Package 'misha' - User Manual

July 9, 2020

'misha' package is intended to help users to efficiently analyze genomic data achieved from various experiments. The data must be stored in *Genomic Database* in certain format that is described later in this document. In addition the document describes fundamental concepts of the package such as *track expression*, *iterators*, etc.

Contents

1 Genomic Database

Genomic Database starts with a *root* (also frequently referred as *GROOT*), i.e. top directory containing certain subdirectories and files. A new database can be created using `gdb.create` function. This is the easiest way to do it. One can also build a database manually by generating all the necessary components that will be described later in this document.

Before the data in a Genomic Database can be accessed one must establish connection with it by calling `gdb.init` function. On launch the package connects to a Genomic Database located in `PACKAGEDIR/track-db/test` which serves all the examples in the reference manual.

A valid Genomic Database should contain the following files and subdirectories:

`chrom_sizes.txt` is a file containing the list of chromosomes and their sizes.

`tracks` is a directory that servers as a repository for all *tracks* and *interval sets*. May contain other subdirectories.

`pssms` is a directory containing PSSM sets (PSSM data and PSSM key files).

`seq` is a directory containing full genomic sequences.

`pssms` and `seq` directories are optional and are required only by a subset of functions in the package.

An example of a Genomic Database file structure:

```
hg18/              <- Genomic Database root directory
   chrom_sizes.txt
   .ro_attributes  <- List of read-only attributes
   pssms/             <- (optional)
      motif1.data        <- pssm data file
      motif1.key         <- pssm key file
      mypssm.data        <- ...
      mypssm.key         <- ...
   seq/               <- (optional)
      chr1.seq           <- seq (sequence) files
      chr2.seq           <- ...
      chr3.seq           <- ...
   tracks/
      tss.interv         <- small intervals set = tss
      big_data.interv/   <- big intervals set = big_data
         .meta              <- summary of the intervals set
         chr1               <- chrom files
         chr5               <- ...
      rpt.track/         <- track = rpt
         .attributes        <- track attributes (optional)
         chr1               <- chrom files
         chr2               <- ...
         chr3               <- ...
         vars/              <- track variables (optional)
             myresult           <- track variable
      test/
         intervals1.interv  <- intervals = test.intervals1
         track1.track/      <- track = test.track1
         .attributes        <- track attributes (optional)
         chr1               <- chrom files
         chr2               <- ...
         chr3               <- ...
```

```
savta/
   fourC.track/    <- track = savtra.fourC
      chr1               <- chrom files
      chr2               <- ...
      chr3               <- ...
```

2 File Formats

2.1 chrom_sizes.txt

`chrom_sizes.txt` file must be located under the root directory of Genomic Database. This file lists the chromosomes and their sizes. The chromosome name appears in the first column, the size is indicated in the second column. The chromosome name should appear without "chr" prefix. The two columns are separated by tab character. Example:

```
1    247249719
2    242951149
3    199501827
X    154913754
Y    57772954
```

2.2 Seq File

Seq (aka sequence) files are located in `seq` directory. Each of the Seq files contains a genomic sequence for a given chromosome as a contiguous string of ASCII characters. The length of the string should match the length of the chromosome. The file must be called `chrXXX.seq` where `XXX` indicates the name of the chromosome as it appears in `chrom_sizes.txt` file.

Here is an example of an unusually short (25 base pairs) Seq file:

```
ggtgaAGccctggagattcttatta
```

2.3 PSSM Set

Each *PSSM Set* consists of two files: *PSSM key* and *PSSM data*. The files should be named `XXX.key` and `XXX.data` accordingly, where `XXX` is the name of PSSM set. Both files must be placed into `pssms` directory.

2.3.1 PSSM Key

PSSM Key file contains description of PSSMs in the following format (columns are separated by tab character):

Column	Type	Description
ID	Integer	Unique ID (referenced in PSSM Data file)
Sequence	String	PSSM sequence
Biderectional	'0' or '1'	If Bidirectional is '1' energy is calculated on complementary strand as well

Example:

```
0    **************ATTAAT**************    1
1    *********A*ACACACACA*****A*******    1
2    **************AAAATGGC*G**********    1
3    **************ACTGCTTG************    1
4    ****WW**GTWGCATACTTTT*GGCG*******    1
```

```
5    *********C*RGCAACATKTTG**********    1
6    ****G*G*G*G*GAGCGAGA*RG**********    1
7    **************CCGAAG*************    1
```

2.3.2 PSSM Data

PSSM Data file contains probability matrices for each PSSM key in the following format (columns are separated by tab character):

Column	Type	Description
ID	Integer	Unique ID (must appear in PSSM Key file)
Position	Integer	Zero based position in the range of [0, length(PSSM sequence)-1]
Probability of 'A'	Numeric	Probability of 'A' in the range of [0, 1]
Probability of 'C'	Numeric	Probability of 'C' in the range of [0, 1]
Probability of 'G'	Numeric	Probability of 'G' in the range of [0, 1]
Probability of 'T'	Numeric	Probability of 'T' in the range of [0, 1]

3 Main Concepts

3.1 Intervals

3.1.1 1D Intervals

1D interval (or one-dimensional interval) represents a genomic section. It is defined by $(chrom, start, end)$ where *start* and *end* are genomic coordinates ($start < end$). The coordinates are zero-based, i.e. the chromosome starts at coordinate 0. The end coordinate marks the last coordinate in the section plus 1. To represent a point in the genome at coordinate X one should create an interval with start coordinate set to X and end coordinate set to $X + 1$.

3.1.2 2D Intervals

2D interval (or two-dimensional interval) represents a rectangle in a genomic space. It is defined by $(chrom_1, start_1, end_1, chrom_2, start_2, end_2)$, where $start_1, start_2, end_1$ and end_2 are start and end coordinates accordingly that mark the limits of a rectangle.

3.1.3 Intervals Sets

Multiple intervals can be combined into a table which is called *intervals set* or frequently simply referred as *intervals*. This table is represented by a data frame. In case of 1D intervals the data frame must have the first 3 columns named `chrom`, `start`, `end`. Likewise 2D intervals must have the first 6 columns named `chrom1`, `start1`, `end1`, `chrom2`, `start2`, `end2`.

Additional columns might be added to the intervals, some of them might be used by various functions. For instance, `gintervals.neighbors` function makes use of `strand` column if it is presented in 1D intervals (should come after the regular 3 columns). Use `gintervals` and `gintervals.2d` functions to create 1D and 2D intervals accordingly.

Both 1D and 2D intervals are widely used in various functions. Some of these functions manipulate the intervals (unify, intersect, ...). Others use the intervals to limit the scope on which the function acts. There are also functions that make their calculation for each interval in the intervals set.

3.1.4 Dual Intervals

Dual intervals is a list containing two elements. The first element is 1D intervals set, while the second element is 2D intervals set.

`ALLGENOME` variable is frequently used as a default value for intervals argument. `ALLGENOME` is an intervals set of dual type. `ALLGENOME[[1]]` represents a set of intervals that covers the whole genome (1D),

while `ALLGENOME[[2]]` contains all the possible pairs between the chromosomes (2D). One can also use `gintervals.all` and `gintervals.2d.all` functions to return all 1D or 2D intervals.

3.1.5 Serializing Intervals, Big and Small Intervals Sets

Intervals sets can be saved in Genomic Database. Use `gintervals.save` and `gintervals.load` functions to save or load an intervals set from the database and `gintervals.update` to update / add / delete a certain chromosome from the set.

Internally intervals sets can be stored in two different formats: *small intervals set* or *big intervals set*. The specific format is chosen depending on the size of the intervals set. Big format is selected for intervals sets that contain more than *gbig.intervals.size* intervals (`gbig.intervals.size` is set via `options`), wherever smaller sets are consequently stored in a small format. Use `gintervals.is.bigset` to determine the format of the stored intervals set.

Saved intervals sets in small format can be seamlessly used in all functions and track expressions without the need to explicitly load them.

```
# 'annotations' is an intervals set saved in Genomic Database
> gintervals.intersect("annotations", gintervals(2))
  chrom start   end
1  chr2    20  2000
2  chr2  3000  8000
3  chr2  9000 11000
```

Likewise big intervals sets can be used in many but not all the functions. The notable exception is `gintervals.load` that allows to load only a single chromosome (or a chromosome pair for 2D cases) of a big intervals set.

3.2 Tracks

Track is a data structure that allows to bind numeric data (floating point values) to genomic space (a set of genomic intervals). The data in the tracks can be typically accessed through *track expressions* that are widely used by various functions of the package.

Two fundamental types of tracks exist: *1D* and *2D*.

3.2.1 1D Track

1D track (or one-dimensional track) maps numeric values $V_0, ..., V_n$ to non-overlapping 1D intervals. Two formats of 1D tracks are supported by the package: *Dense* (sometimes also referred as *Fixed Bin*) and *Sparse*.

For a Dense track the size of the genomic interval is always fixed and called *bin size*. Numeric values are stored for all genomic intervals that cover the genome, however some of the values are allowed to be NaN. Dense track file can be seen as a contiguous chunk of values $V_0, ..., V_n$, where V_i is mapped to an interval $[binsize * i, binsize * (i + 1))$. Dense track's files do not store intervals' coordinates - which allow them to represent large amount of numeric data in a compact way. The size of a Dense track is inversely proportional to the bin size. The complexity of random access to a value at given coordinate is constant, i.e. $O(1)$.

Sparse tracks allow higher degree of freedom vs. Dense tracks. Each numeric value can be mapped to a genomic interval of an arbitrary size. The size of a Sparse track is proportional to the number of numeric values (not including NaNs). On the "cons" side the complexity of random access to a value at given coordinate is $O(logN)$, where N is the number of values in the track.

To sum up the differences between Dense and Sparse tracks please refer the following table:

	Dense	Sparse
Optimal use case	Data covering nearly the whole genome	Data covering a limited portion of a genome
Values stored	Per bin (interval of a fixed size)	Per interval of an arbitrary size
Random access complexity	$O(1)$	$O(logN)$
Disk usage	4 bytes per bin	20 bytes per value

1D tracks can be created by variety of functions such as: `gtrack.create`, `gtrack.create_sparse`, `gtrack.import_set` and more.

3.2.2 Array Track

Array track is similar to Sparse track in a way that it maps data to one-dimensional intervals of an arbitrary size. Yet unlike Sparse track an Array track can map more than one value into each interval. Array tracks allow thus to store large amount of data in one track - a task that would otherwise require maintenance of numerious number of tracks.

The values of an Array track are organized in *columns* each having a name and an index. One can see it as an NxM table where N is the number of intervals and M is the number of columns. The size of an Array track is proportional to the number of total numeric values stored inside (not including NaNs).

Attractive as they are Array tracks should not be abused and serve as a replacement of a Dense or Sparse track. A single Sparse track will always be more compact and efficient than an Array track holding a single column.

Array tracks are created by `gtrack.array.import` function.

3.2.3 2D Track

2D track (or two-dimensional track) maps numeric values $V_0, ..., V_n$ to non-overlapping 2D intervals. A typical use of a 2D track is to represent interaction between different parts of the genome.

2D tracks are internally stored in *chunks*, each chunk containing multiple track values. When a track value is accessed, the whole chunk containing it must be loaded into memory. The size of a chunk in bytes is controlled by `gtrack.chunk.size` option and typically it represents a tradeoff between the optimal access to a single value (a small chunk) and an access to multiple values (a large chunk).

During the access to multiple track values a few chunks can be invloved and loaded into memory. Since 2D tracks can potentially be huge one can limit the total number of chunks simultaneously stored in the memory by setting `gtrack.num.chunks` parameter.

2D tracks usually come in *Rectangles* format. A more space-efficient *Points* format also exists and it behaves similarly to Rectangles. *Computed* format is also supported though it is not covered by this document.

Rectangles track can be created by `gtrack.create`, `gtrack.2d.create`.

Points track is created by `gtrack.2d.import_contacts`.

3.2.4 Track as an Intervals Set

Since tracks represent a set of intervals (plus values) they are allowed to be used in various functions such as `gextract`, `gintervals.neighbors`, `gintervals.chrom_sizes` as a substitute for intervals sets. *Dense* tracks are the only exception to this rule and they cannot substitute intervals sets.

3.2.5 Track Attributes

In addition to numeric data a track may store arbitrary meta-data such as description, source, etc. The meta-data is stored in the form of name-value pairs or attributes where the value is a character string. All tracks created by `gtrack.create`, `gtrack.smooth` and other functions automatically add `created.by`, `created.date` and `description` attributes.

Though not officially enforced attributes are intended to store relatively short (but not empty) character strings. Please use *track variables* to store data in any other format.

A single attribute can be retrieved, added, modified or deleted using `gtrack.attr.get` and `gtrack.attr.set` functions. Bulk access and modification is available through `gtrack.attr.export` and `gtrack.attr.import` functions. Track names whose attributes match a pattern can be retrieved using `gtrack.ls` function.

Attribute can be defined as read-only which will prevent it from being modified or deleted. By default `created.by` and `created.date` attributes are read-only. Use `gdb.get_readonly_attrs`, `gdb.set_readonly_attrs` functions to retrieve or set the list of read-only attributes.

3.2.6 Track Variables

Track statistics, results of time-consuming per-track calculations, historical data and any other data in arbitrary format can be stored in a track's supplementary data in the form of track variables. Track variable can be retrieved, added, modified or deleted using *gtrack.var.get*, *gtrack.var.set*, *gtrack.var.rm* functions. List of track variables can be retrieved using *gtrack.var.ls* function.

3.2.7 Track Attributes vs. Track Variables

Though both track attributes and track variables can be used to store meta-data of a track, there are a few important differences between the two that are summed up in the following table:

	Track Attributes	Track Variables
Optimal use case	Track properties as short, non-empty character strings (description, source, ...)	Arbitrary data associated with the track
Value type	Character string	Arbitrary
Single value retrieval	`gtrack.attr.get`	`gtrack.var.get`
Single value modification	`gtrack.attr.set`	`gtrack.var.set`
Bulk value retrieval	`gtrack.attr.export`	—
Bulk value modification	`gtrack.attr.import`	—
Object names retrieval	`gtrack.attr.import`	`gtrack.var.ls`
Object removal	`gtrack.attr.set` with an empty string	`gtrack.var.rm`
Search by value	`gtrack.ls`	—

3.3 Track Expressions

3.3.1 Introduction

Track expression is a key concept of the package. Track expressions are widely used in various functions (`gscreen`, `gextract`, `gdist`, ...).

Track expression is a character string that closely resembles a valid R expression. Just like any other R expression it may include conditions, functions and variables defined beforehand. `"1 > 2"`, `"mean(1:10)"` and `"myvar < 17"` are all valid track expressions. Unlike regular R expressions track expression might also contain track names or *virtual track* names.

How does a track expression get evaluated? A track expression is accompanied by an *iterator* that determines a set of intervals over which the expression iterator goes. For each each iterator interval the track expression is evaluated. The value of a track expression `"mean(1:10)"` is constant regardless the iterator interval. However suppose the track expression contains a track name `mytrack`, like: `"mytrack * 3"`, and the whole story becomes very different. The library first recognizes that `mytrack` is not a regular R variable but rather a track name. A new R variable named `mytrack` is added then to R environment. For each iterator interval this variable is assigned the corresponding value of the track. This value obviously depends on the iterator interval. Once `mytrack` is assigned the corresponding value, the track expression is evaluated in R.

So how exactly the value of `mytrack` variable is determined given the iterator interval? We will demonstrate the answer by the following example. Suppose the track `mytrack` is in sparse format. It consists of a single chromosome with the following values:

chrom	start	end	value
chr1	100	200	10
chr1	200	250	25
chr1	500	560	17
chr1	600	700	44

What would be the value of the variable `mytrack` given an iterator interval? The resulted value is an average of all values of track `mytrack` covered by the iterator interval. For example, if the iterator interval is `[230, 620)` then the resulted value is an average of values 25, 17 and 44. Similarly if the iterator interval is `[0, 300)` then the resulted value is an average of 10 and 25. Lastly if the iterator intervals is `[300, 400)` then the resulted value is NaN. Same evaluation logics is applied for Dense and Array tracks. (In the latter case the values from all columns are averaged.) On contrary Rectangles track value is calculated as a *weighted* average of the values covered by the iterator interval. The weight equals to the intersection area of the iterator interval and the 2D interval that contains the value.

See the table below:

Track Type	Value
Dense	Average of non NaN values covered by iterator interval.
Sparse	Average of non NaN values covered by iterator interval.
Array	Average of non NaN values from all columns covered by iterator interval.
Rectangles	Weighted average of non NaN values covered by iterator interval. Each weight equals to the intersection area between iterator interval and track interval that contains the value.

3.3.2 Virtual Tracks

So far we showed that the value of a `mytrack` variable is set to be the average (or weighted average) of the track values that are covered by the iterator interval. But what if we do not want to average the values but rather pick up the maximal or minimal values? What if we want to use the percentile of a track value rather than the value itself? And maybe we even want to alter the iterator interval itself on the fly? This is where virtual tracks become useful.

Virtual track is a set of rules that describe how the "source" (a real track or intervals) should be proceeded, and how the iterator interval should be modified. Virtual tracks are created with `gvtrack.create` function:

```
> gvtrack.create("myvtrack", "dense_track")
```

This call creates a new virtual track named `myvtrack`. This virtual track can be used in the track expression instead of a real track `dense_track`. In our example `myvtrack` is just an alias of `dense_track`. Yet we can go on and create a more complicated virtual track if we specify a "function", i.e. instruct the virtual track of what should be its value in track expression.

```
> gvtrack.create("myvtrack", "dense_track", "global.percentile")
```

In this example when `myvtrack` is evaluated in the track expression it will return the percentile of V_{avg} among the values of `dense_track` where V_{avg} is an average (or weighted average) of the track values that are covered by the iterator interval.

Virtual tracks are especially useful for Array tracks. By default if an Array track is used in a track expressions, its interval value would be the average of all non-NaN column values covered by an iterator interval. `gvtrack.array.slice` allows to select specific columns and to specify the function applied to the values of each track interval.

```
> gvtrack.create("myvtrack", "array_track", "sum")
> gvtrack.array.slice("myvtrack", c("col2", "col5"), "max")
```

In this example we create a virtual track based on `array_track`. Assume that an iterator interval I covers n different intervals in `array_track`: $I_0, ..., I_n$. The value of `myvtrack` in a track expression would be then:

$$\sum_{i=1}^{n} max(V_{i,2}, V_{i,5})$$

where $V_{i,j}$ is a value of the track in column j for interval I_i.

Virtual tracks allow also to alter the iterator interval "on the fly":

```
> gvtrack.iterator("myvtrack", sshift = -100, eshift = 200)
```

In this example we expand each iterator interval by adding -100 to its `start` coordinate and 200 to its `end` coordinate.

Similarly iterator modifiers can be defined for 2D intervals. Moreover iterator modifier can create a 1D interval from a 2D iterator interval by projecting one of its axes.

```
> gvtrack.create("myvtrack", "dense_track")
> gvtrack.iterator("myvtrack", dim = "2")
```

It is important to remember that iterator modifiers transform the iterator interval only for the given virtual tracks. Assume an iterator interval I and two virtual tracks V_0 and V_1. If I is a 2D interval than *band* rules are applied first to it. I is tranformed then to I_0 and I_1 according to the modification rules defined by the virtual tracks. Finally I_0 and I_1 are passed to V_0 and V_1 accordingly as the iterator intervals.

So far we have used a track `dense_track` as a "source" of a virtual track. We can also use intervals as a source. In this case the value of the virtual track will be some function that takes into account the "source" intervals and the current iterator interval.

```
> gvtrack.create("myvtrack", "annotations", "distance")
> intervs <- gscreen("dense_track > 0.45")
> gextract("myvtrack", ALLGENOME, iterator = intervs)
```

In this example `myvtrack` returns the minimal distance between intervals from an interval set `annotations` and the center of the current iterator interval from `intervs`.

For a full list of supported functions please see `gvtrack.create` and `gvtrack.array.slice` functions.

3.3.3 Administrating Virtual Tracks

As desribed in the previous chapter virtual tracks define a set of rules of how to access and proceed the values of the "source" object. The connection between the virtual track and the source object is done via "soft link", i.e. by name and not by reference. For example, a virtual track will continue to exist until explicitly removed by `gvtrack.rm` even if the physical track that it is pointing to is deleted or renamed.

Operations such as `gdb.init` and `gdir.cd` alter the list of available tracks and intervals sets. Since these objects are referenced by virtual tracks, these latter are always defined in the context of the current working directory in Genomic Database (not to be confused with shell's current working directory). Changing the current working directory using `gdb.init` or `gdir.cd` will also change the list of available virtual tracks.

Another issue to bare in mind is that unlike regular tracks whose data is stored on disk virtual tracks are non-persistent objects in current R environment. Their definition is stored in `GVTRACKS` R variable. In particular a virtual track named "vtrack" that was created within a context of "/home/user/trackdb" Genomic Database working directory would reside in `GVTRACKS[["/home/user/trackdb"]][["vtrack"]]`. One can also use `gvtrack.info` function that provides a more convenient access to virtual track definitons.

As the virtual tracks are stored in an R variable their behavior hence complies with the rules of other R variables: a virtual track defined by one user will not be seen by another one, virtual tracks might dissapear once R is relaunched, etc.

To preserve the definition of virtual tracks between the sessions one would need to save `GVTRACKS` variable on disk. The serialization of `GVTRACKS` is under user's responsibility. The standard suit of functions for saving / loading R variables can be used for that purpose.

Note that if `GVTRACKS` is loaded from a file or changed manually by a user the *auto-completion* list (in case it is turned on) might need to be refreshed by calling `gdb.reload`.

3.3.4 Track Expression Evaluation under Optimization

Previously we described how a track expression `"mytrack * 3"` (where `mytrack` is a track name) leads to an implicit definition of `mytrack` variable in R environment. To make our explanation easier we presented this variable as a scalar whose value is altered each time the iterator interval changes. It's time to admit that that was oversimplification. In reality the library defines `mytrack` variable as a vector (i.e. an array) and not as a single scalar. The vector is filled then with the corresponding values of the track. Finally the track expression is evaluated in R and the result is expected to be also a vector of the same size as `mytrack` vector. Working with vectors rather than single scalars reduces the number of evaluations within R and hence improves run-times.

The size of the vector is controlled via `gbuf.size` option. By default it equals to 1000. Altering this value (for instance setting it to 1) might significantly affect the run-time of various functions in the library. If you still wish to force the functions to define scalars rather than vectors, set `gbuf.size` to 1:

```
options(gbuf.size = 1)
```

One might wonder why should we care about the fact that `mytrack` is not a scalar but rather a vector? Indeed in many cases it does not really matter. For example `mytrack * 3` expression produces exactly the same results regardless whether `mytrack` is defined internally as a vector or as a scalar. This is due to the fact that the expression `V * 3` (`V` stands for a vector) results in each value of `V` being multiplied by 3.

Multiplication is a good example of "parallel" operation in R (works on each element in vector separatedly). On contrary some functions that accept a vector might return a scalar rather than a vector. Such is, for example, `min` function.

Let's look at the following track expression: `track1 + min(track1, track2)`. This expression was probably meant to produce a sum of `track1` track and a minimum value between `track1` and `track2` tracks for each iterator interval. However the library defines the variables `track1` and `track2` to be vectors of `gbuf.size` size (by default: 1000). `min` is not a "parallel" operation. Given two vectors of any size it returns a single scalar that is the minimal value of all values in both of the vectors. Therefore `track1 + min(track1, track2)` will be interpreted as `track1 + M`, where M is minimum of 2000 values (1000 values from `track1` track, and another 1000 - from `track2` track). We can hardly imagine that a user would have really meant this! Sadly enough the expression will be seamlessly evaluated and produce a valid, but meaningless result. The solution for our example is to use `pmin` rather than `min` function.

The library always verifies that the evaluation of the track expression produces a vector of the same size as the size of a track variable. In many cases this procedure is able to reveal faulty track expressions. Yet in more tricky examples like the one that we used before the library will not warn the user.

Make sure your track expressions work correctly on vectors!

3.3.5 Revealing Current Iterator Interval

During the evaluation of a track expression one can access a specially defined variable named `GITERATOR.INTERVALS`. This variable contains a set of iterator intervals for which the track expression is evaluated. `GITERATOR.INTERVALS` contains the same number of intervals as the size of `mytrack` vector from our previous example. The value of a track `mytrack` for an interval `i` is stored at `mytrack[i]`.

Note that some intervals in `GITERATOR.INTERVALS` might have a start coordinate equal to -1. Skip those intervals and the values of `mytrack` at the corresponding index.

3.3.6 Iterators

So far we have discussed in details how the track expression is evaluated given the *iterator interval*. Yet how the iterator intervals can be controlled?

Most of the functions that accept track expressions have an additional parameter named `iterator`. The value of this parameter determines the iterator intervals which is also sometimes called an *iterator policy*:

Value	Iterator Policy Type	Example	Description
Integer	Fixed Bin	`50`	Iterator intervals will advance by a fixed step (bin) starting from zero coordinate up to chromosome's length: `[0,50)`, `[50,100)`, `[100,150)`, `...`
Dense track	Fixed Bin	`"dense_track"`	Use the bin size of the track as a fixed step.
1D intervals	1D Intervals	`"annotations"`	Iterate over the supplied intervals. *Note: the intervals are sorted and overlapping intervals are unified.*
Sparse track	1D Intervals	`"sparse_track"`	Iterate over the intervals of a sparse track.
Array track	1D Intervals	`"array_track"`	Iterate over the intervals of an array track.
c(integer, integer)	2D Intervals	`c(1000, 2000)`	2D iterator intervals will cover the whole 2D chromosomal space by rectangles of fixed size: Width X Height. Please keep in mind that small rectangles used without a limiting scope might result in immense number of iterator intervals.
2D intervals	2D Intervals	`gintervals.2d(c(1, 2))`	Iterate over the supplied intervals. *Note: the intervals are sorted and overlapping is forbidden.*
Rectangles track	2D Intervals	`"rects_track"`	Iterate over the intervals of a Rectangles track
Cartesian grid iterator	2D Intervals	`giterator.cartesian_grid( intervals1, intervals2, c(10, 20, 30))`	Iterate over 2D cartesian grid (see `giterator.cartesian_grid` function)
NULL	Fixed Bin OR 1D Intervals OR 2D Intervals	NULL	Implicitly determine the iterator policy based on the tracks that appear in the track expression. If no track names presented or two different tracks determine different iterator policy, an error is reported.

3.3.7 Scope

Many functions that accept a track expressions and iterator policy accept an additional set of intervals that limit the scope of a function. This scope also limits the iterator intervals. For instance:

```
> gextract("dense_track", gintervals(2, 340, 520))
  chrom start end    dense_track intervalID
1  chr2   340 350           0.14          1
2  chr2   350 400           0.08          1
3  chr2   400 450           0.16          1
4  chr2   450 500           0.00          1
5  chr2   500 520           0.16          1
```

As one can notice the first and the last intervals in the result are truncated by the scope `[340, 520)`.

In some cases the combination of iterator policy and scope might result in nontrivial set of iterator intervals. Use `giterator.intervals` function to retrieve the iterator intervals given a track expression, scope and an iterator.

3.3.8 Band

As explained before track expression iterator can be determined implicitly or through an `iterator` parameter. In either case the result is a set of 1D or 2D intervals depending on how the iterator was defined. If iterator intervals are 2D an additional filter can be applied to them: a *band*.

A band is a pair of integers: D_1, D_2. We say that a 2D iterator interval $(chrom_1, x_1, x_2, chrom_2, y_1, y_2)$ intersects a band if and only if the next two conditions are true:

1. $chrom_1 = chrom_2$
2. $\exists x, y : x_1 \leq x < x_2 \wedge y_1 \leq y < y_2 \wedge D_1 \leq x - y < D_2$.

In a less formal way we can see a band as a space S between two 45-degrees diagonals where $D1, D2$ determine where these diagonals cross X axis. An iterator interval represents a rectangle in a 2D space and can be therefore intersected with S. The result of the intersection can be a rectangle, a trapeze, a triangle, a hexagon or it can be empty if the interval does not intersect with the band. If the intersection is non empty, the resulted figure, whatever it is, can be bound by some larger rectangle. The rectangle that has the minimal space and yet containing the intersected shape is called *the minimal rectangle*.

After the formal definitions it's time to say how band is actually applied. If the intersection between the 2D iterator interval and the band is non-empty and $chrom_1 = chrom_2$, the minimal rectangle replaces the original iterator interval. Otherwise the iterator interval is skipped as it lies outside of the band or the two chromosomes are not equal.

`gintervals.2d.band_intersect` function can help one better understand the concept:

```
> intervs <- gintervals.2d(1, 200, 800, 1, 100, 1000)
> intervs <- rbind(intervs, gintervals.2d(1, 900, 950, 1, 0, 200))
> intervs <- rbind(intervs, gintervals.2d(1, 0, 100, 1, 0, 400))
> intervs <- rbind(intervs, gintervals.2d(1, 900, 950, 2, 0, 200))
> intervs
  chrom1 start1 end1 chrom2 start2 end2
1   chr1    200  800   chr1    100 1000
2   chr1    900  950   chr1      0  200
3   chr1      0  100   chr1      0  400
4   chr1    900  950   chr2      0  200
> gintervals.2d.band_intersect(intervs, band = c(500, 1000))
  chrom1 start1 end1 chrom2 start2 end2
1   chr1    600  800   chr1    100  300
2   chr1    900  950   chr1      0  200
```

`gintervals.2d.band_intersect` intersects the intervals with the band and returns the intervals shrunk to the minimal rectangle. As you can see we have four different intervals. The first one `(chr1, 200, 800, chr1, 100, 1000)` intersects the band and after shrinking to the minimal rectangle it becomes `(chr1, 600, 800, chr1, 100, 300)`. The second interval lies entirely within the band and hence is returned without any change. The third interval lies entirely outside of the band, and hence is eliminated from the result. The last interval is coming from two different chromosomes and therefore is also filtered out.

As said band filters out and alters 2D iterator intervals. Yet it also affects the result of 2D tracks. Let's look at the following example:

```
> intervs <- gintervals.2d(1, c(100, 400), c(300, 490), 1, c(120, 180), c(200, 500))
> gtrack.2d.create("test2d", intervs, c(10, 20))
> gextract("test2d", ALLGENOME)
  chrom1 start1 end1 chrom2 start2 end2 test2d intervalID
1   chr1    100  300   chr1    120  200     10          1
2   chr1    400  490   chr1    180  500     20          1
> gextract("test2d", ALLGENOME, iterator = gintervals.2d(1, 0, 1000, 1, 0, 1000))
  chrom1 start1 end1 chrom2 start2 end2   test2d intervalID
1   chr1      0 1000   chr1      0 1000 16.42857          1
> gintervals.2d.band_intersect(intervs, band = c(150, 1000))
  chrom1 start1 end1 chrom2 start2 end2
1   chr1    270  300   chr1    120  150
2   chr1    400  490   chr1    180  340
> gextract("test2d", ALLGENOME, iterator = gintervals.2d(1, 0, 1000, 1, 0, 1000),
  band = c(150, 1000))
  chrom1 start1 end1 chrom2 start2 end2   test2d intervalID
1   chr1    150 1000   chr1      0  850 19.57182          1
> gtrack.rm("test2d", force = TRUE)
```

We created a 2D track `test2d` and inserted two values into it: 10 and 20. If an iterator interval covers all the track's rectangles, the resulted value of the track would be a weighted average of its values where the weight is equal to the intersected area. In our example it is 16.42857.

We added a band then. `gintervals.2d.band_intersect` shows the minimal rectangles: the intersection result of the original rectangles with the band. The output of the new `gextract` has been changed accordingly: the new weights in the weighted average are equal to the new and smaller intersected area. The value has changed therefore to: 19.57182.

Note, however, that the space used in the calculation of the weighted average is the actual space of the intersection and not the space occupied by the minimal rectangles!

4 Input Mode and Auto-Completion

By default track expressions, track names, virtual tracks and interval sets are passed to the functions as character strings. Being good for scripts, this mode is however less appropriate for interactive work in R where user might miss the ability to use auto-completion of the object names with a TAB key - in a way similar to how R variables and functions are auto-completed.

`gset_input_mode` allows the user to pass track expressions, track names, virtual tracks and interval sets unquoted, i.e. to use them as if they were valid R variables and expressions. In this "unquoted" (or "interactive") mode all the track names, virtual tracks and intervals sets are indeed defined as R variables (auxiliary variables) which allows them to be auto-completed by TAB. The values of these variables are meaningless for the user and they should not be altered.

```
> gset_input_mode(interactive = FALSE)   # this is the default mode
> gsummary("dense_track+10")
> gset_input_mode(interactive = TRUE)
> gsummary(dense_track+10)
```

Please beware of the consequences of using interactive mode as it creates a bunch of new variables in R environment. Though collision with the existing variables is checked at the time of the call to `gset_input_mode`, yet nothing prevents the user to modify the value of the auxiliary variables later. This might cause unexpected behaviour in some of the package functions. Also the auxiliary variables are automatically undefined once the interactive mode is switched off. User who mistakenly uses auxiliary variables to store the data might therefore accidentially loose it.

5 Random Algorithms

Various functions in the library such as `gsample` make use of pseudo-random number generator. Each time the function is invoked a unique series of random numbers is issued. Hence two identical calls might produce different results. To guarantee reproducible results call `set.seed` before invoking the function.

```
> set.seed(1)
> r1 <- gsample("dense_track", 10)
> r2 <- gsample("dense_track", 10)  # r2 differs from r1
> set.seed(1)
> r3 <- gsample("dense_track", 10)  # r3 == r1
```

6 Multitasking

6.1 Controlling the Number of Processes

To boost the run time performance various functions in the library support multitasking mode, i.e. parallel computation of the result by several concurrent processes. The exact number of processes internally launched depends on the specific call however the upper bound can be controlled by a few parameters such as `gmax.processes` (absolute upper bound), `gmax.processes2core` (maximal number of processes per CPU core) and `gmin.scope4process` (minimal scope range / surface assigned to a process). Multitasking can also be completely switched off by setting `gmultitasking` parameter to `FALSE`.

6.2 Limiting the Memory Consumption

For certain functions multitasking might result in higher memory consumption. Users who have per process virtual memory limit (see: `ulimit -v`) might be the first to suffer from memory allocation errors.

Various factors can affect the memory usage such as the number of running processes used for parallel computation, the value of `gmax.data.size` option or the combination of both. Some of the functions such as `gscreen` or `gextract` consume in multitasking mode amount of memory proportional to `gmax.data.size`. Please be aware of it while altering the value of this option.

To limit memory consumption in multitasking mode one might lower down the values of `gmax.data.size` and `gmax.mem.usage` options or even switch off multitasking mode completely. `gmax.mem.usage` indicates the upper limit in KB of memory consumed cumulatively by the child processes. Once this limit is breached an internal mechanism tries to pause some of the running child processes, thereby preventing them from allocating more memory. The paused processes are resumed once the memory consumption drops or other sibling processes end.

One should not expect the internal limiting mechanism to be the panacea for memory hungry tasks. First, the memory consumption of some of the functions is proportional to `gmax.data.size` option regardless of the number of running processes. Second, even when the memory limit is exceeded at least one process is still left to run and to potentially increase the memory consumption further. Third, the mechanism is mainly periodic, i.e. excessive memory consumption is detected only once in a while. The decision to pause running processes is thus periodic as well. The memory that has already been consumed in the time gap between the checks will not be release up until the whole task is complete.

It is worth to say a word about memory consumption. Deducting real memory usage of the process based on "top", "ps" or other utilities of similar kind might be highly misleading. Since all the processes are spawned from R, their memory usage as reported by these utilities will be at least as high as that of their

parent process. If, for example, R process uses 5 Gb of memory and 10 processes are spawned from it, the virtual memory of all these 11 processes will top 55 Gb. Yet the majority of the consumed memory will be shared and unless the child processes start modifying this memory or allocating new one, the physical free memory of the machine will remain almost unaltered. The internal memory consumption limiting mechanism tries to estimate the drop of system free memory and hence deducts its data from counting "Private Dirty" bytes (on Linux) or from internal estimation (on other platforms) - a very different datum from what "top" is reporting.

6.3 Other Considerations

In multitasking mode the return value of `gquantiles` may vary depending on the number of CPU cores. For more details please refer the documentation of this function.