frequent copies due to R copy-on-write paradigm, within the strict context of object oriented programming. Please refer to the ”Reference classes” chapter above for a quick reminder on reference class pitfalls in R.

Data can be imported in such objects using the `refTable` constructor, with a single `data.frame` or a collection of vectors:

```
>   library(Rgb)
>   df <- data.frame(colA=letters[1:5], colB=5:1)
>   rt <- refTable(df)
>   rt <- refTable(colA=letters[1:5], colB=5:1)
>   print(rt)
```

```
  "refTable" reference class object

  colA colB
1    a    5
2    b    4
3    c    3
4    d    2
5    e    1
```

Extraction from `refTable` objects is handled by the `extract` method, which returns data.frames or vectors. As with the R classical "[", lines can be extracted using several vector types: integers for row indexes, (recycled) logicals, or characters if row names were provided to refTable. Additionaly, an expression using column names and returning such a vector can be used, similarly to `with` behavior in data.frames:

```
>   library(Rgb)
>   data(hsFeatures)
>   rf <- refTable(hsGenes)
>   rf$extract(1:5)
```

```
     name chrom start   end strand
1  WASH7P     1 14362 29370      -
2 FAM110C     2 38814 46588      -
3  ZNF595     4 53227 88099      +
4 OR4G11P     1 63016 63885      +
5 DEFB125    20 68351 77296      +
```

```
>   rf$extract(c(TRUE, rep(FALSE, 799)))
```

```
          name chrom     start       end strand
1       WASH7P     1     14362     29370      -
801   C19orf38    19  10959106  10980360      +
1601     HMGCL     1  24128367  24151949      -
2401    DLGAP4    20  34995444  35157040      +
3201 LINC00316    21  46758505  46761910      -
4001     PRR11    17  57232860  57284070      +
4801   TAF13P2     2  74578213  74578546      +
5601    GPR183    13  99946789  99959749      -
6401     ROPN1     3 123687878 123710199      -
7201   HMGN3P1     1 152371939 152376159      +
```

```
>   rf$extract(expression(name == "RDX"))
```

```
     name chrom     start       end strand
6001  RDX    11 110100166 110167437      -
```

```
>   rf$extract(expression(chrom == "X" & grepl("^AR", name)))
```

```
         name chrom     start       end strand
258      ARSD     X   2822011   2847392      -
268      ARSF     X   2958275   3030770      +
3256     ARAF     X  47420516  47431320      +
4523     ARR3     X  69488185  69501690      +
5647   ARMCX4     X 100673266 100788446      +
6597 ARHGAP36     X 130192216 130223857      +
```

To mutate `refTable` objects, several methods are provided: `addList`, `addVectors` and `addDataFrame` to add new rows, `addColumn` and `delColumns` to update whole columns, and `fill` to modify single cells. Finally the `rowOrder` method can prove useful to play with row ordering or subsetting (keep in mind that `track.table` objects require rows to be genomically ordered, but `refTable` does not). See the `refTable-class` manual page for examples.

```
>   example(topic="refTable-class", package="Rgb")
```

2.4 track.table: genomically located tabular data

The `track.table` inherits most of its methods from `refTable`, please have a look at the previous section to know how to handle such data. Objects from this class can be produced using the `track.table` constructor, in the same way as the `refTable` constructor presented above. Notice though the few required columns: chrom (factor), strand (”+” or ”-”), start (integer), end (integer) and name (character). Notice the following example raises warnings about meta-data missingness (organism, assembly ...), they can be provided or silenced by `warn = FALSE` in the `track.table` constructor.

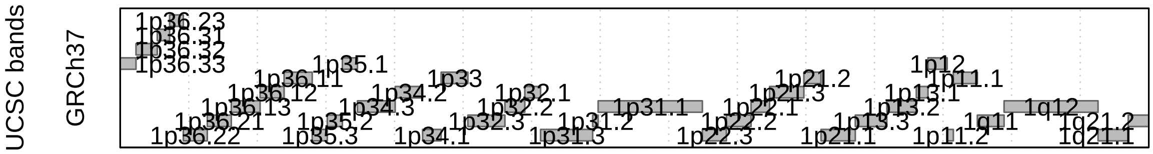
```
>   library(Rgb)
>   tl <- track.table(name=letters[1:5], chrom=1:5, strand="+", start=1:5, end=2:6)
>   df <- data.frame(chrom=1:5, strand="+", start=1:5, end=2:6)
>   tl <- track.table(df, .makeNames=TRUE, .organism="Human", warn=FALSE)
>   print(tl)

  "track.table" reference class object
  organism   : Human
  assembly   : NA

  Extends "drawable"
  name       : NA

  Extends "refTable"

    name chrom strand start end
1 chr1.0     1      +     1   2
2 chr2.0     2      +     2   3
3 chr3.0     3      +     3   4
4 chr4.0     4      +     4   5
5 chr5.0     5      +     5   6
```

The most valuable features of `track.table` are the `slice` and `cross` methods, which are presented in the ”Quick start” section of the current manual. Notice chromosomes can be armed (e.g. ”1p” and ”1q” in humans) or not, and two methods are provided to switch from one representation to the other (`eraseArms` and `addArms`, the former needing centromere positions as provided by the `track.bands.UCSC` function).

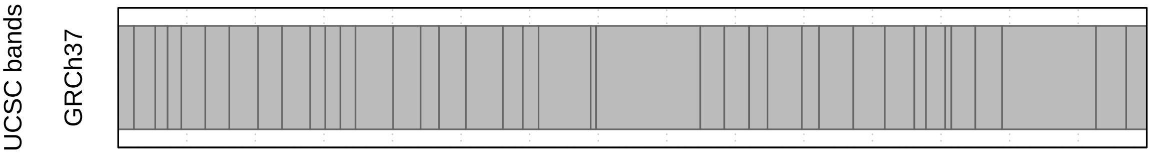
2.5 drawable: drawing management

The `drawable` class implements some features to be added to inheriting classes, such as `track.table`. It provides a common mechanism to manage drawing parameters, mainly through the `getParam` and `setParam` methods, and requires inheriting classes to define a `draw` method to plot its content in a given genomic window. `track.table` and `track.bam` both rely on the `sliceable` virtual class to implement such a method, itself requiring only a slice method to extract data in the given genomic window, and a drawing function to plot this slice of data.

```
>   library(Rgb)
>   data(hsFeatures)
>   hsBands$draw("1", 0, 150e6)
```

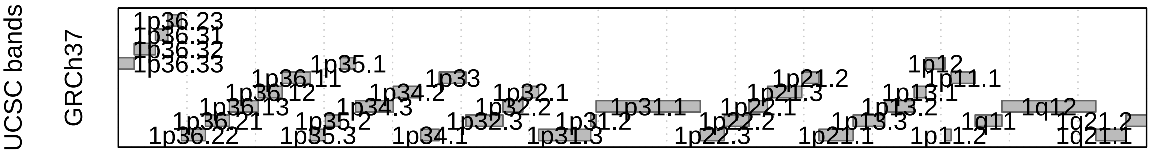


```
>   hsBands$getParam("drawFun")

[1] "draw.boxes"

>   hsBands$setParam("label", FALSE)
>   hsBands$draw("1", 0, 150e6)
```



Drawing parameters can be set at many levels, and are collated following a strict hierarchy: drawing parameters passed as additional arguments to `draw` have the top-most priority, preceding parameters set in objects (using the `setParam` method), which precede class-specific defaults (defined at class definition, see 5) following the inheritance order (`track.table` defaults are prioritary over `drawable` and so on). At the lowest priority are the default values of the R drawing functions (named in the ”drawFun” parameter).

```
>   library(Rgb)
>   data(hsFeatures)
>   hsBands$setParam("label", FALSE)
>   hsBands$draw("1", 0, 150e6, label=TRUE)
```



If a parameter was not defined specifically in an object, a call to `getParam` will rely on `defaultParams` to return a value. To erase an object-level setting (and thus turn back to class-specific default), just call `setParam` without value. Notice default parameters returned by `defaultParams` depend on the drawing function currently defined by the ”drawFun” drawing parameter itself:

```
>   library(Rgb)
>   data(hsFeatures)
>   hsBands$getParam("drawFun")

[1] "draw.boxes"

>   names(hsBands$defaultParams())

 [1] "height"        "mar"           "new"           "drawFun"
 [5] "ylab"          "ysub"          "xaxt"          "yaxt"
 [9] "yaxs"          "yaxp"          "ylim"          "cex.lab"
[13] "bty"           "las"           "xgrid"         "maxElements"
[17] "maxDepth"      "label"         "labelStrand"   "labelCex"
[21] "labelSrt"      "labelAdj"      "labelOverflow" "labelFamily"
[25] "colorVal"      "colorFun"      "border"        "spacing"
[29] "groupBy"       "groupPosition" "groupSize"     "groupLwd"

>   hsBands$setParam("drawFun", "draw.points")
>   names(hsBands$defaultParams())

 [1] "height"   "mar"      "new"      "drawFun"  "ylab"     "ysub"
 [7] "xaxt"     "yaxt"     "yaxs"     "yaxp"     "ylim"     "cex.lab"
[13] "bty"      "las"      "xgrid"    "column"   "colorVal" "colorFun"
[19] "cex"      "pch"
```

3 User case : ATM mutations in human

This user case introduces the handling of BAM files in a well annotated organism, for which custom annotation tracks can be found at the UCSC. It makes extensive use of the interactive genome browser, but also describes an automation scheme.

3.1 Objectives

The ATM gene was sequenced by the Ion Torrent technology in a tumoral sample, and we would like to visualize the mutations found. The resulting BAM file and its BAI index, as processed and aligned by the Torrent Suite, are provided with the Rgb package:

```
>   library(Rgb)
>   system.file("extdata/ATM.bam", package="Rgb")
```

```
[1] "/tmp/RtmpFhk6k2/Rinst32a25e815287/Rgb/extdata/ATM.bam"
```

```
>   system.file("extdata/ATM.bam.bai", package="Rgb")
```

```
[1] "/tmp/RtmpFhk6k2/Rinst32a25e815287/Rgb/extdata/ATM.bam.bai"
```

To minimize their impact on the Rgb package size, they were downsampled to 2000 reads from chromosome 11. However Rgb can handle standard-sized BAMs as well, and users are encouraged to use one of them if available.

3.2 Interactive browsing

3.2.1 Launch the interactive browser

As a first approach to the dataset, we will begin to browse it interactively. First a track file must be produced from the data, and saved as a file that can be imported in the interactive genome browser:

```
>   track <- track.bam(
+     bamPath = system.file("extdata/ATM.bam", package="Rgb"),
+     .organism = "Human",
+     .assembly = "hg19"
+   )
>   saveRDS(track, file="ATM.rds")
```

Then the genome browser can be launched:

```
>   tk.browse()
```

Hit the ”Tracks” button, ”Add file” and select the ”ATM.rds” file produced above (you may need to change the extensions shown as by default only ”.rdt” files are displayed). Wait for the track to appear in the list, and hit ”Done” to go back to the main window.

As no region was defined, there is currently nothing drawn. However to get a first idea of our data, let’s have a look at the ”chr11:108225450-108225660” region: write ”11”, ”108.22545” and ”108.22566” in the coordinate boxes (notice the coordinates are expected in megabases, Mb). Hit the ”Enter” key or the ”Jump” button.

Data is represented as a **pileup**, an histogram of all nucleotide frequences at each position of the genome. The arrow keys can be used to zoom in and out, and to move left and right. Zooming in will allow the nucleotide letters to appear, while zooming out may disable the representation (”maxRange reached”).

3.2.2 Add standard annotation

To make the interpretation simpler, let’s gather some basic annotation, beginning with datasets already handled by Rgb. This can be achieved in the R terminal while the genome browser is running. Firstly the cytogenetic banding, to be downloaded from UCSC:

```
>   track <- track.bands.UCSC(
+     file = "cytoBand.txt.gz",
+     .organism = "Human",
+     .assembly = "hg19"
+   )
>   saveRDT(track, file="cytoBands.rdt")
```

Then the coding sequences (exons), to be downloaded from the CCDS database:

```
>   track <- track.exons.CCDS(
+     file = "CCDS.current.txt",
+     .organism = "Human",
+     .assembly = "hg19"
+   )
>   saveRDT(track, file="exons.rdt")
```

Both can be added from the ”Tracks” interface of the genome browser seen previously. After closing the ”Tracks” window with the ”Done” button, hit the ”R” key or an arrow key to refresh the plot.

3.2.3 Customize the representation

The order of the tracks may not be optimal. Try to move the cytoband track to the top: open the ”Tracks” window, check the radio button in the ”Action” column for the ”cytoBands.rdt” file, and hit ”Move up”. When finished, hit ”Done”, and refresh the plot as previously.

The exon track may be too large for the small content it has to offer in this genomic location. Hit the ”Tracks” window, check the ”exons.rdt” line as previously and hit ”Edit”. The first menu on the left allows switching the track currently edited, while the menu on the right lists the parameters that can be edited. Search ”height”, and replace ”1” by ”0.5”. Hit ”Save in memory” or ”Save in file”, according to your preference (in the first case, changes will be lost at the closure of the genome browser). Hit ”Close and discard” to close this window, ”Done”, and refresh the plot. This parameter is particularly important, as it controls the relative sizes of tracks: by default most are set to 1, which means that every track with such a value must have the same height, equally sharing the window size. A value of 0.5 will result in a track twice as short as the others, and so on. Fixed heights can also be obtained, defining the height in centimeters (”3 cm”).

As you may have noticed, zooming out quickly results in ”maxRange reached” messages. The maximum range at which the plot is no longer displayed is a parameter that can be changed. As for the ”height” updated previously, try to replace the ”5000L” by a larger value (the ”L” is optional and only enforces the value as an integer). Beware however as the drawing time will increase exponentially when large pileups are to be plotted ...

By default, the Y axis is dynamic, which can make the comparison of depth of coverage tricky. As previously, try to replace the ”NA” value of the ”ylim” parameter by ”c(0, 50)” (a vector of two values defining the minimum and maximum to use). If you are familiar with R, this syntax should be evident: each parameter actually supports any valid R expression as a value.

3.2.4 Add annotation from UCSC

As we are looking for mutations in a cancer sample, the location of all known COSMIC mutations may prove particularly valuable to interpret our data. The UCSC Table Browser allows such data to be downloaded from its Table Browser, let’s have a look at it:

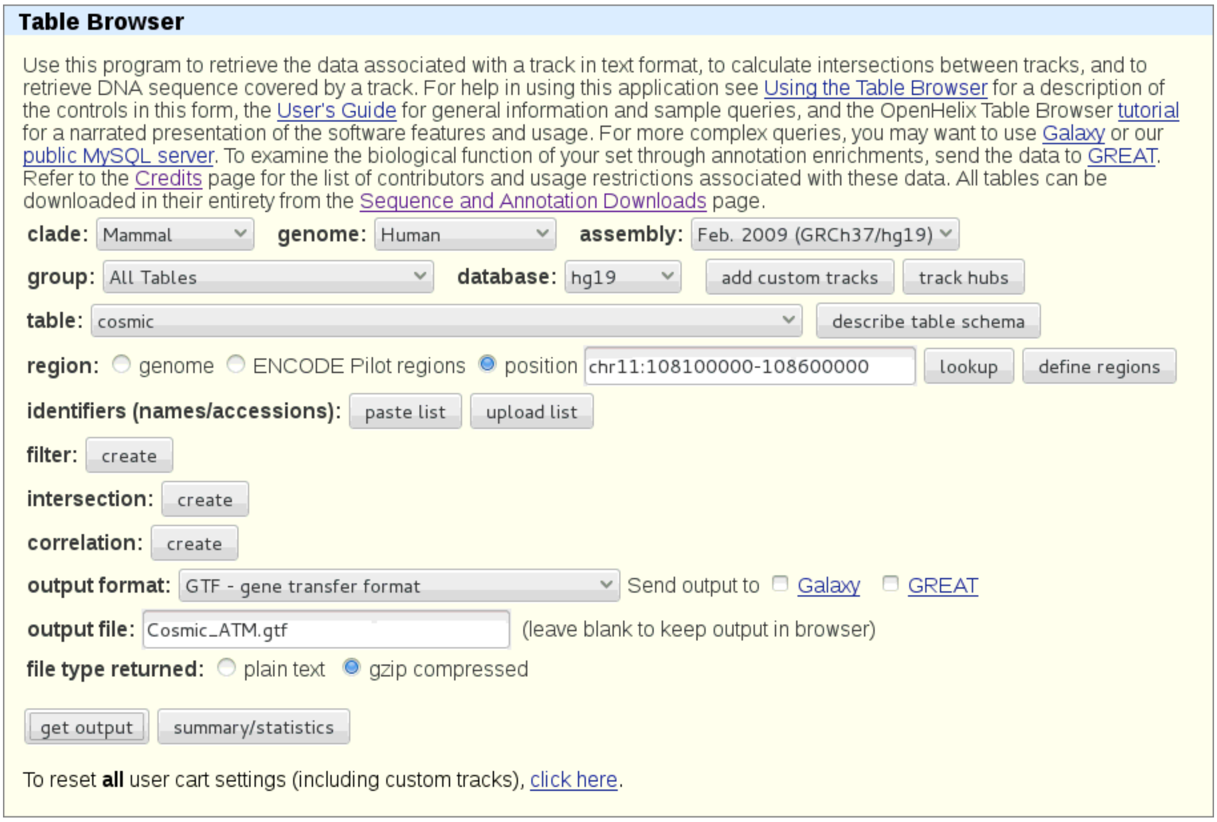


Table Browser

Use this program to retrieve the data associated with a track in text format, to calculate intersections between tracks, and to retrieve DNA sequence covered by a track. For help in using this application see Using the Table Browser for a description of the controls in this form, the User's Guide for general information and sample queries, and the OpenHelix Table Browser tutorial for a narrated presentation of the software features and usage. For more complex queries, you may want to use Galaxy or our public MySQL server. To examine the biological function of your set through annotation enrichments, send the data to GREAT. Refer to the Credits page for the list of contributors and usage restrictions associated with these data. All tables can be downloaded in their entirety from the Sequence and Annotation Downloads page.

clade: Mammal **genome:** Human **assembly:** Feb. 2009 (GRCh37/hg19)

group: All Tables **database:** hg19

table: cosmic

region: ○ genome ○ ENCODE Pilot regions ● position chr11:108100000-108600000

identifiers (names/accessions):

filter:

intersection:

correlation:

output format: GTF - gene transfer format Send output to □ Galaxy □ GREAT

output file: Cosmic_ATM.gtf (leave blank to keep output in browser)

file type returned: ○ plain text ● gzip compressed

To reset **all** user cart settings (including custom tracks), click here.

The region was restrained to the ATM gene location to minimize download time and disk usage, however feel free to download the whole COSMIC dataset. The resulting file is available in Rgb for testing purposes:

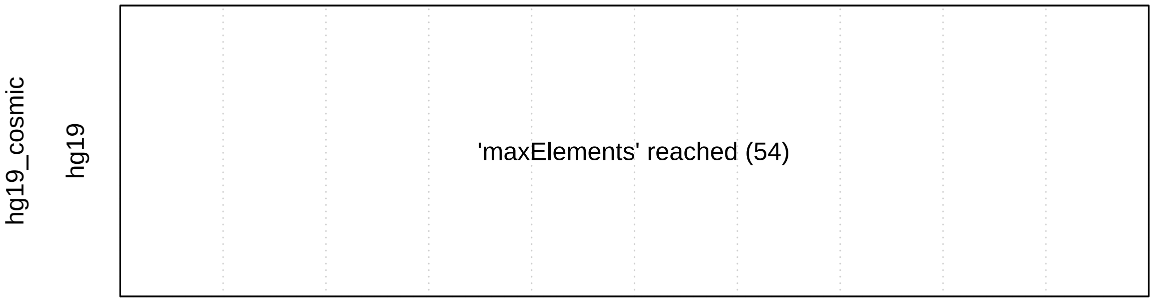
```
>   file <- system.file("extdata/Cosmic_ATM.gtf.gz", package="Rgb")
>   track <- track.table.GTF(file, .organism="Human", .assembly="hg19")
```

As GTF is a very generic file format, it usually requires a few modifications to produce statisfying visualizations. Let's have a look at it, using the `draw` method in a random region directly:

```
>   print(track)

  "track.table" reference class object
  organism    : Human
  assembly    : hg19

  Extends "drawable"
  name        : hg19_cosmic

  Extends "refTable"

            name chrom strand     start       end feature score frame
1       chr11.0     11   <NA> 108100018 108100018    exon     0    NA
2       chr11.1     11   <NA> 108100018 108100018    exon     0    NA
3       chr11.2     11   <NA> 108106424 108106424    exon     0    NA
...         ...    ...    ...       ...       ...     ...   ...   ...
1497 chr11.1496     11   <NA> 108594090 108594090    exon     0    NA
1498 chr11.1497     11   <NA> 108594137 108594137    exon     0    NA
1499 chr11.1498     11   <NA> 108594174 108594174    exon     0    NA

          gene_id transcript_id
1     COSM1585373   COSM1585373
2      COSM922697    COSM922697
```

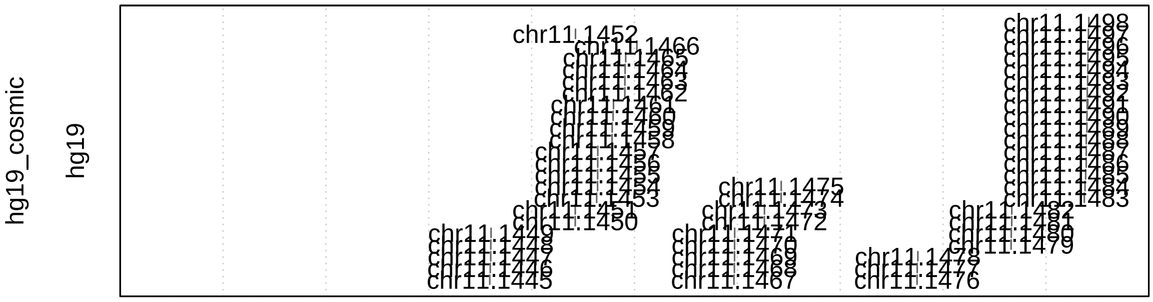
```
3     COSM1289449   COSM1289449
...           ...           ...
1497   COSM922828    COSM922828
1498 COSM1203207   COSM1203207
1499   COSM922829    COSM922829
```

```
>   track$draw("11", 108.5e6, 108.6e6)
```



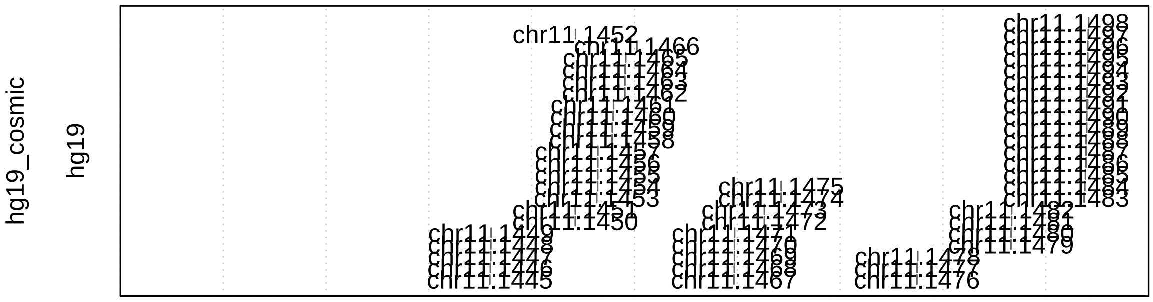
It seems that the track contains many records, so the "maxElements" limit that prevents overcrowded tracks to be plotted is suboptimal. Before making more permanent changes in the file or manual changes in the interactive browser, let's find a more optimal value by temporarly setting it in the `draw` call:

```
>   track$draw("11", 108.5e6, 108.6e6, maxElements=100)
```



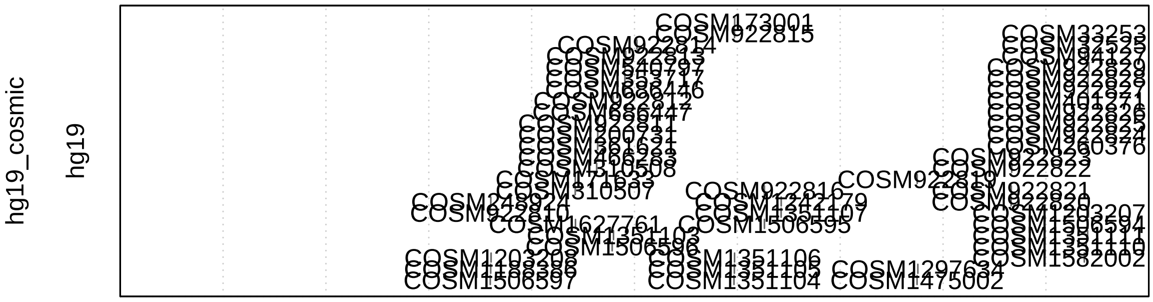
Once the appropriate value is found, we can edit it more permanently using the `setParam` method:

```
>   track$setParam("maxElements", 100)
>   track$draw("11", 108.5e6, 108.6e6)
```



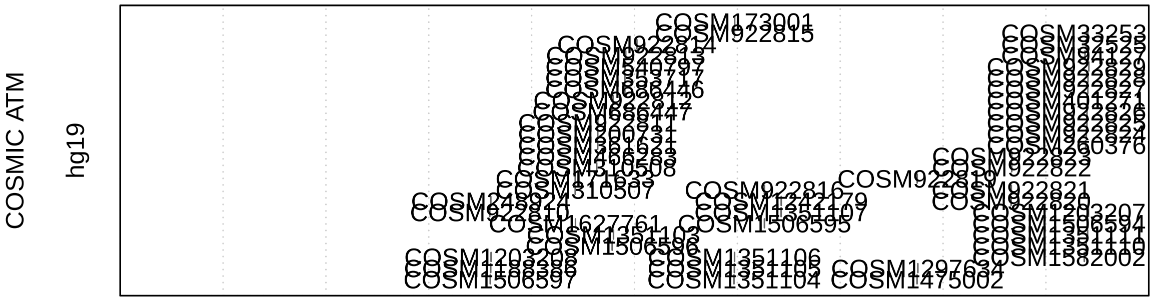
Now let's focus on record names. As Rgb is unable to predict which column of the table contains the data you wish to display as feature names, it generates names concatenating the chromosome location and an order number. As you can see using `print`, the "gene_id" and "transcript_id" columns seem to offer more interesting information (the COSMIC ID). As Rgb always use the "name" column to label features, all we have to do is replace its content by the content of the column of our choice:

```
>   newNames <- track$extract(,"gene_id")
>   track$fill(, "name", newNames)
>   track$draw("11", 108.5e6, 108.6e6)
```



To finish with a more cosmetic change, let's give a neater name to the track that the one extracted from the file:

```
>   track$name <- "COSMIC ATM"
>   track$draw("11", 108.5e6, 108.6e6)
```



Keep in mind that all the modifications we made here only apply to the object stored in R memory, it must be saved in an ".rdt" file to avoid the hassle of rerunning all these commands in each new R session. Once saved, we can import it in our current interactive browser window, using the "Tracks" button as previously.

```
>   saveRDT(track, file="COSMIC_ATM.rdt")
```

3.2.5 Manual check of the exons

Zooming out with the down arrow key, we are able to zoom into each exon to visualize the data, even if the "maxRange reached" message masks them at the gene level zooming. To zoom into a particular exon, just click the left mouse button at its start, move it to the ending position and release the button ("drag and drop").

3.3 Automation

3.3.1 Produce a single plot

All the operations manually performed above can be achieved using the `browsePlot` equivalent of the `tk.browse` function, in order to automatically generate representations of the 62 exons rather than manually jump to each of them. First the tracks must be collected in a `drawable.list` object:

```
>   dl <- drawable.list(
+     files = c(
+       "cytoBands.rdt",
+       "ATM.rds",
+       "exons.rdt",
+       "COSMIC_ATM.rdt"
+     )
+   )
```

Here they can be edited interactively by the `fix.param` method, and expanded using `fix.files` in the same way as in the interactive genome browser.

```
>   dl$fix.param()
>   dl$fix.files()
```

However to make the whole process automatic, we will edit them using commands only. Tracks in drawable lists may be selected using one of the following methods: getByNames, getByClasses and getByPositions, each of them returning tracks as an R list (which means further subsetting is required).

```
>   print(dl)

  "drawable.list" reference class object

        name       class
1 UCSC bands track.bands
2    ATM.bam   track.bam
3 CCDS exons track.exons
4 COSMIC ATM track.table

>   dl$getByNames("UCSC bands")

[[1]]

  "track.bands" reference class object

  Extends "track.table"
  organism   : Human
  assembly   : hg19

  Extends "drawable"
  name       : UCSC bands

  Extends "refTable"

        name chrom strand    start      end  stain
1    1p36.33     1      +        1  2300000   gneg
2    1p36.32     1      +  2300000  5400000 gpos25
3    1p36.31     1      +  5400000  7200000   gneg
...      ...   ...    ...      ...      ...    ...
860 Yq11.223     Y      + 22100000 26200000 gpos50
861  Yq11.23     Y      + 26200000 28800000   gneg
862     Yq12     Y      + 28800000 59373566   gvar
```

To obtain a list of handled parameters as the interactive browser offers, the defaultParams method may prove useful:

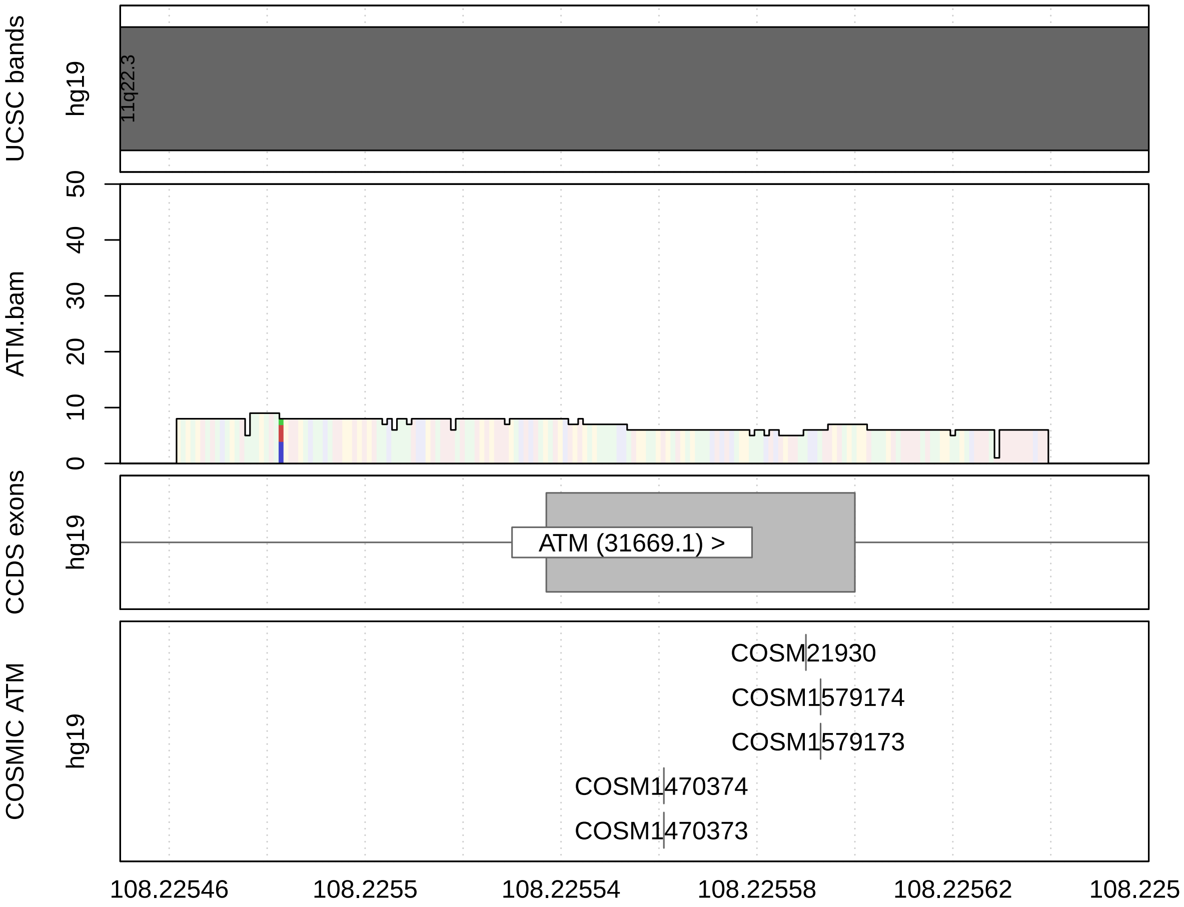
```
>   target <- dl$getByNames("UCSC bands")[[1]]
>   names(target$defaultParams())

 [1] "height"        "mar"           "new"           "drawFun"
 [5] "ylab"          "ysub"          "xaxt"          "yaxt"
 [9] "yaxs"          "yaxp"          "ylim"          "cex.lab"
[13] "bty"           "las"           "xgrid"         "maxElements"
[17] "maxDepth"      "label"         "labelStrand"   "labelCex"
[21] "labelSrt"      "labelAdj"      "labelOverflow" "labelFamily"
[25] "colorVal"      "colorFun"      "border"        "spacing"
[29] "groupBy"       "groupPosition" "groupSize"     "groupLwd"
```

Then the setParam method can be called to update the parameters. As reference classes are used here, feel free to store getByNames output in an intermediary variable, any modification applied to the intermediate will also apply to the drawable list:

```
>   dl$getByNames("CCDS exons")[[1]]$setParam("height", 0.5)
>   target <- dl$getByNames("ATM.bam")[[1]]
>   target$setParam("maxRange", 8000)
>   target$setParam("ylim", c(0, 50))
```

Finally the drawable list can be handed to `browsePlot` for plotting:

```
>   browsePlot(dl, chrom="11", start=108225450, end=108225660)
```



Keep in mind that `tk.browse` also handles drawable lists as input, a feature that can prove useful for script debugging or to launch the browser with a predefined set of tracks.

```
>   tk.browse(dl)
```

3.3.2 Loop on exons

While automating a single representation may make its future modification easier, its strength resides in its ability to save a lot of manual manipulations. Let's try to generate representations using the previous settings on the 62 exons of ATM.

First we need to collect the coordinates of the exons to loop on. More than an input for `tk.browse` and `browse.plot`, the track files we produced are also easy to query datasets. Let's have a look at the exon track:

```
>   exons <- readRDT("exons.rdt")
>   print(exons)
```

```
"track.exons" reference class object

Extends "track.table"
organism    : Human
assembly    : hg19

Extends "drawable"
name        : CCDS exons

Extends "refTable"
```

```
            name chrom strand     start       end       transcript
1        CCDS5.1|4     1      -    934438    934811       HES4 (5.1)
2        CCDS5.1|3     1      -    934905    934992       HES4 (5.1)
3        CCDS5.1|2     1      -    935071    935166       HES4 (5.1)
...            ...   ...    ...       ...       ...              ...
191  CCDS6752.1|1     9      - 104133220 104133685    BAAT (6752.1)
192 CCDS14487.1|1     X      + 100807913 100809274 ARMCX1 (14487.1)
193 CCDS14772.1|1     Y      -   2655029   2655643    SRY (14772.1)

    groupPosition groupSize
1               4         4
2               3         4
3               2         4
...           ...       ...
191             1         3
192             1         1
193             1         1
```

As you can see, the track is basically a table with more than 300 000 rows and a few columns. The gene symbol in which to look for ATM is held in the ”transcript” column, which can be queried in-situ using the `extract` method:

```
>   loci <- exons$extract(expression(grep("^ATM ", transcript)))
>   print(head(loci))
```

```
            name chrom strand     start       end    transcript groupPosition
38 CCDS31669.1|1    11      + 108098351 108098422 ATM (31669.1)             1
39 CCDS31669.1|2    11      + 108098502 108098614 ATM (31669.1)             2
40 CCDS31669.1|3    11      + 108099904 108100049 ATM (31669.1)             3
41 CCDS31669.1|4    11      + 108106396 108106560 ATM (31669.1)             4
42 CCDS31669.1|5    11      + 108114679 108114844 ATM (31669.1)             5
43 CCDS31669.1|6    11      + 108115514 108115752 ATM (31669.1)             6
   groupSize
38        62
39        62
40        62
41        62
42        62
43        62
```

However R users not familiar with reference classes may find it easier (but slower) to convert the whole track to a `data.frame` and use base R mechanisms:

```
>   exonTable <- exons$extract()
>   print(head(exonTable))
```

```
           name chrom strand    start      end      transcript groupPosition
1     CCDS5.1|4     1      -   934438   934811      HES4 (5.1)             4
2     CCDS5.1|3     1      -   934905   934992      HES4 (5.1)             3
3     CCDS5.1|2     1      -   935071   935166      HES4 (5.1)             2
4     CCDS5.1|1     1      -   935245   935352      HES4 (5.1)             1
5 CCDS44469.1|1    10      + 99344460 99344670 HOGA1 (44469.1)             1
6 CCDS44469.1|2    10      + 99361613 99361746 HOGA1 (44469.1)             2
  groupSize
1         4
2         4
3         4
4         4
5         3
6         3
```

```
>   loci <- subset(exonTable, grepl("^ATM ", transcript))
```

Looping throught the loci is now straight-forward, and resulting representations can be derived to a plotting device such as a PDF file:

```
>   pdf("ATM.pdf", width=12)
>   for(i in 1:nrow(loci)) {
+     browsePlot(dl,
+       chrom = loci[i,"chrom"],
+       start = loci[i,"start"] - 150,
+       end = loci[i,"end"] + 150
+     )
+   }
>   dev.off()
```

4 User case : Gene expression mapping in A. thaliana

This user case shows how Rgb can be used on more confidential organisms, for which annotation data must be processed manually from heterogeneous data sources.

4.1 Objectives

In this user case, we are interested in visualizing gene expression mapping data in Arabidopsis thaliana. The dataset we are going to use is freely available in the Gene Expression Omnibus database of the NCBI, under the accession number GSM589609. As most micro-array datasets published nowadays are stored in GEO, it may prove valuable to know how to process data from this repository.

While some R packages from Bioconductor (GEOquery) propose downloading and pre-formatting data from the GEO database, we will process it manually to avoid Bioconductor's cumbersomeness and understand the whole process.

As the data files handled here can be very large (hundreds of Mo) and could not be distributed with the package, the current vignette was designed with files manually subset to a predefined window (chr1:16123000-16158000). However feel free to download the full data sets and run the commands on them, rather than using the distributed ones.

4.2 Micro-array data from GEO

4.2.1 Aggregate the dataset

On GEO, records are from two main types: **platforms**, documenting probesets, their annotations and locations in the genome, and **sample** data. The first step will be to download the two datasets, and merge them into a single track that can be handled by Rgb.

Many file formats are provided, however the "Full table" proposed by GEO has the advantage of being platform-independent. The sample (70 Mo) and platform (215 Mo) tables can be found on the dedicated webpages, and parsed using R basic mecanisms:

```
>   gpl <- read.table(
+     file = "GPL10855-34953.txt",
+     sep = "\t",
+     header = TRUE,
+     stringsAsFactors = FALSE
+   )
>   gsm <- read.table(
+     file = "GSM589609-38201.txt",
+     sep = "\t",
+     header = TRUE,
+     stringsAsFactors = FALSE
+   )
```

Printing the first lines, notice that both rely on a single "ID" column to identify probes:

```
>   head(gpl)
```

```
                        ID PROBE_TYPE                  SEQUENCE    RANGE_GB
1 X241_Y500_1_16124444_C_G        SNP AGGATCTGGAAACTAGAGGATAGAG NC_003070.9
2 X518_Y332_1_16124444_C_G        SNP CTCTATCCTCTAGTTTCCAGATCCT NC_003070.9
3 X241_Y499_1_16124444_C_G        SNP AGGATCTGGAAAGTAGAGGATAGAG NC_003070.9
4 X518_Y331_1_16124444_C_G        SNP CTCTATCCTCTACTTTCCAGATCCT NC_003070.9
5 X1007_Y26_1_16124498_T_G        SNP TTCCAGTTTGATTTTGACCATGAGA NC_003070.9
6 X237_Y626_1_16124498_T_G        SNP TCTCATGGTCAAAATCAAACTGGAA NC_003070.9
  RANGE_STRAND RANGE_START RANGE_END REFERENCE_ALLELE SNP_ALLELE
1            -    16124432  16124456                C          G
2            +    16124432  16124456                C          G
3            -    16124432  16124456                C          G
4            +    16124432  16124456                C          G
5            -    16124486  16124510                T          G
6            +    16124486  16124510                T          G
```

```
>   head(gsm)

                   ID_REF VALUE
1 X241_Y500_1_16124444_C_G  37.3
2 X518_Y332_1_16124444_C_G  37.9
3 X241_Y499_1_16124444_C_G  59.0
4 X518_Y331_1_16124444_C_G  50.1
5 X1007_Y26_1_16124498_T_G  58.8
6 X237_Y626_1_16124498_T_G 132.2
```

A quick check can confirm that the two tables are ordered in the same way, and thus can be merged side-by-side without further processing. Consider the `merge` function for cases where these conditions fail (it may take a few minutes of computation on datasets of this size):

```
>   nrow(gpl) == nrow(gsm)

[1] TRUE

>   all(gpl$ID == gsm$ID_REF)

[1] TRUE
```

The track can now be constructed using the generic `track.table` function, providing columns as vectors as for `data.frame`. Arguments beginning by dots are reserved for annotation meta-data to store along the dataset, which can be retrieved from the GEO webpage. Optional columns can also be picked from `gpl`, as long as the 5 mandatory columns (name, chrom, start, end, strand) are provided. Notice "start" and "end" requires columns to be of class "integer", and it may have to be enforced using `as.integer` if `read.table` didn't guess it while parsing:

```
>   cgh <- track.table(
+     name = gpl$ID,
+     chrom = gpl$RANGE_GB,
+     start = as.integer(gpl$RANGE_START),
+     end = as.integer(gpl$RANGE_END),
+     strand = gpl$RANGE_STRAND,
+     value = gsm$VALUE,
+     .name = "GSM589609",
+     .organism = "Arabidopsis thaliana",
+     .assembly = "TAIR9"
+   )
```

Printing the object, noticed that chromosome names are not very informative, which may cause conflicts with the future annotation sets:

```
>   cgh

  "track.table" reference class object
  organism   : Arabidopsis thaliana
  assembly   : TAIR9

  Extends "drawable"
  name       : GSM589609

  Extends "refTable"

                        name       chrom strand    start      end value
1      X694_Y1231_1_16124101 NC_003070.9      + 16124089 16124113  91.5
2     X1324_Y1096_1_16124170 NC_003070.9      - 16124158 16124182 137.1
3      X575_Y1359_1_16124302 NC_003070.9      + 16124290 16124314  66.1
...                      ...         ...    ...      ...      ...   ...
708 X95_Y257_1_16157855_A_G NC_003070.9      - 16157843 16157867  48.0
709 X1498_Y5_1_16157855_A_G NC_003070.9      + 16157843 16157867  74.8
710  X1360_Y1192_1_16157919 NC_003070.9      - 16157907 16157931  68.5
```

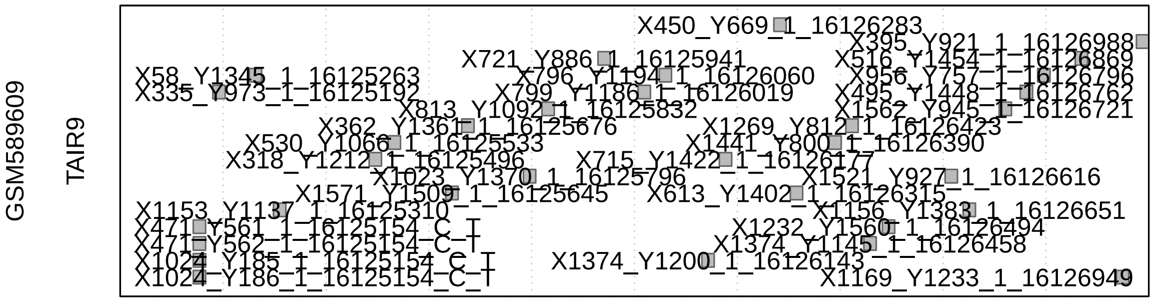
The coderefTable class proposes the `getLevels` and `setLevels` to its offspring, which can prove useful in this situation (correspondance between sequence accessions used here and general chromosome numbering can be found by querying Nucleotide at NCBI). To enforce compatibility with tracks produced in 4.3.3, we will also define levels for the chloroplastic (C) and mitochrondrial (M) genomes:

```
>   cgh$getLevels("chrom")

[1] "NC_003070.9"

>   cgh$chromosomes()

[1] "NC_003070.9"

>   cgh$setLevels("chrom", newLevels=c(1:5, "C", "M"))
>   cgh$chromosomes()

[1] "1" "2" "3" "4" "5" "C" "M"

>   cgh

  "track.table" reference class object
  organism   : Arabidopsis thaliana
  assembly   : TAIR9

  Extends "drawable"
  name       : GSM589609

  Extends "refTable"

                         name chrom strand    start      end value
1      X694_Y1231_1_16124101     1      + 16124089 16124113  91.5
2     X1324_Y1096_1_16124170     1      - 16124158 16124182 137.1
3      X575_Y1359_1_16124302     1      + 16124290 16124314  66.1
...                       ...   ...    ...      ...      ...   ...
708 X95_Y257_1_16157855_A_G     1      - 16157843 16157867  48.0
709 X1498_Y5_1_16157855_A_G     1      + 16157843 16157867  74.8
710  X1360_Y1192_1_16157919     1      - 16157907 16157931  68.5
```

4.2.2 Customize the representation

Now that the dataset is ready, let's have a look at how Rgb plots it:

```
>   cgh$draw(chrom="1", start=16125e3, end=16127e3)
```

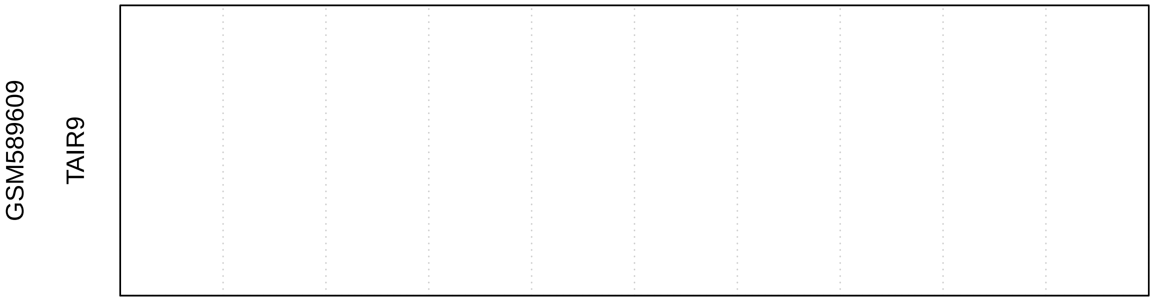


Pretty disappointing isn't it ? As a default, Rgb considers features in tracks as boxes, which is nice for most datasets (genes, exons, CNV ...) but suboptimal in this case. However it offers solutions to switch the representation mode, which relies on the **drawing function**. Three of them are provided for generic plotting: `draw.boxes` (default), `draw.hist` (for histograms) and `draw.points` (for scatter plots), but custom functions are also handled (see 5.1). The selected drawing function sets the other parameters that can be changed, so have a look at the corresponding manual page for an exhaustive list of them and their signification. Notice the three of them rely on `draw.bg` to plot the background, so parameters proposed by this function are always handled:

```
>   help(draw.boxes)
>   help(draw.points)
>   help(draw.hist)
>   help(draw.bg)
```

The scatter plot seems the most fitting for our dataset, let's give it a try. Parameters can be changed using the `setParam` method:

```
>   cgh$setParam("drawFun", "draw.points")
>   cgh$draw(chrom="1", start=16125e3, end=16127e3)
```



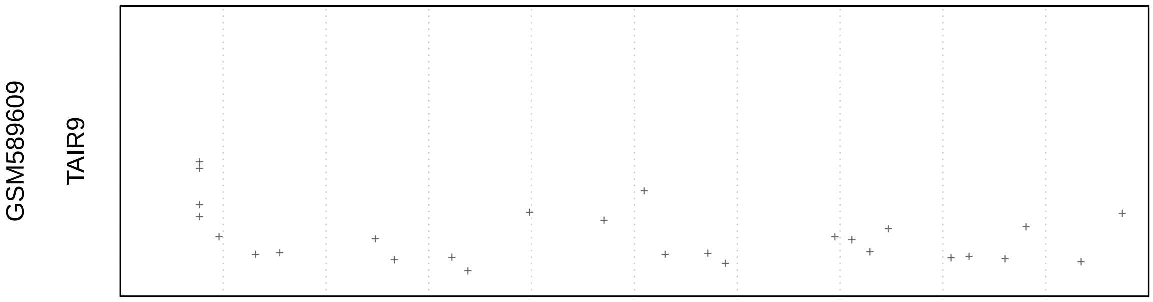
The queried region seems empty, but a quick look at the track content can confirm it is not. Actually there are features in the region, but parameters that were set for `draw.boxes` don't really fit the new drawing function. Comparing the handled parameters (as described in `draw.points`) and the current values can help pinpoint the problem:

```
>   cgh$defaultParams()$ylim
```

```
[1] 0 1
```

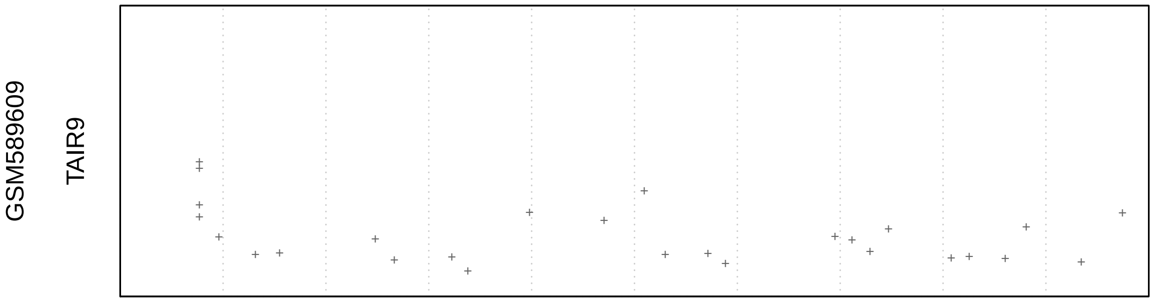
"ylim", which controls the Y axis boundaries, does not fit the data we used here (the "value" column ranges from units to thousands). Let's redefine it to a more suitable value. While `setParam` was used previously to make the change permanent, we are here tuning without really knowing the optimal value, so temporary changes may be more practical. Parameters can be passed directly to the `draw` method to be forgotten after the call:

```
>   cgh$draw(chrom="1", start=16125e3, end=16127e3, ylim=c(0, 500))
```



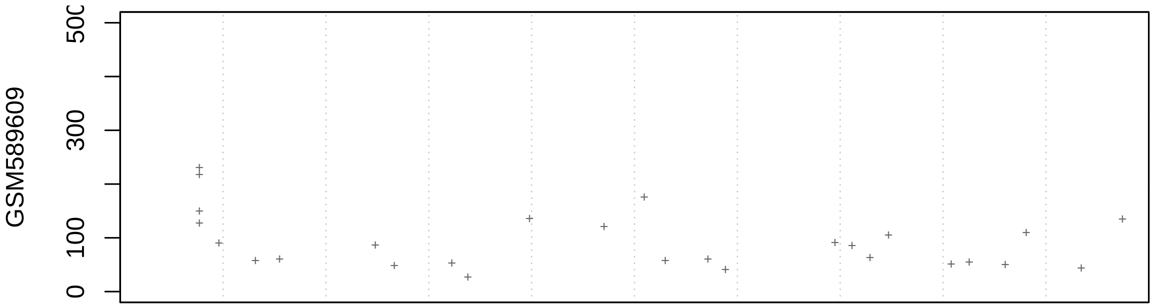
When satisfied by the value, the change can be made permanent:

```
>   cgh$setParam("ylim", c(0, 500))
>   cgh$draw(chrom="1", start=16125e3, end=16127e3)
```



While the Y axis is of better use like this, it still lacks an explicit legend. This parameter is handled by `draw.bg`, and to comply with R standards it is named ”yaxt” (see the documentation of `par`, which describes many standard parameters that are also handled by drawing functions):

```
>   cgh$setParam("yaxt", "s")
>   cgh$draw(chrom="1", start=16125e3, end=16127e3)
```



Once finalized, the track can be saved to a file for further use:

```
>   saveRDT(cgh, file="GSM589609.rdt")
```

4.3 Annotation from TAIR

To interpret this data, we now need some annotation. As for many species, biologists working on Arabidopsis thaliana have gathered into a community with a dedicated website, The Arabidopsis Information Resource (TAIR). On such species-specific websites, you can generally browse the data directly and download it in various file formats.

4.3.1 Note on assembly versions

When browsing the web for annotation data, a critical point must be kept in mind: genome assemblies vary, so take care to always compare data from the same assembly. For Arabidopsis thaliana, releases are numbered as TAIR7, TAIR8, TAIR9 and so on by the TAIR cited above. As the platform annotation we have downloaded refers to TAIR9, we need to download TAIR9 compliant annotation. Notice the sequences of the probes are available, so remapping them using Next-Generation Sequencing aligners like bowtie, bwa or simply blat can be considered, but it falls outside of this user case scope (consider the cghRA package to apply such methodology).

4.3.2 Tab-separated genetic markers

Let’s begin with the genomic markers, which can be downloaded from the TAIR FTP. The ”.data” extension seems very unfamiliar, so a first approach would be to open the file with a notepad, just as a first approach (or read the associated README file if it exists, it usually does on FTP servers). It seems to be a tabulation-separated file, a good point as R can easily parse them, with the minimal information required: chromosome number, starting and ending positions.

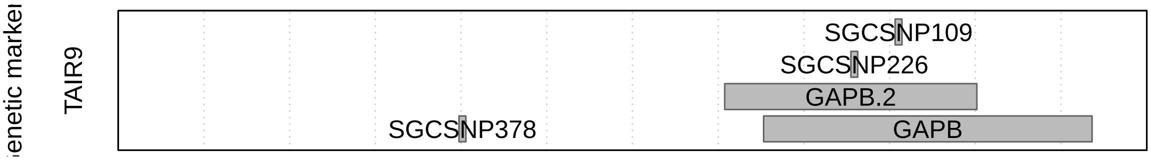
```
>   tab <- read.table(
+     file = "TAIR9_AGI_marker.data",
+     sep = "\t",
+     header = FALSE,
+     stringsAsFactors = FALSE
+   )
```

As previously, the data.frame can be turned into `track.table` without major difficulty. As no strand information is present but `track.table` requires it, we can use the `NA` value provided by R:

```
>   mrk <- track.table(
+     name = tab$V2,
+     chrom = tab$V5,
+     start = tab$V3,
+     end = tab$V4,
+     strand = NA,
+     .name = "Genetic markers",
+     .organism = "Arabidopsis thaliana",
+     .assembly = "TAIR9"
+   )
```

As previously, the track can be customised, but its content fits well with the default `draw.boxes` behavior, so it is ready to be saved to a track file. We will only enforce chromosome levels as for CGH data for future compatibility:

```
>   mrk$setLevels("chrom", newLevels=c(1:5, "C", "M"))
>   mrk$draw(chrom="1", start=16124e3, end=16130e3)
>   saveRDT(mrk, file="GeneticMarkers.rdt")
```



4.3.3 GFF3 exons

In the TAIR FTP, another file seems to fit our needs: the gene track. It is particularly interesting for this user case as it introduces two specificities: the widely spread GFF format, and the drawing of exonic tracks.

While Rgb does not provide an explicit GFF3 parser, users familiar with such file formats should have noticed that GTF is an extension of GFF3, and thus the `read.gtf` and `track.table.GTF` functions from Rgb are able to parse such files.

GFF3 files can store highly hierarchized content, describing relationships between genes, transcripts, exons, CDS, UTR ... The data is stored in a tabulation-separated file, which is nice to handle with R, but its ”attributes” column which allows row-specific data makes them hard to parse with `read.table`. Let’s use Rgb’s `read.gtf` function without subsetting to get an idea of the file complexity:

```
>   gtf <- read.gtf("TAIR9_GFF3_genes.gff")
>   head(gtf)
```

```
  seqname source         feature    start      end score strand frame
1    Chr1  TAIR9      chromosome        1 30427671    NA      .    NA
2    Chr1  TAIR9            gene 16125782 16127288    NA      +    NA
3    Chr1  TAIR9            mRNA 16125782 16127288    NA      +    NA
4    Chr1  TAIR9         protein 16125863 16127080    NA      +    NA
5    Chr1  TAIR9            exon 16125782 16125929    NA      +    NA
6    Chr1  TAIR9 five_prime_UTR 16125782 16125862    NA      +    NA
                  ID        Name                Note      Parent Index
1               Chr1        Chr1                <NA>        <NA>    NA
2          AT1G42960   AT1G42960 protein_coding_gene        <NA>    NA
3        AT1G42960.1 AT1G42960.1                <NA>   AT1G42960     1
4 AT1G42960.1-Protein AT1G42960.1               <NA>        <NA>    NA
5               <NA>        <NA>                <NA> AT1G42960.1    NA
6               <NA>        <NA>                <NA> AT1G42960.1    NA
  Derives_from
1         <NA>
2         <NA>
3         <NA>
4  AT1G42960.1
5         <NA>
6         <NA>
```

```
>   dim(gtf)
```

```
[1] 193  14
```

Records in such files are typed, according to the content of the ”feature” column:

```
>   table(gtf$feature)

            CDS      chromosome            exon  five_prime_UTR            gene
             73               1              73               8               8
           mRNA         protein three_prime_UTR
             11              11               8
```

In this user case, we will focus on exons. While parsing the whole table is interesting to get an idea of the file, keep in mind that `read.gtf` and `track.table.GTF` can subset on features earlier to make parsing faster:

```
>   gtf <- read.gtf("TAIR9_GFF3_genes.gff", features="exon")
>   trk <- track.table.GTF(
+     file = "TAIR9_GFF3_genes.gff",
+     name = "Exons",
+     attr = "split",
+     features = "exon",
+     .organism = "Arabidopsis thaliana",
+     .assembly = "TAIR9"
+   )
>   trk

  "track.table" reference class object
  organism   : Arabidopsis thaliana
  assembly   : TAIR9

  Extends "drawable"
  name       : Exons

  Extends "refTable"

          name chrom strand    start      end source feature score frame
1    chrChr1.0  Chr1      + 16125782 16125929  TAIR9    exon    NA    NA
2    chrChr1.1  Chr1      + 16126227 16126347  TAIR9    exon    NA    NA
3    chrChr1.2  Chr1      + 16126521 16126617  TAIR9    exon    NA    NA
...        ...   ...    ...      ...      ...    ...     ...   ...   ...
71  chrChr1.63  Chr1      + 16157246 16157374  TAIR9    exon    NA    NA
72  chrChr1.47  Chr1      + 16157456 16157911  TAIR9    exon    NA    NA
73  chrChr1.64  Chr1      + 16157461 16157911  TAIR9    exon    NA    NA

         Parent
1   AT1G42960.1
2   AT1G42960.1
3   AT1G42960.1
...         ...
71  AT1G43020.3
72  AT1G43020.1
73  AT1G43020.3
```

As GTF enforces a few columns that are not used in this dataset, let’s begin with freeing some wasted memory:

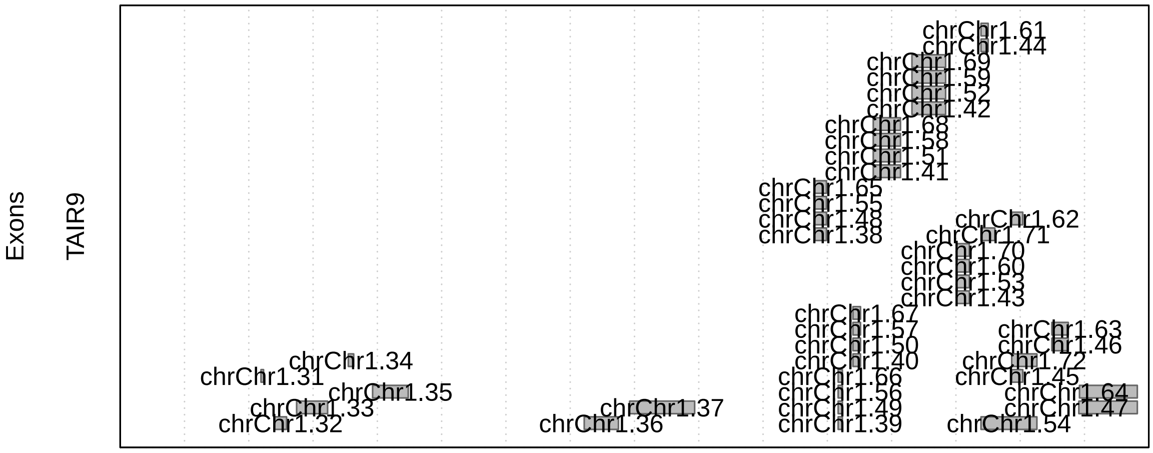
```
>   trk$delColumns(c("source","feature","score","frame"))
>   trk

  "track.table" reference class object
  organism   : Arabidopsis thaliana
  assembly   : TAIR9
  Extends "drawable"
  name       : Exons

  Extends "refTable"

          name chrom strand    start      end      Parent
1    chrChr1.0  Chr1      + 16125782 16125929 AT1G42960.1
2    chrChr1.1  Chr1      + 16126227 16126347 AT1G42960.1
3    chrChr1.2  Chr1      + 16126521 16126617 AT1G42960.1
...        ...   ...    ...      ...      ...         ...
71  chrChr1.63  Chr1      + 16157246 16157374 AT1G43020.3
72  chrChr1.47  Chr1      + 16157456 16157911 AT1G43020.1
73  chrChr1.64  Chr1      + 16157461 16157911 AT1G43020.3
```

As previously, chromosomes have to be recoded to be consistent with other tracks:

```
>   trk$setLevels("chrom", c(1:5, "C", "M"))
```

While the track is now functional, it is pretty unsatisfying:

```
>   trk$draw(chrom="1", start=16150e3, end=16158e3)
```



Exons are usually drawn side-by-side and grouped by transcript to make the reading easier. The `draw.boxes` function can handle such representations, as long as a littke more information is precomputed and passed to the function as drawing parameters:

groupBy The name of the column that stores transcript names

groupPosition The name of the column that stores exon number

groupSize The name of the column that stores exon count

While ”groupBy” can be directly set to ”Parent”, the two others need to be computed first. The `track.exons` class was designed to handle such datasets and provide convenient methods to build them, so let's begin with converting the generic `track.table` to the inheriting class `track.exons`. The second advantage to converting to `track.exons` is that it enforces several other drawing parameters that we no longer need to be concerned about, such as ”maxElements” or ”maxDepth”.

```
>   exn <- new("track.exons")
>   exn$import(trk)
>   exn
```

```
  "track.exons" reference class object

  Extends "track.table"
  organism   : Arabidopsis thaliana
  assembly   : TAIR9

  Extends "drawable"
  name       : Exons

  Extends "refTable"

          name chrom strand    start      end      Parent
1    chrChr1.0     1      + 16125782 16125929 AT1G42960.1
2    chrChr1.1     1      + 16126227 16126347 AT1G42960.1
3    chrChr1.2     1      + 16126521 16126617 AT1G42960.1
...        ...   ...    ...      ...      ...         ...
71  chrChr1.63     1      + 16157246 16157374 AT1G43020.3
72  chrChr1.47     1      + 16157456 16157911 AT1G43020.1
73  chrChr1.64     1      + 16157461 16157911 AT1G43020.3
```

Now the methods can be called, passing the grouping column:

```
>  exn$buildGroupSize("Parent", "exonCount")
>  exn$buildGroupPosition("Parent", "exonNumber")
>  exn
```

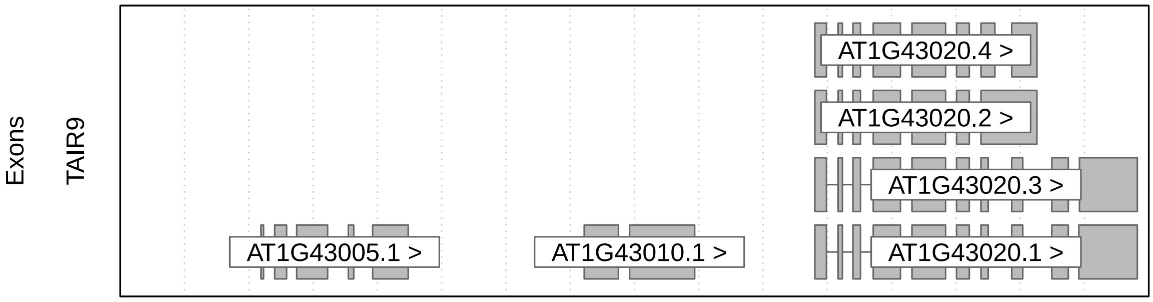
```
  "track.exons" reference class object

  Extends "track.table"
  organism   : Arabidopsis thaliana
  assembly   : TAIR9

  Extends "drawable"
  name       : Exons

  Extends "refTable"

          name chrom strand    start      end      Parent exonCount exonNumber
1    chrChr1.0     1      + 16125782 16125929 AT1G42960.1         5          1
2    chrChr1.1     1      + 16126227 16126347 AT1G42960.1         5          2
3    chrChr1.2     1      + 16126521 16126617 AT1G42960.1         5          3
...        ...   ...    ...      ...      ...         ...       ...        ...
71  chrChr1.63     1      + 16157246 16157374 AT1G43020.3        10          9
72  chrChr1.47     1      + 16157456 16157911 AT1G43020.1        10         10
73  chrChr1.64     1      + 16157461 16157911 AT1G43020.3        10         10
```

These new columns may be useful to define more informative row names:

```
>  newNames <- paste(exn$extract(,"Parent"), exn$extract(,"exonNumber"), sep="#")
>  exn$fill(, "name", newNames)
```

Finally the three drawing parameters cited above have to be updated to reflect our naming convention:

```
>  exn$setParam("groupBy", "Parent")
>  exn$setParam("groupPosition", "exonNumber")
>  exn$setParam("groupSize", "exonCount")
>  exn$draw(chrom="1", start=16150e3, end=16158e3)
>  saveRDT(exn, file="TAIR9 exons.rdt")
```

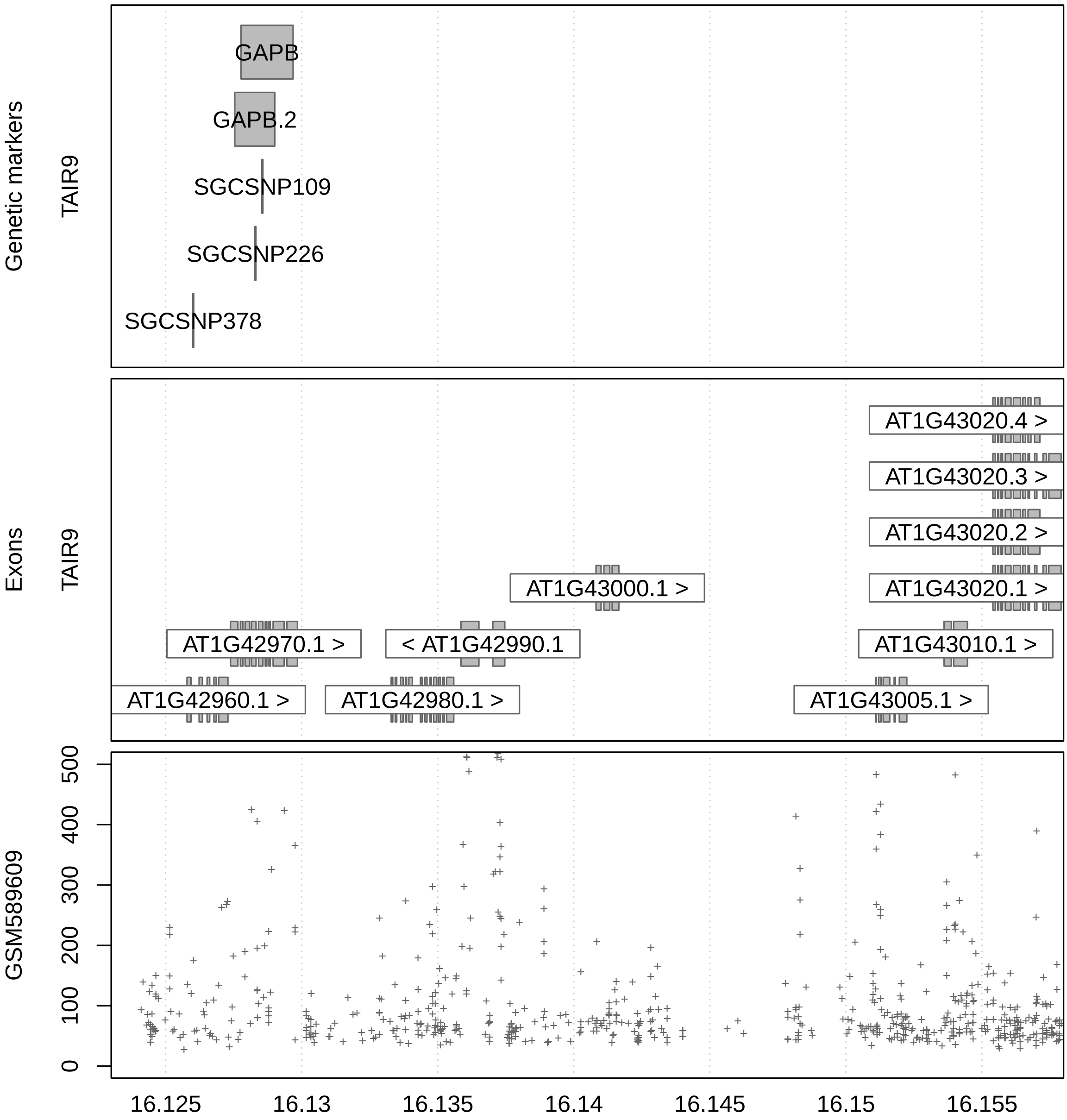


4.4 Integrated analysis

4.4.1 Visualization

As for the previous user case, browsing the datasets conjointly is now straight-forward, and can be achieved interactively by using commands. While `tk.browse` offers interfaced solutions to select tracks, both methods can be initiated from the same drawable list:

```
>   dl <- drawable.list()
>   dl$add(file="GeneticMarkers.rdt")
>   dl$add(file="TAIR9 exons.rdt")
>   dl$add(file="GSM589609.rdt")
>   browsePlot(dl, chrom="1", start=16123e3, end=16158e3)
```



```
>   tk.browse(dl)
```

4.4.2 Computation

The gene expression data used here is mapped to the genome rather than individual genes, but Rgb can help solving this issue. As RNA was hybridized during this experiment, it is judicious to limit our analysis to probes located in exons of genes of interest. Given the size of the exon track (roughly 200 000 rows), using `refTable`-inherited methods rather than converting to `data.frame` is recommended to minimize computing time and memory consumption. Let's focus on the gene homing the most expressed probe as an example:

```
>   gsm <- readRDT("GSM589609.rdt")
>   exn <- readRDT("TAIR9 exons.rdt")
>   gen <- gsm$extract(expression(which.max(value)), asObject=TRUE)
>   gen$cross(exn, type="Parent", colname="gene")
>   gen

  "track.table" reference class object
  organism   : Arabidopsis thaliana
  assembly   : TAIR9
  Extends "drawable"
  name       : GSM589609

  Extends "refTable"

                   name chrom strand    start      end   value        gene
1 X1234_Y1220_1_16128818     1      - 16128806 16128830 53079.3 AT1G42970.1

>   exn$extract(expression(Parent == gen$extract(,"gene")))

           name chrom strand    start      end      Parent exonCount
6  AT1G42970.1#1     1      + 16127381 16127653 AT1G42970.1         9
7  AT1G42970.1#2     1      + 16127744 16127839 AT1G42970.1         9
8  AT1G42970.1#3     1      + 16127919 16128083 AT1G42970.1         9
9  AT1G42970.1#4     1      + 16128153 16128320 AT1G42970.1         9
10 AT1G42970.1#5     1      + 16128412 16128572 AT1G42970.1         9
11 AT1G42970.1#6     1      + 16128657 16128718 AT1G42970.1         9
12 AT1G42970.1#7     1      + 16128795 16128844 AT1G42970.1         9
13 AT1G42970.1#8     1      + 16128947 16129353 AT1G42970.1         9
14 AT1G42970.1#9     1      + 16129452 16129843 AT1G42970.1         9
   exonNumber
6           1
7           2
8           3
9           4
10          5
11          6
12          7
13          8
14          9
```

While extract's default behavior is to return data.frame, it may be more interesting in our case to profit from track.table-inherited methods. As an example, the cross method may prove valuable to count probes overlapping with the exons of interest:

```
>   atg <- exn$extract(expression(Parent == gen$extract(,"gene")), asObject=TRUE)
>   atg$cross(gsm, type="count")

[1]  8  3  7  8  9  5  7 17  9

>   atg$cross(gsm, type="count", colname="probeCount")
>   atg

  "track.exons" reference class object

  Extends "track.table"
  organism   : Arabidopsis thaliana
  assembly   : TAIR9

  Extends "drawable"
  name       : Exons

  Extends "refTable"

            name chrom strand    start      end      Parent exonCount
1   AT1G42970.1#1     1      + 16127381 16127653 AT1G42970.1         9
2   AT1G42970.1#2     1      + 16127744 16127839 AT1G42970.1         9
3   AT1G42970.1#3     1      + 16127919 16128083 AT1G42970.1         9
...           ...   ...    ...      ...      ...         ...       ...
```

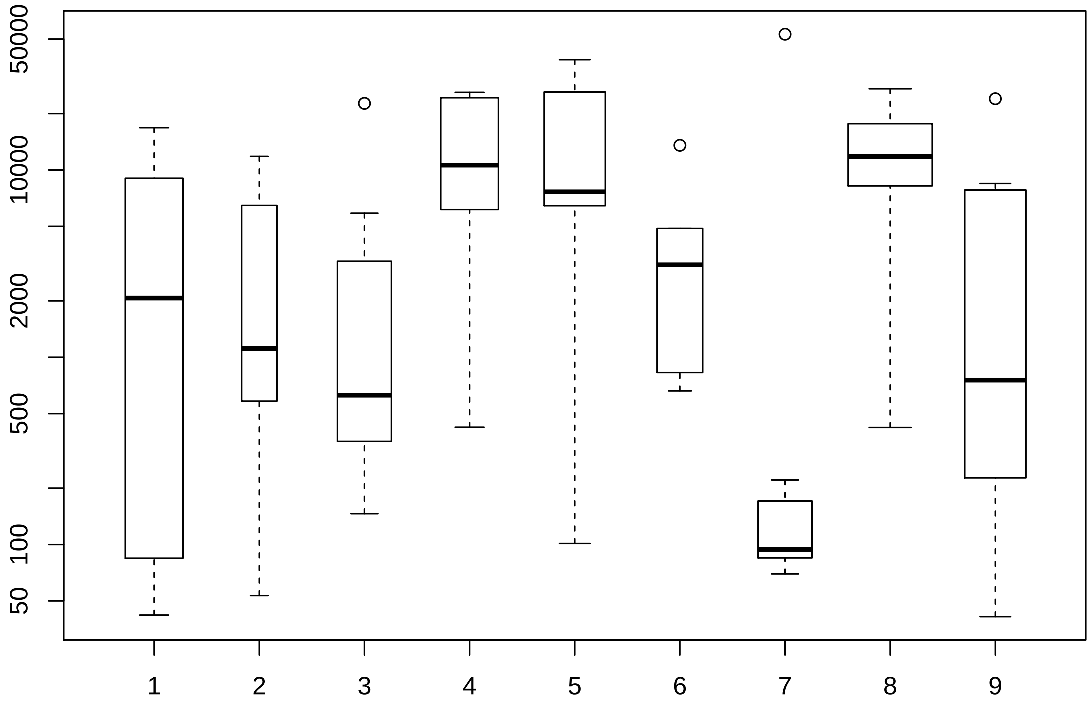
```
7   AT1G42970.1#7     1      + 16128795 16128844 AT1G42970.1         9
8   AT1G42970.1#8     1      + 16128947 16129353 AT1G42970.1         9
9   AT1G42970.1#9     1      + 16129452 16129843 AT1G42970.1         9

    exonNumber probeCount
1            1          8
2            2          3
3            3          7
...        ...        ...
7            7          7
8            8         17
9            9          9
```

As can be seen in the previous example, cross can be used to perform temporary computation or to output its computation in a new column. All exons seem to be covered, and it would be interesting to compute the mean expression for each of them. While cross proposes diverse computations, more custom schemes need to be scripted, using the slice method cross relies on:

```
>   atg$addColumn(
+     content = rep(as.double(NA), atg$getRowCount()),
+     name = "expr"
+   )
>   for(i in 1:atg$getRowCount()) {
+     probes <- gsm$slice(
+       chrom = atg$extract(i, "chrom"),
+       start = atg$extract(i, "start"),
+       end = atg$extract(i, "end")
+     )
+     atg$fill(i, "expr", mean(probes$value))
+   }
>   atg
  "track.exons" reference class object

  Extends "track.table"
  organism   : Arabidopsis thaliana
  assembly   : TAIR9

  Extends "drawable"
  name       : Exons

  Extends "refTable"

             name chrom strand    start      end      Parent exonCount
1   AT1G42970.1#1     1      + 16127381 16127653 AT1G42970.1         9
2   AT1G42970.1#2     1      + 16127744 16127839 AT1G42970.1         9
3   AT1G42970.1#3     1      + 16127919 16128083 AT1G42970.1         9
...           ...   ...    ...      ...      ...         ...       ...
7   AT1G42970.1#7     1      + 16128795 16128844 AT1G42970.1         9
8   AT1G42970.1#8     1      + 16128947 16129353 AT1G42970.1         9
9   AT1G42970.1#9     1      + 16129452 16129843 AT1G42970.1         9

    exonNumber probeCount      expr
1            1          8  4904.113
2            2          3  4327.533
3            3          7  4382.314
...        ...        ...       ...
7            7          7  7679.214
8            8         17 13055.794
9            9          9  5390.611
```

In the same spirit, expression by exon can easily be plotted using R functions:

```
>   expr <- list()
>   for(i in 1:atg$getRowCount()) {
+     probes <- gsm$slice(
+       chrom = atg$extract(i, "chrom"),
+       start = atg$extract(i, "start"),
+       end = atg$extract(i, "end")
+     )
+     expr[[i]] <- probes$value
+   }
>   boxplot(expr, varwidth=TRUE, log="y")
```



It suggests an interesting splicing pattern: exons 1, 2, 3, 6 and 9 seem to be expressed roughly at the same levels, exons 4, 5 and 8 at a higher level and exon 7 to not be expressed (value nearing the background level).

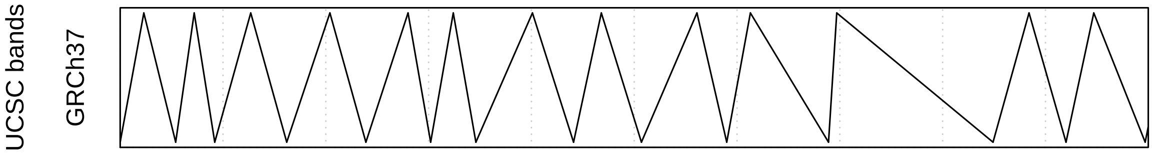
5 Extending Rgb capabilities

5.1 New representations of tabular content

Most genomic data is currently stored as a table of features, a format already handled by the `track.table` class. New data representations can thus be added to Rgb by expanding this class, and notably by developing new **drawing functions**. In the current implementation, a `track.table` that needs to be drawn in a given genomic window is first ”sliced”, then the slice is passed as a data.frame to the drawing function named in the ”drawFun” parameter. Defining a new drawing function that takes such a slice and the genomic coordinates as arguments is enough to allow Rgb to draw it, whether with the interactive genome browser or with the scripted interface.

It is heavily recommended that drawing function developers take a look at existing ones' code before starting (**draw.points**, **draw.hist** and **draw.boxes**). For `track.table` drawing functions, at least the ”slice”, ”start”, ”end” and ”...” arguments are mandatory, and the background of the plot should be drawn by a call to `draw.bg`.

```
>   # Really simple drawing function, just drawing lines
>   draw.custom <- function(slice, start, end, ...) {
+      draw.bg(start=start, end=end, ...)
+           segments(x0=slice$start, x1=slice$end, y0=0:1, y1=1:0)
+   }
>   # Edit a track to use it
>   data(hsFeatures)
>   hsBands$setParam("drawFun", "draw.custom")
>   # Let's draw
>   hsBands$draw("1", 0, 100e6)
```



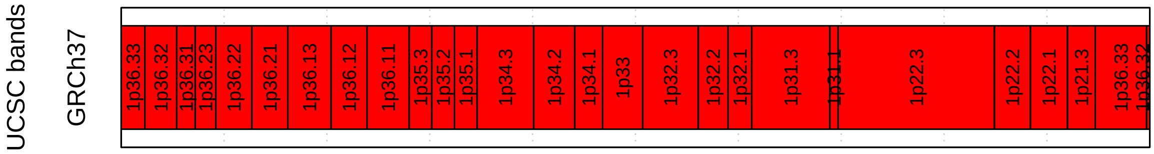
5.2 New drawing parameter defaults

As the available drawing functions are quite generic, it can prove useful to define standard behaviors for certain data types, and avoid a long series of `setParam` calls at each track construction. To achieve this, it is strongly recommended to define classes inheriting from `track.table`, enforcing new drawing parameters defaults by overloading the `defaultParams` method. As default parameters are collated along the inheritance tree from the children to the mothers (see 2.5), only new defaults are to be defined, superseding defaults defined in mother classes.

Notice class definitions are not stored in .rdt files, so class defaults can be updated without updating all the objects created from it: just redefine a class with the same name before the call to `readRDT`. Unfortunately, this behavior is not retrieved with drawable objects exported with `saveRDS` and this format may be abandoned in the future.

```
>   # Define a new class, just drawing red boxes
>   setRefClass(
+     Class = "track.custom",
+     contains = "track.bands",
+     methods = list(
+       defaultParams = function(...) {
+         params <- callSuper(...)
+         params$colorVal <- "red"
+         params$colorFun <- function(slice) NULL
+         return(params)
+       }
+     )
+   )
```

```
>   # Class switch
>   data(hsFeatures)
>   obj <- new("track.custom")
>   obj$import(hsBands)
>   # Let's draw
>   obj$draw("1", 0, 100e6)
```



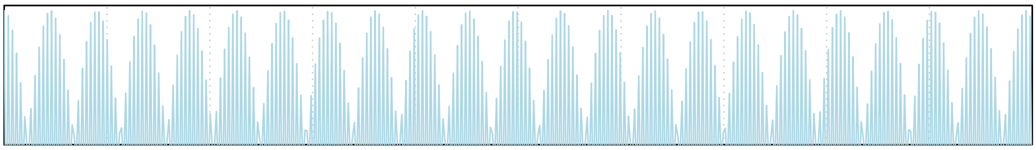
5.3 New data storage

The `track.table` class should fit most of your needs, but you may be confronted with data that cannot be stored efficiently as a single table of features. This is typically the case with external data sources like huge files or web services.

A first example of such a need is illustrated in Rgb by the **track.bam** class, designed lately to provide pileup representations of BAM files. As bioinformaticians may know, such files may be huge, and it is faster to query them directly via SAMtools in-situ than to preprocess their content to produce a tabular track. Such is designed `track.bam`, implementing a `slice` method relying on R binary file processing capabilities to generate pileups in the queried window on-the-fly. The pileup slice is then passed to a custom drawing function, `draw.pileup`, making use of the whole system described earlier. In most of the cases extending the `sliceable` or `crossable` classes is the simplest solution, only requiring you to define an inheriting class with a `slice` method and possibly some custom drawing functions (see Rgb source code for examples).

One can also be interested in defining more atypical tracks, for which ”slicing” is not pertinent. This can englobe tracks without embedded data such as scales, designed only to facilitate the reading. In such a case, the `drawable` class may be directly extended, providing a custom `draw` method. Such method should rely on the `callParams` method to handle drawing arguments in a consistent way, as `sliceable`’s `draw` method does. Consider also using the `draw.bg` function to plot a background consistent with other tracks:

```
>   setRefClass(
+     Class = "track.scale",
+     contains = "drawable",
+     methods = list(
+       defaultParams = function(...) {
+         # Define new class defaults
+         params <- callSuper(...)
+         params$col <- "lightblue"
+         params$points <- 500L
+         return(params)
+       },
+       draw = function(chrom, start=NA, end=NA, ...) {
+         # Aggregate and prioritize drawing parameters
+         argList <- callParams(chrom, start, end, ...)
+
+         # Plot background, using drawing parameters
+         do.call(what=draw.bg, args=argList)
+
+         # Data points
+         x <- seq(from=start, to=end, length.out=argList$points)
+         y <- cos(x)
+
+         # Plot, using drawing parameters
+         lines(x=x, y=y, col=argList$col)
+       }
+     )
+   )
>   object <- new("track.scale")
>   object$draw("11", 0, 10e6)
>   object$defaultParams()

$height
[1] 1

$mar
[1] 0.2 5.0 0.2 1.0

$new
[1] FALSE

$col
[1] "lightblue"

$points
[1] 500
```



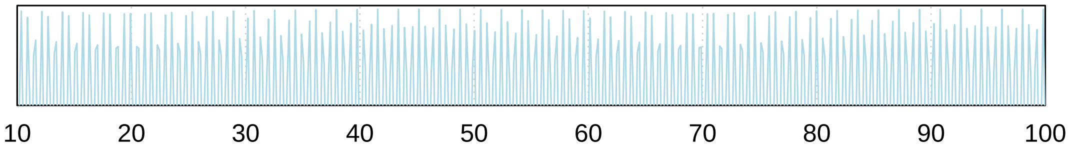
In the simple example above, notice that `draw.bg` arguments seem not to be handled as drawing parameters. Actually they do, but it is up to you to make them returned by `defaultParams`, as this behavior is implemented in `sliceable`. This is quite optional, but keep in mind that only parameters detected by `defaultParams` are editable via the GUI interface.

```
> defaultParams = function(...) {
+   # Get inherited defaults
+   params <- callSuper(...)
+
+   # Get draw.bg defaults
+   form <- formals("draw.bg")
+   form <- form[ setdiff(names(form), c("start", "end", "...")) ]
+   for(fname in names(form)) params[[ fname ]] <- eval(form[[ fname ]])
+
+   # Define new class defaults
+   params$col <- "lightblue"
+   params$points <- 500L
+
+   return(params)
+ }
```

As a good habit and to ensure consistent behavior, consider overloading the following virtual methods of the `drawable` class: `draw`, `defaultParams`, `show`, `check`, `chromosomes` and `getChromEnd`. However as can be seen in the example above, only the `draw` one is mandatory as most of them are provided as stubs by `drawable`.

To conclude with such custom classes, notice the `saveRDT` export function works only with `track.table` inheriting objects, so instanciated tracks need to be exported with `saveRDS`. This function is defined in R base and is supported by the Rgb interface.

```
>   saveRDS(object, "custom.rds")
>   dl <- drawable.list(files="custom.rds")
>   browsePlot(dl, chrom="1", start=10e6, end=100e6)
```


10
20
30
40
50
60
70
80
90
100