Package: misha (via r-universe)

October 23, 2026

Type Package

Title Toolkit for Analysis of Genomic Data

Version 5.11.23

Description A toolkit for analysis of genomic data. The 'misha' package implements an efficient data structure for storing genomic data, and provides a set of functions for data extraction, manipulation and analysis. Some of the 2D genome algorithms were described in Yaffe and Tanay (2011) <doi:10.1038/ng.947>.

License MIT + file LICENSE

URL https://tanaylab.github.io/misha/, https://github.com/tanaylab/misha

BugReports https://github.com/tanaylab/misha/issues

Depends R (>= 3.0.0)

Imports magrittr, curl, digest, ps, parallel, utils, tools, yaml

SystemRequirements samtools (>= 1.0; optional, required only for BAM input), bedGraphToBigWig or wigToBigWig from UCSC (optional, required only for bigWig export; available as the bioconda package ucsc-bedgraphtobigwig)

Suggests data.table, dplyr, glue, knitr, readr, rmarkdown, spelling, stats, stringr, testthat (>= 3.0.0), tibble, withr

Config/testthat/edition 3

Config/testthat/start-first liftover, multifasta-import

Encoding UTF-8

Language en-US

LazyLoad yes

NeedsCompilation yes

OS_type unix

VignetteBuilder knitr, rmarkdown

Config/roxygen2/version 8.0.0

Author Misha Hoichman [aut], Aviezer Lifshitz [aut, cre], Eitan Yaffe [aut], Amos Tanay [aut], Weizmann Institute of Science [cph]

Maintainer Aviezer Lifshitz <aviezer.lifshitz@weizmann.ac.il>

Repository https://cran.r-universe.dev

Date/Publication 2026-08-24 13:30:02 UTC

RemoteUrl https://github.com/cran/misha

RemoteRef HEAD

RemoteSha edadf70f2d40028215a3b5097d236db1f559ff1b

Contents

misha-package 6
gbins.quantiles 7
gbins.summary 9
gcis_decay 10
gcluster.run 12
gcompute_strands_autocorr 13
gcor 15
gdataset.example_path 16
gdataset.info 17
gdataset.load 18
gdataset.ls 19
gdataset.save 19
gdataset.unload 20
gdb.build_genome 21
gdb.convert_to_indexed 23
gdb.create 26
gdb.create_genome 28
gdb.create_linked 29
gdb.export_fasta 30
gdb.genome_info 31
gdb.get_readonly_attrs 32
gdb.info 32
gdb.init 33
gdb.init_examples 34
gdb.install_gff3_converter 35
gdb.install_gtf_converter 36
gdb.install_intervals 37
gdb.list_genomes 39
gdb.mark_cache_dirty 40
gdb.reload 41
gdb.set_readonly_attrs 41
gdb.unload 42
gdir.cd 43

gdir.create . . . 44
gdir.cwd . . . 44
gdir.rm . . . 45
gdist . . . 46
gextract . . . 48
ggenome.implant . . . 50
ggenome.transplant . . . 52
gintervals . . . 54
gintervals.2d . . . 55
gintervals.2d.all . . . 56
gintervals.2d.band_intersect . . . 57
gintervals.2d.convert_to_indexed . . . 58
gintervals.2d.intersect . . . 59
gintervals.2d.union . . . 60
gintervals.all . . . 61
gintervals.annotate . . . 61
gintervals.as_chain . . . 64
gintervals.attr.export . . . 65
gintervals.attr.get . . . 66
gintervals.attr.import . . . 67
gintervals.attr.set . . . 68
gintervals.canonic . . . 69
gintervals.chrom_sizes . . . 71
gintervals.convert_to_indexed . . . 72
gintervals.coverage_fraction . . . 73
gintervals.covered_bp . . . 74
gintervals.dataset . . . 75
gintervals.dbs . . . 76
gintervals.diff . . . 77
gintervals.exists . . . 78
gintervals.force_range . . . 78
gintervals.from_mat . . . 79
gintervals.from_strings . . . 80
gintervals.import_bed . . . 81
gintervals.import_genes . . . 82
gintervals.import_gff . . . 83
gintervals.import_vcf . . . 84
gintervals.intersect . . . 85
gintervals.is.bigset . . . 86
gintervals.liftover . . . 86
gintervals.load . . . 89
gintervals.load_chain . . . 90
gintervals.ls . . . 92
gintervals.mapply . . . 93
gintervals.mark_overlaps . . . 95
gintervals.neighbors . . . 96
gintervals.neighbors.upstream . . . 99
gintervals.normalize . . . 101

gintervals.path . . . 102
gintervals.quantiles . . . 103
gintervals.random . . . 104
gintervals.rbind . . . 106
gintervals.rm . . . 107
gintervals.save . . . 108
gintervals.summary . . . 109
gintervals.to_mat . . . 110
gintervals.union . . . 111
gintervals.update . . . 112
giterator.cartesian_grid . . . 113
giterator.intervals . . . 115
glookup . . . 117
gpartition . . . 119
gquantiles . . . 120
grevcomp . . . 121
gsample . . . 122
gscreen . . . 123
gsegment . . . 124
gseq.comp . . . 126
gseq.extract . . . 127
gseq.kmer . . . 128
gseq.kmer.dist . . . 130
gseq.pwm . . . 131
gseq.pwm_edits . . . 134
gseq.read_homer . . . 136
gseq.read_jaspar . . . 137
gseq.read_meme . . . 138
gseq.rev . . . 139
gseq.revcomp . . . 140
gsummary . . . 140
gsynth.bin_map . . . 142
gsynth.cell_merge . . . 143
gsynth.convert . . . 144
gsynth.forbid_kmer . . . 145
gsynth.load . . . 146
gsynth.random . . . 147
gsynth.replace_kmer . . . 149
gsynth.sample . . . 150
gsynth.save . . . 153
gsynth.score . . . 154
gsynth.train . . . 155
gtrack.2d.convert_to_indexed . . . 158
gtrack.2d.create . . . 159
gtrack.2d.import . . . 160
gtrack.2d.import_contacts . . . 161
gtrack.array.extract . . . 163
gtrack.array.get_colnames . . . 164

gtrack.array.import . . . 165
gtrack.array.set_colnames . . . 166
gtrack.attr.export . . . 167
gtrack.attr.get . . . 168
gtrack.attr.import . . . 169
gtrack.attr.set . . . 170
gtrack.convert . . . 171
gtrack.convert_to_indexed . . . 172
gtrack.copy . . . 173
gtrack.create . . . 174
gtrack.create_dense . . . 175
gtrack.create_dirs . . . 177
gtrack.create_pwm_energy . . . 177
gtrack.create_sparse . . . 178
gtrack.dataset . . . 180
gtrack.dbs . . . 181
gtrack.exists . . . 182
gtrack.export_bedgraph . . . 182
gtrack.export_bigwig . . . 184
gtrack.import . . . 185
gtrack.import_mappedseq . . . 187
gtrack.import_set . . . 189
gtrack.info . . . 190
gtrack.liftover . . . 191
gtrack.lookup . . . 193
gtrack.ls . . . 195
gtrack.modify . . . 196
gtrack.mv . . . 197
gtrack.path . . . 198
gtrack.rm . . . 199
gtrack.smooth . . . 200
gtrack.var.get . . . 201
gtrack.var.ls . . . 202
gtrack.var.rm . . . 203
gtrack.var.set . . . 204
gvtrack.array.slice . . . 205
gvtrack.clear . . . 206
gvtrack.create . . . 207
gvtrack.filter . . . 218
gvtrack.info . . . 220
gvtrack.iterator . . . 221
gvtrack.iterator.2d . . . 222
gvtrack.ls . . . 223
gvtrack.rm . . . 224
gwget . . . 225
gwilcox . . . 226
print.gsynth.model . . . 227

Index **228**

misha-package *Toolkit for analysis of genomic data*

Description

'misha' package is intended to help users to efficiently analyze genomic data achieved from various experiments.

Details

For a complete list of help resources, use `library(help = "misha")`.

The following options are available for the package. Use 'options' function to alter the value of the options.

NAME	DEFAULT	DESCRIPTION
gmax.data.size	AUTO	Auto-configured based on system RAM and processes. Formula: min((RAM * 0.7) / gmax.processes, 10GB). Controls max in-memory result buffer size for streaming/sampling. Operations like gextract, gscreen use this as upper bound.
gbig.intervals.size	1000000	Threshold for converting interval sets to disk-based "big" format. When interval count exceeds this, intervals are stored per-chromosome instead of all in memory. Note: Independent of gmax.data.size (different purposes).
gmax.mem.usage	10000000	Maximal memory consumption of all child processes in KB before the limiting algorithm is invoked.
gmax.processes	AUTO	Auto-configured to min(32, 70% of CPU cores). Maximal number of processes for multitasking.
gmax.processes2core	2	Maximal number of processes per CPU core for multitasking
gseq.extract.probe.usec	100	gseq.extract times its first reads and distributes across processes when the measured cost per interval reaches this many microseconds. 0 always distributes, a large value never does.
gmin.scope4process	10000	Minimal scope range (for 2D: surface) assigned to a process in multitasking mode.
gbuf.size	1000	Size of track expression values buffer.
gmultitask.max.records.factor	64	Multiplier to inflate multitask max_records estimates to avoid under-allocation.
gtrack.chunk.size	100000	Chunk size in bytes of a 2D track. If '0' chunk size is unlimited.
gtrack.num.chunks	0	Maximal number of 2D track chunks simultaneously stored in memory.
gmultitasking	TRUE	Enable/disable automatic parallelization. Some functions may choose single-threaded mode for very small workloads.
gpermissions.umask	"0007"	umask applied around misha's own writes into a database: 660 files, 770 directories - group-writable, no world access - matching what the C++ layer enforces. Set to

"0002" for the world-readable 664/775 that misha produced before 5.11.21, or to NULL to leave the process umask untouched. The umask is restored as soon as the write returns.

More information about the options can be found in 'User manual' of the package.

Author(s)

Maintainer: Aviezer Lifshitz <aviezer.lifshitz@weizmann.ac.il>

Authors:

- Aviezer Lifshitz <aviezer.lifshitz@weizmann.ac.il>
- Misha Hoichman <misha@hoichman.com>
- Eitan Yaffe <eitan.yaffe@weizmann.ac.il>
- Amos Tanay <amos.tanay@weizmann.ac.il>

Other contributors:

- Weizmann Institute of Science [copyright holder]

See Also

Useful links:

- https://tanaylab.github.io/misha/
- https://github.com/tanaylab/misha
- Report bugs at https://github.com/tanaylab/misha/issues

---

gbins.quantiles *Calculates quantiles of a track expression for bins*

Description

Calculates quantiles of a track expression for bins.

Usage

```
gbins.quantiles(
  ...,
  expr = NULL,
  percentiles = 0.5,
  intervals = get("ALLGENOME", envir = .misha),
  include.lowest = FALSE,
  iterator = NULL,
  band = NULL
)
```

Arguments

...	pairs of track expressions ('bin_expr') that determines the bins and breaks that define the bins. See gdist. An extra, unpaired trailing argument is taken as 'expr'; it cannot be combined with a named 'expr' argument - that combination is ambiguous and raises an error.
expr	track expression for which quantiles are calculated
percentiles	an array of percentiles of quantiles in [0, 1] range
intervals	genomic scope for which the function is applied.
include.lowest	if 'TRUE', the lowest value of the range determined by breaks is included
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expressions.
band	track expression band. If 'NULL' no band is used.

Details

This function is a binned version of 'gquantiles'. For each iterator interval the value of 'bin_expr' is calculated and assigned to the corresponding bin determined by 'breaks'. The quantiles of 'expr' are calculated then separately for each bin.

The bins can be multi-dimensional depending on the number of 'bin_expr'-'breaks' pairs.

The range of bins is determined by 'breaks' argument. For example: 'breaks=c(x1, x2, x3, x4)' represents three different intervals (bins): (x1, x2], (x2, x3], (x3, x4].

If 'include.lowest' is 'TRUE' the the lowest value will be included in the first interval, i.e. in [x1, x2].

Value

Multi-dimensional array representing quantiles for each percentile and bin.

See Also

gquantiles, gintervals.quantiles, gdist

Examples

```
gdb.init_examples()
gbins.quantiles("dense_track", c(0, 0.2, 0.4, 2), "sparse_track",
    percentiles = c(0.2, 0.5),
    intervals = gintervals(1),
    iterator = "dense_track"
)
```

gbins.summary *Calculates summary statistics of a track expression for bins*

Description

Calculates summary statistics of a track expression for bins.

Usage

```
gbins.summary(
  ...,
  expr = NULL,
  intervals = get("ALLGENOME", envir = .misha),
  include.lowest = FALSE,
  iterator = NULL,
  band = NULL
)
```

Arguments

...	pairs of track expressions ('bin_expr') that determines the bins and breaks that define the bins. See gdist. An extra, unpaired trailing argument is taken as 'expr'; it cannot be combined with a named 'expr' argument - that combination is ambiguous and raises an error.
expr	track expression for which summary statistics is calculated
intervals	genomic scope for which the function is applied
include.lowest	if 'TRUE', the lowest value of the range determined by breaks is included
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expressions.
band	track expression band. If 'NULL' no band is used.

Details

This function is a binned version of 'gsummary'. For each iterator interval the value of 'bin_expr' is calculated and assigned to the corresponding bin determined by 'breaks'. The summary statistics of 'expr' are calculated then separately for each bin.

The bins can be multi-dimensional depending on the number of 'bin_expr'-'breaks' pairs.

The range of bins is determined by 'breaks' argument. For example: 'breaks=c(x1, x2, x3, x4)' represents three different intervals (bins): (x1, x2], (x2, x3], (x3, x4].

If 'include.lowest' is 'TRUE' the the lowest value will be included in the first interval, i.e. in [x1, x2].

Value

Multi-dimensional array representing summary statistics for each bin.

See Also

gsummary, gintervals.summary, gdist

Examples

```
gdb.init_examples()
gbins.summary("dense_track", c(0, 0.2, 0.4, 2), "sparse_track",
    intervals = gintervals(1), iterator = "dense_track"
)
```

---

gcis_decay *Calculates distribution of contact distances*

Description

Calculates distribution of contact distances.

Usage

```
gcis_decay(
  expr = NULL,
  breaks = NULL,
  src = NULL,
  domain = NULL,
  intervals = NULL,
  include.lowest = FALSE,
  iterator = NULL,
  band = NULL
)
```

Arguments

expr	track expression
breaks	breaks that determine the bin
src	source intervals
domain	domain intervals
intervals	genomic scope for which the function is applied
include.lowest	if 'TRUE', the lowest value of the range determined by breaks is included
iterator	2D track expression iterator. If 'NULL' iterator is determined implicitly based on track expressions.
band	track expression band. If 'NULL' no band is used.

Details

A 2D iterator interval '(chrom1, start1, end1, chrom2, start2, end2)' is said to represent a contact between two 1D intervals I1 and I2: '(chrom1, start1, end1)' and '(chrom2, start2, end2)'.

For contacts where 'chrom1' equals to 'chrom2' and I1 is within source intervals the function calculates the distribution of distances between I1 and I2. The distribution is calculated separately for intra-domain and inter-domain contacts.

An interval is within source intervals if the unification of all source intervals fully overlaps it. 'src' intervals are allowed to contain overlapping intervals.

Two intervals I1 and I2 are within the same domain (intra-domain contact) if among the domain intervals exists an interval that fully overlaps both I1 and I2. Otherwise the contact is considered to be inter-domain. 'domain' must contain only non-overlapping intervals.

The distance between I1 and I2 is the absolute distance between the centers of these intervals, i.e.: 'l(start1 + end1 - start2 - end2) / 2l'.

The range of distances for which the distribution is calculated is defined by 'breaks' argument. For example: 'breaks=c(x1, x2, x3, x4)' represents three different intervals (bins): (x1, x2], (x2, x3], (x3, x4].

If 'include.lowest' is 'TRUE' the the lowest value will be included in the first interval, i.e. in [x1, x2]

Value

2-dimensional vector representing the distribution of contact distances for inter and intra domains.

See Also

gdist, gtrack.2d.import_contacts

Examples

```
gdb.init_examples()

src <- rbind(
    gintervals(1, 10, 100),
    gintervals(1, 200, 300),
    gintervals(1, 400, 500),
    gintervals(1, 600, 700),
    gintervals(1, 7000, 9100),
    gintervals(1, 9000, 18000),
    gintervals(1, 30000, 31000),
    gintervals(2, 1130, 15000)
)

domain <- rbind(
    gintervals(1, 0, 483000),
    gintervals(2, 0, 300000)
)
```

```
gcis_decay("rects_track", 50000 * (1:10), src, domain)
```

`gcluster.run` *Runs R commands on a cluster*

Description

Runs R commands on a cluster that supports SGE.

Usage

```
gcluster.run(
  ...,
  opt.flags = "",
  max.jobs = 400,
  debug = FALSE,
  R = "R",
  control_dir = NULL
)
```

Arguments

`...`	R commands
`opt.flags`	optional flags for qsub command
`max.jobs`	maximal number of simultaneously submitted jobs
`debug`	if 'TRUE', additional reports are printed
`R`	command that launches R
`control_dir`	directory where the control files are stored. Note that this directory should be accessible from all nodes. If 'NULL', a temporary directory would be created under the current misha database.

Details

This function runs R commands on a cluster by distributing them among cluster nodes. It must run on a machine that supports Sun Grid Engine (SGE). The order in which the commands are executed can not be guaranteed, therefore the commands must be inter-independent.

Optional flags to 'qsub' command can be passed through 'opt.flags' parameter. Users are strongly recommended to use only '-l' flag as other flags might interfere with those that are already used (-terse, -S, -o, -e, -V). For additional information please refer to the manual of 'qsub'.

The maximal number of simultaneously submitted jobs is controlled by 'max.jobs'.

Set 'debug' argument to 'TRUE to allow additional report prints.

'gcluster.run' launches R on the cluster nodes to execute the commands. 'R' argument specifies how R executable should be invoked.

Value

Return value ('retv') is a list, such that 'retv[[i]]' represents the result of the run of command number 'i'. Each result consists of 4 fields that can be accessed by 'retv[[i]]$FIELDNAME':

FIELDNAME	*DESCRIPTION*
exit.status	Exit status of the command. Possible values: 'success', 'failure' or 'interrupted'.
retv	Return value of the command.
stdout	Standard output of the command.
stderr	Standard error of the command.

Examples

```
gdb.init_examples()
# Run only on systems with Sun Grid Engine (SGE)
if (FALSE) {
    v <- 17
    gcluster.run(
        gsummary("dense_track + v"),
        {
            intervs <- gscreen("dense_track > 0.1", gintervals(1, 2))
            gsummary("sparse_track", intervs)
        },
        gsummary("rects_track")
    )
}
```

`gcompute_strands_autocorr` *Computes auto-correlation between the strands for a file of mapped sequences*

Description

Calculates auto-correlation between plus and minus strands for the given chromosome in a file of mapped sequences.

Usage

```
gcompute_strands_autocorr(
  file = NULL,
  chrom = NULL,
  binsize = NULL,
  maxread = 400,
  cols.order = c(9, 11, 13, 14),
  min.coord = 0,
  max.coord = 3e+08
)
```

Arguments

file	the name of the file containing mapped sequences
chrom	chromosome for which the auto-correlation is computed
binsize	calculate the auto-correlation for bins in the range of [-maxread, maxread]
maxread	maximal length of the sequence used for statistics
cols.order	order of sequence, chromosome, coordinate and strand columns in file
min.coord	minimal coordinate used for statistics
max.coord	maximal coordinate used for statistics

Details

This function calculates auto-correlation between plus and minus strands for the given chromosome in a file of mapped sequences. Each line in the file describes one read. Each column is separated by a TAB character.

The following columns must be presented in the file: sequence, chromosome, coordinate and strand. The position of these columns are controlled by 'cols.order' argument accordingly. The default value of 'cols.order' is a vector (9,11,13,14) meaning that sequence is expected to be found at column number 9, chromosome - at column 11, coordinate - at column 13 and strand - at column 14. The first column should be referenced by 1 and not by 0.

Coordinates that are not in [min.coord, max.coord] range are ignored.

gcompute_strands_autocorr outputs the total statistics and the auto-correlation given by bins. The size of the bin is indicated by 'binsize' parameter. Statistics is calculated for bins in the range of [-maxread, maxread].

Value

Statistics for each strand and auto-correlation by given bins.

Examples

```
gdb.init_examples()
gcompute_strands_autocorr(paste(.misha$GROOT, "reads", sep = "/"),
    "chr1", 50,
    maxread = 300
)
```

gcor *Calculates correlation between track expressions*

Description

Calculates correlation between track expressions over iterator bins inside the supplied genomic scope. Expressions are processed in pairs: (expr1, expr2), (expr3, expr4), etc. Only bins where both expressions are not NaN are used.

Usage

```
gcor(
  expr1 = NULL,
  expr2 = NULL,
  ...,
  intervals = NULL,
  iterator = NULL,
  band = NULL,
  method = c("pearson", "spearman", "spearman.exact"),
  details = FALSE,
  names = NULL
)
```

Arguments

expr1	first track expression
expr2	second track expression
...	additional track expressions, supplied as pairs (expr3, expr4, ...)
intervals	genomic scope for which the function is applied
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expression.
band	track expression band. If 'NULL' no band is used.
method	correlation method to use. One of 'pearson' (default), 'spearman' (approximate, memory-efficient), or 'spearman.exact' (exact, requires O(n) memory where n is number of non-NaN pairs).
details	if 'TRUE' returns summary statistics for each pair, otherwise returns correlations only. For Pearson, includes n, n.na, mean1, mean2, sd1, sd2, cov, cor. For Spearman methods, includes n, n.na, cor.
names	optional names for the pairs. If supplied, length must match the number of pairs.

Value

If 'details' is 'FALSE', a numeric vector of correlations. If 'details' is 'TRUE', a data frame with summary statistics for each pair.

See Also

gextract, gscreen, gsummary

Examples

```
gdb.init_examples()
gcor("dense_track", "sparse_track", intervals = gintervals(1, 0, 10000), iterator = 1000)

# Spearman correlation (approximate, memory-efficient)
gcor("dense_track", "sparse_track",
    intervals = gintervals(1, 0, 10000),
    iterator = 1000, method = "spearman"
)

# Exact Spearman correlation
gcor("dense_track", "sparse_track",
    intervals = gintervals(1, 0, 10000),
    iterator = 1000, method = "spearman.exact"
)
```

---

gdataset.example_path *Create an example dataset on the fly*

Description

Creates a small dataset in a temporary directory using the built-in example database. This function has side effects: it calls gdb.init_examples which resets the working database, and it creates then deletes temporary tracks ('example_dataset_track') and intervals ('example_dataset_intervals') in that database.

Usage

```
gdataset.example_path()
```

Details

This function performs the following steps:

1. Calls gdb.init_examples() to set the working database
2. Removes any existing 'example_dataset_track' and 'example_dataset_intervals'
3. Creates temporary track and intervals in the example database
4. Saves them to a new dataset in a temporary directory
5. Removes the temporary track and intervals from the example database

This is primarily intended for use in examples and tests. Users should be aware that calling this function will change their current working database.

Value

Path to the created dataset directory (in a temporary location)

See Also

gdataset.save, gdataset.load, gdb.init_examples

Examples

```
dataset_path <- gdataset.example_path()
gdataset.load(dataset_path)
gdataset.unload(dataset_path)
```

---

gdataset.info *Get dataset information*

Description

Returns metadata and contents of a dataset.

Usage

```
gdataset.info(path)
```

Arguments

path	Path to any dataset (loaded or not)

Value

List with dataset information

See Also

gdataset.ls, gdataset.load

Examples

```
dataset_path <- gdataset.example_path()
gdataset.info(dataset_path)
```

`gdataset.load` *Load a dataset into the namespace*

Description

Loads tracks and intervals from a dataset directory, making them available for analysis alongside the working database.

Usage

```
gdataset.load(path, force = FALSE, verbose = FALSE)
```

Arguments

`path`	Path to a dataset or misha database directory
`force`	If TRUE, ignore name collisions (working db wins; for dataset-to-dataset, later-loaded wins)
`verbose`	If TRUE, print loaded track/interval names and summary counts

Value

Invisibly returns a list with:

`tracks`	Number of visible tracks loaded
`intervals`	Number of visible intervals loaded
`shadowed_tracks`	Number of tracks shadowed by collisions
`shadowed_intervals`	Number of intervals shadowed by collisions

See Also

`gdataset.unload`, `gdataset.save`, `gdataset.ls`

Examples

```
dataset_path <- gdataset.example_path()
gdataset.load(dataset_path)
gdataset.unload(dataset_path)
```

gdataset.ls *List working database and loaded datasets*

Description

Returns a list of the working database and all loaded datasets.

Usage

```
gdataset.ls(dataframe = FALSE)
```

Arguments

dataframe	If FALSE, return character vector; if TRUE, return data frame

Value

Character vector of paths or data frame with detailed information

See Also

gdataset.load, gdataset.info

Examples

```
dataset_path <- gdataset.example_path()
gdataset.load(dataset_path)
gdataset.ls()
gdataset.unload(dataset_path)
```

gdataset.save *Save a dataset*

Description

Creates a new dataset directory containing selected tracks and/or intervals from the working database.

Usage

```
gdataset.save(
  path,
  description,
  tracks = NULL,
  intervals = NULL,
  symlinks = FALSE,
  copy_seq = FALSE
)
```

Arguments

path	Destination directory (must not exist)
description	Required description for metadata
tracks	Character vector of track names to include
intervals	Character vector of interval set names to include
symlinks	If TRUE, create symlinks to tracks/intervals instead of copying
copy_seq	If TRUE, copy seq/ directory instead of symlinking

Value

Invisible path

See Also

gdataset.load, gdataset.info

Examples

```
gdb.init_examples()
example_intervs <- gintervals(1, 0, 10000)
gintervals.save("example_dataset_intervals", example_intervs)
gtrack.create(
    "example_dataset_track",
    "Example dataset track",
    "dense_track",
    iterator = "example_dataset_intervals"
)
dataset_path <- tempfile("misha_dataset_")
gdataset.save(
    path = dataset_path,
    description = "Example dataset",
    tracks = "example_dataset_track",
    intervals = "example_dataset_intervals"
)
gtrack.rm("example_dataset_track", force = TRUE)
gintervals.rm("example_dataset_intervals", force = TRUE)
```

---

gdataset.unload *Unload a dataset from the namespace*

Description

Removes all tracks and intervals from a previously loaded dataset. If a track was shadowing another, the shadowed track becomes visible again.

Usage

```
gdataset.unload(path, validate = FALSE)
```

Arguments

path	Path to a previously loaded dataset
validate	If TRUE, error if path is not currently loaded; otherwise silently no-op

Value

Invisible NULL

See Also

gdataset.load, gdataset.ls

Examples

```
dataset_path <- gdataset.example_path()
gdataset.load(dataset_path)
gdataset.unload(dataset_path, validate = TRUE)
```

---

gdb.build_genome *Build a misha genome database from a name*

Description

Builds a misha genomic database for a named assembly. Resolves the name through the registry chain (or pattern-fallback for GC[FA]_* accessions), downloads the FASTA, calls gdb.create to build the seq-only groot, then dispatches to gdb.install_intervals for the requested sets.

Usage

```
gdb.build_genome(
  name,
  path = name,
  registry = NULL,
  sets = c("genes", "rmsk", "cgi", "cytoband"),
  prefix = "",
  gene_sets = c(tss = "tss", exons = "exons", utr3 = "utr3", utr5 = "utr5"),
 gtf_priority = c("ncbiRefSeq", "bestRefSeq", "ensGene", "augustus", "xenoRefGene"),
  chrom_naming = NULL,
  target_chroms = NULL,
  target_lengths = NULL,
  min_coverage = 1,
```

```
  match_by_length = TRUE,
  format = NULL,
  verbose = TRUE
)
```

Arguments

name	Genome name (registry key, alias, or GC[FA]_* accession).
path	Output directory; must not exist.
registry	Optional path to an explicit registry YAML.
sets	Subset of c("genes", "rmsk", "cgi", "cytoband"). Empty vector character(0) = sequence-only build.
prefix	Character scalar prepended to set names (see gdb.install_intervals).
gene_sets	Named character vector mapping the four gintervals.import_genes() roles to on-disk set names; NA skips a role.
gtf_priority	Character vector ordering GTF source preference.
chrom_naming	Optional override for the recipe's chrom_naming. Selects which name space the canonical chrom names should come from. For ucsc-hub: any chromAlias column ("ucsc", "genbank", "refseq", "ncbi"), plus the friendly aliases "sequence_name" (= "assembly") and "accession" (keep the FASTA's source column). For ncbi: "sequence_name" (default), "ucsc", or "accession". NULL (default) keeps whatever the recipe specifies.
target_chroms	Optional character vector of chrom names the resulting groot should align to (typically the output of halStats --sequenceStats, the chrom names in a HAL file you intend to liftover against). When supplied, misha auto-picks the chromAlias column whose values cover target_chroms best and uses that column as the canonical naming, instead of chrom_naming. Honored only by the ucsc-hub backend; supplying it with any other source is an error (raised before any download).
target_lengths	Optional numeric vector aligned with target_chroms (typically the second field of halStats --sequenceStats). When supplied alongside target_chroms and with match_by_length = TRUE, this is the strong-guarantee path: misha force-aligns the hub FASTA to target_chroms, placing every target on its chromAlias row by name match across columns or unique-on-both-sides length pairing. If any target can't be placed, the build errors (in the pre-flight, before the multi-GB FASTA download). On success the resulting groot's chrom names are exactly target_chroms (alias rows not in target_chroms keep their original FASTA-header accession). Honored only by the ucsc-hub backend.
min_coverage	Minimum fraction of groot chroms that must appear in a chromAlias column for that column to be picked as canonical (forwarded to gdb.install_intervals). Default 1.0 (strict). Lower to e.g. 0.99 when a column has small gaps – typical when a target column doesn't span every contig (e.g. UCSC's genbank column has no value for the mitochondrion in many hubs, leaving 1 stray chrom). Honored only by the ucsc-hub backend; supplying a non-default value for any other source is an error (raised before any download).

match_by_length
: Forwarded to gdb.install_intervals. When TRUE (default), complements column-based canonical detection with a per-row length match for alias rows the chosen column couldn't cover, and switches asset translation to a cross-column per-row lookup so GFFs in any naming scheme import cleanly. Set FALSE for the stricter single-column-only behavior.

format
: "indexed" or "per-chromosome"; NULL => getOption("gmulticontig.indexed_format", TRUE).

verbose
: If TRUE, prints progress.

Details

For details on resolution, sources, sets, and chromosome-alias handling, see gdb.install_intervals.

Value

None (invisible NULL). The installed gene-derived sets (tss, exons, utr3, utr5) carry a name column (transcript/RNA accession) and a geneName column (gene symbol from the source annotation; blank when the source has no symbol).

See Also

gdb.install_intervals, gdb.create, gdb.list_genomes, gdb.genome_info.

Examples

```
## Not run:
gdb.build_genome("hg38", path = "~/genomes/hg38")
gdb.build_genome("GCA_004023825.1",
    path   = "~/genomes/arctic_fox",
    prefix = "intervs.global."
)
# Match HAL/Cactus canonical names (GenBank accessions like JH880237.1):
gdb.build_genome("GCF_000298355.1",
    path         = "~/genomes/Bos_mutus",
    chrom_naming = "genbank",
    prefix       = "intervs.global."
)

## End(Not run)
```

---

gdb.convert_to_indexed

Change Database to Indexed Genome Format

Description

Converts a per-chromosome database to indexed genome format with a single consolidated genome.seq file and genome.idx index. Optionally also converts tracks and interval sets to indexed format.

Usage

```
gdb.convert_to_indexed(
  groot = NULL,
  remove_old_files = FALSE,
  force = FALSE,
  validate = TRUE,
  convert_tracks = FALSE,
  convert_intervals = FALSE,
  verbose = FALSE,
  chunk_size = 104857600,
  threads = NULL
)
```

Arguments

groot	Root directory of the database to change to indexed format. If NULL, uses the currently active database.
remove_old_files	Logical. If TRUE, removes old per-chromosome files after successful conversion. Default: FALSE.
force	Logical. If TRUE, forces the conversion without confirmation. Default: FALSE.
validate	Logical. If TRUE, validates the conversion by comparing sequences. Default: TRUE.
convert_tracks	Logical. If TRUE, also converts all eligible tracks to indexed format. Default: FALSE.
convert_intervals	Logical. If TRUE, also converts all eligible interval sets to indexed format. Default: FALSE.
verbose	Logical. If TRUE, prints verbose messages. Default: FALSE.
chunk_size	Integer. The size of the chunk to read from the sequence files. Default: 104857600 (100MB). Reduce if you are running into memory issues.
threads	Integer or NULL. Number of parallel processes used when converting tracks and interval sets (each worker handles one track/interval set via `parallel::mclapply`). If `NULL` (default), uses `min(parallel::detectCores(), 8)`. Set to `1` for serial execution. Falls back to serial on non-Unix platforms (mclapply requires fork).

Details

This function converts a per-chromosome database (with separate .seq files per contig) to indexed format (single genome.seq + genome.idx). The indexed format provides better performance and scalability, especially for genomes with many contigs.

Important: Preserving Chromosome Order

For exact conversion that produces bit-for-bit identical results before and after conversion, you should load the source database first using `gsetroot()` or `gdb.init()`:

- If database is loaded: Uses chromosome order from ALLGENOME (exact preservation)
- If database is not loaded: Uses order from chrom_sizes.txt (may differ from ALLGENOME)

This ensures that the converted database has the exact same chromosome ordering, which affects iteration order, interval IDs, and other operations that depend on chromosome order.

The conversion process:

1. Checks if database is already in indexed format
2. Gets chromosome order from ALLGENOME (if loaded) or chrom_sizes.txt
3. Consolidates all per-chromosome .seq files into genome.seq
4. Creates genome.idx with CRC64 checksum
5. Optionally validates the conversion
6. Optionally removes old .seq files
7. If convert_tracks=TRUE, converts all eligible tracks (1D: dense, sparse, array; 2D: rectangles, points)
8. If convert_intervals=TRUE, converts all eligible interval sets (1D and 2D)

Tracks and intervals that cannot be converted (and are skipped):

- Tracks: virtual tracks, single-file tracks, already converted tracks
- Intervals: Single-file interval sets, already converted interval sets

Value

Invisible NULL

See Also

`gdb.create`, `gdb.init`, `gtrack.convert_to_indexed`, `gintervals.convert_to_indexed`, `gintervals.2d.convert_to`

Examples

```
## Not run:
# Recommended: Load database first for exact conversion
gsetroot("/path/to/database")
gdb.convert_to_indexed(
    convert_tracks = TRUE,
    convert_intervals = TRUE,
    remove_old_files = TRUE,
    verbose = TRUE
)

# Convert current database to indexed format (genome only)
gdb.convert_to_indexed()
```

```
# Convert specific database without loading it first
# Note: chromosome order may differ from ALLGENOME
gdb.convert_to_indexed(groot = "/path/to/database")

# Convert genome and all tracks to indexed format
gdb.convert_to_indexed(convert_tracks = TRUE)

# Full conversion with validation and cleanup
gsetroot("/path/to/database") # Load first for exact order preservation
gdb.convert_to_indexed(
    convert_tracks = TRUE,
    convert_intervals = TRUE,
    remove_old_files = TRUE,
    validate = TRUE,
    verbose = TRUE
)

## End(Not run)
```

---

gdb.create *Creates a new Genomic Database*

Description

Creates a new Genomic Database.

Usage

```
gdb.create(
  groot = NULL,
  fasta = NULL,
  genes.file = NULL,
  annots.file = NULL,
  annots.names = NULL,
  format = NULL,
  verbose = FALSE
)
```

Arguments

groot	path to newly created database
fasta	an array of names or URLs of FASTA files. Can contain wildcards for multiple files
genes.file	name or URL of file that contains genes. If 'NULL' no genes are imported
annots.file	name of URL file that contains annotations. If 'NULL' no annotations are imported
annots.names	annotations names

format	database format: "indexed" (default, single genome.seq + genome.idx) or "per-chromosome" (separate .seq file per contig). If NULL, uses the value from getOption("gmulticontig.indexed_format", TRUE)
verbose	if TRUE, prints verbose messages

Details

This function creates a new Genomic Database at the location specified by 'groot'. FASTA files are converted to 'Seq' format and appropriate 'chrom_sizes.txt' file is generated (see "User Manual" for more details).

Two database formats are supported:

- **indexed**: Single genome.seq + genome.idx (default). Recommended for genomes with many contigs. Provides better performance and scalability.
- **per-chromosome**: Separate .seq file per contig.

If 'genes.file' is not 'NULL' four sets of intervals are created in the database: `tss`, `exons`, `utr3` and `utr5`. See gintervals.import_genes for more details about importing genes intervals.

'fasta', 'genes.file' and 'annots.file' can be either a file path or URL in a form of 'ftp://[address]/[file]'. 'fasta' can also contain wildcards to indicate multiple files. Files that these arguments point to can be zipped or unzipped.

See the 'Genomes' vignette for details on how to create a database from common genome sources.

Value

None.

See Also

gdb.init, gdb.reload, gintervals.import_genes

Examples

```
# ftp <- "ftp://hgdownload.soe.ucsc.edu/goldenPath/mm10"
# mm10_dir <- file.path(tempdir(), "mm10")
# # only a single chromosome is loaded in this example
# # see "Genomes" vignette how to download all of them and how
# # to download other genomes
# gdb.create(
#     mm10_dir,
#     paste(ftp, "chromosomes", paste0(
#         "chr", c("X"),
#         ".fa.gz"
#     ), sep = "/"),
#     paste(ftp, "database/knownGene.txt.gz", sep = "/"),
#     paste(ftp, "database/kgXref.txt.gz", sep = "/"),
#     c(
#         "kgID", "mRNA", "spID", "spDisplayID", "geneSymbol",
#         "refseq", "protAcc", "description", "rfamAcc",
#         "tRnaName"
```

```
#     )
# )
# gdb.init(mm10_dir)
# gintervals.ls()
# gintervals.all()
```

---

gdb.create_genome *Create and Load a Genome Database*

Description

This function downloads, extracts, and loads a misha genome database for the specified genome.

Usage

```
gdb.create_genome(genome, path = getwd(), tmpdir = tempdir())
```

Arguments

genome	A character string specifying the genome to download. Supported genomes are "mm9", "mm10", "mm39", "hg19", and "hg38".
path	A character string specifying the directory where the genome will be extracted. Defaults to genome name (e.g. "mm10") in the current working directory.
tmpdir	A character string specifying the directory for storing temporary files. This is used for storing the downloaded genome file.

Details

The function checks if the specified genome is available. If tmpdir, it constructs the download URL, downloads the genome file, extracts it to the specified directory, and loads the genome database using gsetroot. The function also calls gdb.reload to reload the genome database.

Value

None.

Examples

```
## Not run:
mm10_dir <- tempdir()
gdb.create_genome("mm10", path = mm10_dir)
list.files(file.path(mm10_dir, "mm10"))
gsetroot(file.path(mm10_dir, "mm10"))
gintervals.ls()

## End(Not run)
```

gdb.create_linked *Create a linked database with symlinks to a parent database*

Description

Creates a new database directory structure with symbolic links to the parent database's seq/ directory and chrom_sizes.txt file.

Usage

```
gdb.create_linked(path, parent)
```

Arguments

path	Path for the new linked database
parent	Path to the parent database (with seq and chrom_sizes.txt)

Details

This is useful for creating a writable database that shares sequence data with a read-only main database. The new database can be set as the working database via gsetroot(), and then the parent database can be loaded as a dataset via gdataset.load().

Value

Invisible TRUE on success

See Also

gsetroot, gdb.create, gdataset.load, gdataset.ls

Examples

```
## Not run:
# Create linked database sharing sequence data with main database
gdb.create_linked("~/my_tracks", parent = "/shared/genomics/hg38")

# Set linked database as working database and load parent as dataset
gsetroot("~/my_tracks")
gdataset.load("/shared/genomics/hg38")

## End(Not run)
```

gdb.export_fasta *Export a database genome as FASTA*

Description

Writes all contigs from a misha database to a multi-FASTA file.

Usage

```
gdb.export_fasta(
  file = NULL,
  groot = NULL,
  line_width = 80L,
  chunk_size = 1000000L,
  overwrite = FALSE,
  verbose = FALSE
)
```

Arguments

file	Output FASTA file path
groot	Optional database root path. If NULL, uses current database.
line_width	Number of bases per FASTA line. Default: 80.
chunk_size	Number of bases to extract per chunk while writing. Default: 1000000.
overwrite	Logical. If TRUE, overwrite existing output file. Default: FALSE.
verbose	Logical. If TRUE, prints progress messages. Default: FALSE.

Details

By default, the currently active database is used. You can also provide groot to export another database without changing the caller's active database.

Value

Invisibly returns file.

See Also

gdb.init, gseq.extract

Examples

```
## Not run:
gdb.init_examples()
out <- tempfile(fileext = ".fa")
gdb.export_fasta(out)
head(readLines(out))

## End(Not run)
```

---

gdb.genome_info *Inspect a resolved genome recipe without building*

Description

Resolves `name` through the registry chain and returns the recipe (a list) along with the source it was resolved from. Useful for previewing what `gdb.build_genome` would do.

Usage

```
gdb.genome_info(name, registry = NULL)
```

Arguments

`name`	Genome name.
`registry`	Optional path to an explicit registry YAML.

Value

A list with components `recipe` (the resolved recipe) and `resolved_from` (the registry source).

See Also

`gdb.build_genome`, `gdb.list_genomes`.

Examples

```
gdb.genome_info("hg38")
gdb.genome_info("GCF_009806435.1")
```

gdb.get_readonly_attrs *Returns a list of read-only track attributes*

Description

Returns a list of read-only track attributes.

Usage

```
gdb.get_readonly_attrs()
```

Details

This function returns a list of read-only track attributes. These attributes are not allowed to be modified or deleted.

If no attributes are marked as read-only a 'NULL' is returned.

Value

A list of read-only track attributes.

See Also

gdb.set_readonly_attrs, gtrack.attr.get, gtrack.attr.set

gdb.info *Get Database Information*

Description

Returns information about a misha genome database including format, number of chromosomes, total genome size, and whether it uses the indexed format.

Usage

```
gdb.info(groot = NULL)
```

Arguments

groot	Root directory of the database. If NULL, uses the currently active database.

Value

A list with database information:

- path - Full path to the database
- is_db - TRUE if this is a valid misha database
- format - "indexed" or "per-chromosome"
- num_chromosomes - Number of chromosomes/contigs
- genome_size - Total length of genome in bases
- chromosomes - Data frame with chromosome names and sizes

Examples

```
## Not run:
# Get info about currently active database
info <- gdb.info()
cat("Database format:", info$format, "\n")
cat("Genome size:", info$genome_size / 1e6, "Mb\n")

# Get info about specific database
info <- gdb.info("/path/to/database")

## End(Not run)
```

---

gdb.init *Initializes connection with Genomic Database*

Description

Initializes connection with Genomic Database: loads the list of tracks, intervals, etc.

Usage

```
gdb.init(groot = NULL, dir = NULL, rescan = FALSE)

gsetroot(groot = NULL, dir = NULL, rescan = FALSE)
```

Arguments

groot	the root directory of the Genomic Database
dir	the current working directory inside the Genomic Database
rescan	indicates whether the file structure should be rescanned

Details

'gdb.init' initializes the connection with the Genomic Database. It is typically called first prior to any other function. When the package is attached it internally calls to 'gdb.init.examples' which opens the connection with the database located at 'PKGDIR/trackdb/test' directory, where 'PKGDIR' is the directory where the package is installed.

The current working directory inside the Genomic Database is set to 'dir'. If 'dir' is 'NULL', the current working directory is set to 'GROOT/tracks'.

If 'rescan' is 'TRUE', the list of tracks and intervals is achieved by rescanning directory structure under the current current working directory. Otherwise 'gdb.init' attempts to use the cached list that resides in 'groot/.db.cache' file.

Upon completion the connection is established with the database. If auto-completion mode is switched on (see 'gset_input_method') the list of tracks and intervals sets is loaded and added as variables to the global environment allowing auto-completion of object names with <TAB> key. Also a few variables are defined at an environment called `.misha`, and can be accessed using `.misha$variable`, e.g. `.misha$ALLGENOME`. These variables should not be modified by user.

GROOT	Root directory of Genomic Database
GWD	Current working directory inside Genomic Database
GTRACKS	List of all available tracks
GINTERVS	List of all available intervals
GVTRACKS	List of all available virtual tracks
ALLGENOME	List of all chromosomes and their sizes
GITERATOR.INTERVALS	A set of iterator intervals for which the track expression is evaluated

When option 'gmulticontig.indexed_format' is set to TRUE, the function loads a database with "indexed" track format.

Value

None.

See Also

`gdb.reload`, `gdb.create`, `gdir.cd`, `gtrack.ls`, `gintervals.ls`, `gvtrack.ls`

`gdb.init_examples` *Initialise the example Genomic Database*

Description

Extracts the bundled `testdb` tarball under `dir` and points the active groot at `<dir>/trackdb/test`. Used by examples and tests.

Usage

```
gdb.init_examples(dir = NULL)
```

Arguments

dir	Directory under which to extract the example trackdb. Created if it does not exist. Defaults to `Sys.getenv("MISHA_EXAMPLES_DIR")` if set, otherwise `tempdir()`.

Details

If `dir` is `NULL` (default), uses the value of environment variable `MISHA_EXAMPLES_DIR`, falling back to `tempdir()`. Set `MISHA_EXAMPLES_DIR` to a roomy path when `/tmp` is tight.

Value

None.

See Also

`gdb.init`

---

gdb.install_gff3_converter

Pre-install UCSC's gff3ToGenePred binary

Description

Downloads UCSC's `gff3ToGenePred` static binary (~25 MB) into `tools::R_user_dir("misha", "cache")/bin/`, verifies its SHA256, and makes it executable. Used by `gdb.build_genome` when the `ncbi` backend (or the `manual` backend with `genes_format: gff3`) is invoked. Calling it directly is useful in CI or in non-interactive scripts where the consent prompt would otherwise fail.

Usage

```
gdb.install_gff3_converter(force = FALSE)
```

Arguments

force	If `TRUE`, skip the consent prompt and re-download even if the binary is already cached.

Details

Override the binary location by setting environment variable `MISHA_GFF3_TO_GENEPRED` to a binary you provide (for example, one installed via `conda install -c bioconda ucsc-gff3togenepred`). This is the recommended workaround on systems whose glibc is older than the one UCSC's prebuilt binary requires.

Value

The cache path of the installed binary (invisibly).

Examples

```
## Not run:
gdb.install_gff3_converter()
Sys.setenv(MISHA_GFF3_TO_GENEPRED = "/path/to/your/gff3ToGenePred")

## End(Not run)
```

---

gdb.install_gtf_converter

Pre-install UCSC's gtfToGenePred binary

Description

Mirrors gdb.install_gff3_converter. Required for the ucsc-hub backend's genes set (UCSC mammal hubs ship GTFs).

Usage

```
gdb.install_gtf_converter(force = FALSE)
```

Arguments

force	If TRUE, skip consent prompt and re-download even if cached.

Details

Override the binary location by setting environment variable MISHA_GTF_TO_GENEPRED.

Value

The cache path (invisibly).

Examples

```
## Not run:
gdb.install_gtf_converter()
Sys.setenv(MISHA_GTF_TO_GENEPRED = "/path/to/your/gtfToGenePred")

## End(Not run)
```

gdb.install_intervals *Install interval sets onto an existing groot*

Description

Given an existing groot and a source recipe (or registry name, or accession), fetches the relevant annotation files and installs interval sets - one or more of genes / rmsk / cgi / cytoband.

Usage

```
gdb.install_intervals(
  groot,
  source,
  sets = c("genes", "rmsk", "cgi", "cytoband"),
  prefix = "",
  gene_sets = c(tss = "tss", exons = "exons", utr3 = "utr3", utr5 = "utr5"),
 gtf_priority = c("ncbiRefSeq", "bestRefSeq", "ensGene", "augustus", "xenoRefGene"),
  overwrite = FALSE,
  registry = NULL,
  target_chroms = NULL,
  target_lengths = NULL,
  min_coverage = 1,
  match_by_length = TRUE,
  force = FALSE,
  verbose = TRUE,
  prefetched_alias = NULL,
  .from_build_genome = FALSE
)
```

Arguments

groot	Path to a misha groot. NULL uses the active groot.
source	Either a registry name, a recipe list, or a bare GC[FA]_<digits>.<digits> accession.
sets	Subset of c("genes", "rmsk", "cgi", "cytoband").
prefix	Character scalar prepended verbatim to each set name. Include the trailing dot if you want one (e.g. "intervs.global.").
gene_sets	Named character vector mapping c("tss", "exons", "utr3", "utr5") to the on-disk set name. NA value skips that role.
gtf_priority	Character vector ordering GTF source preference for sources that ship multiple GTFs (currently ucsc-hub). First found wins.
overwrite	If FALSE (default), error on existing target sets. If TRUE, remove existing sets before saving.
registry	Optional path to a registry YAML; overrides the resolution chain.

`target_chroms` Optional character vector to pin the canonical column to (e.g. chrom names from `halStats --sequenceStats` for a HAL you intend to liftover against). When `NULL` (default) misha uses the groot's own chrom names (i.e. picks the alias column matching whatever is currently in the database). When supplied, misha picks the alias column matching `target_chroms` instead, and switches detection to count-weighted coverage (bp weighting requires lengths, which target chrom lists typically don't carry).

`target_lengths` Optional numeric vector aligned with `target_chroms`. Only honored when this call originates from `gdb.build_genome` (the groot was just force-aligned to `target_chroms`); standalone calls ignore it and use the strict column gate. When honored, canonical is set to a synthetic `".target_chroms"` column populated by name match + unique-length pairing, so `chrom_aliases.tsv` writes `target_chroms` as canonical with all other chromAlias columns as aliases.

`min_coverage` Minimum fraction that must be covered by a chromAlias column for it to be picked as the canonical mapping. Default `1.0` (strict). On the groot side this is bp-weighted (fraction of genome basepairs covered) - a long-tail of small unmapped contigs (e.g. a 16 kb mitochondrion missing from UCSC's genbank column out of a 3 Gb genome) costs ~0.0005 (asset chroms read from a GTF/GFF) the metric is the count-weighted fraction of distinct names. Unmapped contigs receive no annotations.

`match_by_length`

If `TRUE` (default), complement the column-based canonical detection with a per-row length-based fill: alias rows whose chosen column is empty are paired with a groot chrom of the same length, but only when the length is unique on both sides (ambiguous lengths are skipped, never guessed). Asset translation also switches to a per-row cross-column lookup, so a GFF in any naming scheme (or mixed schemes) imports cleanly. Currently honored only by the `ucsc-hub` backend (which ships per-contig lengths in `<acc>.chrom.sizes.txt`); other backends are unaffected. Set `FALSE` for the stricter single-column-only behavior.

`force` If `FALSE` (default), any requested set that the source doesn't provide raises an error and aborts the call before touching the groot. If `TRUE`, missing sets are demoted to a single summary warning and the available sets are installed.

`verbose` If `TRUE`, prints progress.

`prefetched_alias`

Optional pre-fetched chromAlias bundle (the return value of `.hub_preflight_coverage`). When supplied the ucsc-hub fetcher reuses it instead of re-downloading. Internal; users never set this directly.

`.from_build_genome`

Internal flag; when `TRUE`, `gdb.build_genome` signals that it has already rescanned the groot and we can skip the entry rescan. Users never set this directly.

Details

Decoupled from `gdb.build_genome` so that:

- users with a private FASTA build can layer canonical annotations onto it;
- failed installs can be resumed without re-fetching the FASTA;

- the same groot can host annotations from multiple sources under different prefixes (e.g. `intervs.global.`, `intervs.repeats.`).

Value

Invisible `NULL`. Side effects: writes `.interv` files under `<groot>/tracks/`, extends `<groot>/chrom_aliases.tsv`, appends to `<groot>/genome_info.yaml`, and re-initializes the active groot.

The gene-derived sets (`tss`, `exons`, `utr3`, `utr5`) carry a `name` column (transcript/RNA accession) and a `geneName` column (gene symbol from the source annotation; blank when the source has no symbol).

See Also

`gdb.build_genome`, `gdb.install_gtf_converter`.

Examples

```
## Not run:
# Standalone install on an existing groot.
gdb.install_intervals(
    groot  = "/genomes/arctic_fox",
    source = "GCA_004023825.1",
    prefix = "intervs.global."
)

# Layered: private FASTA groot + intervals from a UCSC hub assembly.
gdb.install_intervals(
    groot  = "/genomes/my_private",
    source = list(source = "ucsc-hub", accession = "GCF_009806435.1"),
    sets   = c("genes", "rmsk")
)

## End(Not run)
```

---

gdb.list_genomes *List resolvable genome names*

Description

Returns a data frame describing every genome resolvable from the active registry chain (see `gdb.build_genome` for the chain order).

Usage

```
gdb.list_genomes(registry = NULL)
```

Arguments

`registry`	Optional path to an explicit registry YAML, overriding the resolution chain.

Value

A data frame with columns:

- `name` - registry key.
- `source` - backend (`ucsc`, `ncbi`, `s3`, `local`, `manual`).
- `detail` - assembly / accession / path.
- `resolved_from` - which registry the entry came from.

See Also

`gdb.build_genome`, `gdb.genome_info`.

Examples

```
gdb.list_genomes()
```

`gdb.mark_cache_dirty` *Mark cached track list as dirty*

Description

When tracks or interval sets are modified outside of misha (e.g. files copied manually), the cached inventory may become out of date. Calling this helper marks the cache as dirty so the next `gsetroot()` forces a rescan.

Usage

```
gdb.mark_cache_dirty()
```

Value

Invisible `TRUE` if the dirty flag was written, `FALSE` otherwise.

See Also

`gdb.reload`, `gsetroot`

gdb.reload *Reloads database from the disk*

Description

Reloads database from disk: list of tracks, intervals, etc.

Usage

```
gdb.reload(rescan = TRUE)
```

Arguments

rescan	indicates whether the file structure should be rescanned

Details

Reloads Genomic Database from disk: list of tracks, intervals, etc. Use this function if you manually add tracks or if for any reason the database becomes corrupted. If 'rescan' is 'TRUE', the list of tracks and intervals is achieved by rescanning directory structure under the current current working directory. Otherwise 'gdb.reload' attempts to use the cached list that resides in 'GROOT/.db.cache' file.

Value

No return value, called for side effects.

See Also

gdb.init, gdb.create, gdir.cd,

gdb.set_readonly_attrs *Sets read-only track attributes*

Description

Sets read-only track attributes.

Usage

```
gdb.set_readonly_attrs(attrs)
```

Arguments

attrs	a vector of read-only attributes names or 'NULL'

Details

This function sets the list of read-only track attributes. The specified attributes may or may not already exist in the tracks.

If 'attrs' is 'NULL' the list of read-only attributes is emptied.

Value

None.

See Also

gdb.get_readonly_attrs, gtrack.attr.get, gtrack.attr.set

---

gdb.unload *Unloads the genome database*

Description

Unloads the currently active genome database and clears the session state.

Usage

```
gdb.unload()
```

Details

Resets the misha session to an uninitialized state: the database root, the working directory, the genome intervals, the virtual tracks, the loaded datasets, the chromosome aliases and the track / intervals-set auto-completion symbols are all cleared. After calling this function gdb.init (or gsetroot) must be called again before any other misha function is used.

Package internals are preserved, so the package itself stays loaded; this is not the same as detaching the package.

Value

None.

See Also

gdb.init, gsetroot, gdb.reload

Examples

```
gdb.init_examples()
gdb.unload()
gdb.init_examples()
```

gdir.cd *Changes current working directory in Genomic Database*

Description

Changes current working directory in Genomic Database.

Usage

```
gdir.cd(dir = NULL)
```

Arguments

dir	directory path

Details

This function changes the current working directory in Genomic Database (not to be confused with shell's current working directory). The list of database objects - tracks, intervals, track variables - is rescanned recursively under 'dir'. Object names are updated with the respect to the new current working directory. Example: a track named 'subdir.dense' will be referred as 'dense' once current working directory is set to 'subdir'. All virtual tracks are removed.

Value

None.

See Also

gdb.init, gdir.cwd, gdir.create, gdir.rm

Examples

```
gdb.init_examples()
gdir.cd("subdir")
gtrack.ls()
gdir.cd("..")
gtrack.ls()
```

gdir.create *Creates a new directory in Genomic Database*

Description

Creates a new directory in Genomic Database.

Usage

```
gdir.create(dir = NULL, showWarnings = TRUE, mode = "0777")
```

Arguments

dir	directory path
showWarnings	see 'dir.create'
mode	see 'dir.create'

Details

This function creates a new directory in Genomic Database. Creates only the last element in the specified path.

Value

None.

Note

A new directory cannot be created within an existing track directory.

See Also

dir.create, gdb.init, gdir.cwd, gdir.rm

gdir.cwd *Returns the current working directory in Genomic Database*

Description

Returns the absolute path of the current working directory in Genomic Database.

Usage

```
gdir.cwd()
```

Details

This function returns the absolute path of the current working directory in Genomic Database (not to be confused with shell's current working directory).

Value

A character string of the path.

See Also

gdb.init, gdir.cd, gdir.create, gdir.rm

---

gdir.rm *Deletes a directory from Genomic Database*

Description

Deletes a directory from Genomic Database.

Usage

```
gdir.rm(dir = NULL, recursive = FALSE, force = FALSE)
```

Arguments

dir	directory path
recursive	if 'TRUE', the directory is deleted recursively
force	if 'TRUE', suppresses user confirmation of tracks/intervals removal

Details

This function deletes a directory from Genomic Database. If 'recursive' is 'TRUE', the directory is deleted with all the files/directories it contains. If the directory contains tracks or intervals, the user is prompted to confirm the deletion. Set 'force' to 'TRUE' to suppress the prompt.

Value

None.

See Also

gdb.init, gdir.create, gdir.cd, gdir.cwd

gdist *Calculates distribution of track expressions*

Description

Calculates distribution of track expressions' values over the given set of bins.

Usage

```
gdist(
  ...,
  intervals = NULL,
  include.lowest = FALSE,
  iterator = NULL,
  band = NULL,
  dataframe = FALSE,
  names = NULL
)
```

Arguments

...	pairs of 'expr', 'breaks' where 'expr' is a track expression and the breaks determine the bin. An extra, unpaired trailing argument is taken as the genomic scope (equivalent to 'intervals'); it cannot be combined with a named 'intervals' argument - that combination is ambiguous and raises an error.
intervals	genomic scope for which the function is applied
include.lowest	if 'TRUE', the lowest value of the range determined by breaks is included
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expressions.
band	track expression band. If 'NULL' no band is used.
dataframe	return a data frame instead of an N-dimensional vector.
names	names for track expressions in the returned dataframe (only relevant when dataframe == TRUE)

Details

This function calculates the distribution of values of the numeric track expressions over the given set of bins.

The range of bins is determined by 'breaks' argument. For example: 'breaks=c(x1, x2, x3, x4)' represents three different intervals (bins): (x1, x2], (x2, x3], (x3, x4].

If 'include.lowest' is 'TRUE' the the lowest value will be included in the first interval, i.e. in [x1, x2]

'gdist' can work with any number of dimensions. If more than one 'expr'-'breaks' pair is passed, the result is a multidimensional vector, and an individual value can be accessed by [i1,i2,...,iN] notation, where 'i1' is the first track and 'iN' is the last track expression.

Value

N-dimensional vector where N is the number of 'expr'-'breaks' pairs. If `dataframe == TRUE` - a data frame with a column for each track expression and an additional column 'n' with counts.

NaN values

A track expression evaluates to `NaN` wherever the iterator produces a bin the track has no data for. What happens next depends on the function:

- `gextract` **keeps** `NaN` rows, so the result has one row per iterator interval whether or not the track covered it.
- `gsummary` **counts** them and reports the count as the "NaN intervals" element, while the statistics themselves are computed over the non-`NaN` values only.
- `gdist`, `gquantiles` and `gscreen` **drop** them: `NaN` bins are not counted into any distribution bin, do not contribute to a percentile, and never satisfy a screening condition - including a condition that would be true of every real value.
- `gsegment` **spans** them: a `NaN` bin contributes no evidence to the test that places a boundary, but it still falls inside whichever segment surrounds it, so the returned segments tile the scope continuously rather than skipping the gaps.

So on 20 bins of which 7 are `NaN`, `gextract` returns 20 rows, `gsummary` reports 20 total and 7 `NaN`, and `gdist` counts 13; and on a 300 kb scope where 120 of 300 bins are `NaN`, `gsegment` still returns segments covering the full 300 kb.

The practical consequence is that `NaN` and zero are different, and collapsing them with `ifelse(is.na(x), 0, x)` turns "no data here" into a measured value of zero. Where that is genuinely what you want, note that it also changes every mean, quantile and distribution computed downstream.

See Also

`gextract`

Examples

```
gdb.init_examples()

## calculate the distribution of dense_track for bins:
## (0, 0.2], (0.2, 0.5] and (0.5, 1]
gdist("dense_track", c(0, 0.2, 0.5, 1))

## calculate two-dimensional distribution:
## dense_track vs. sparse_track
gdist("dense_track", seq(0, 1, by = 0.1), "sparse_track",
    seq(0, 2, by = 0.2),
    iterator = 100
)
```

gextract *Returns evaluated track expression*

Description

Returns the result of track expressions evaluation for each of the iterator intervals.

Usage

```
gextract(
  ...,
  intervals = NULL,
  colnames = NULL,
  iterator = NULL,
  band = NULL,
  file = NULL,
  intervals.set.out = NULL,
  intervals_join = c("id", "intervals", "none")
)
```

Arguments

`...`	track expression
`intervals`	genomic scope for which the function is applied
`colnames`	sets the columns names in the returned value. If 'NULL' names are set to track expression.
`iterator`	track expression iterator. If 'NULL' iterator is determined implicitly based on track expressions.
`band`	track expression band. If 'NULL' no band is used.
`file`	file name where the function result is optionally outputted in tab-delimited format
`intervals.set.out`	intervals set name where the function result is optionally outputted
`intervals_join`	how the output relates to the input intervals data frame. `"id"` (default) appends an `intervalID` integer column carrying the 1-based row index of the originating input interval. `"intervals"` drops `intervalID` and instead attaches every column of the input intervals data frame (coords + metadata) to each output row, suffixing names that collide with existing output columns with `"1"`. `"none"` drops `intervalID` and attaches nothing. The `"intervals"` mode is only supported when the result is returned in memory; combining it with `file` or `intervals.set.out` raises an error.

Details

This function returns the result of track expressions evaluation for each of the iterator intervals. The returned value is a set of intervals with an additional column for each of the track expressions. This value can be used as an input for any other function that accepts intervals. If the intervals inside 'intervals' argument overlap gextract returns the overlapped coordinate more than once.

The order inside the result might not be the same as the order of intervals. An additional column 'intervalID' is added to the return value. Use this column to refer to the index of the original interval from the supplied 'intervals'.

If 'file' parameter is not 'NULL' the result is outputted to a tab-delimited text file (without 'intervalID' column) rather than returned to the user. This can be especially useful when the result is too big to fit into the physical memory. The resulted file can be used as an input for 'gtrack.import' or 'gtrack.array.import' functions.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Similarly to 'file' parameter 'intervals.set.out' can be useful to overcome the limits of the physical memory.

'colnames' parameter controls the names of the columns that contain the evaluated expressions. By default the column names match the track expressions.

Value

If 'file' and 'intervals.set.out' are 'NULL' a set of intervals with an additional column for each of the track expressions and 'intervalID' column.

NaN values

A track expression evaluates to `NaN` wherever the iterator produces a bin the track has no data for. What happens next depends on the function:

- `gextract` **keeps** `NaN` rows, so the result has one row per iterator interval whether or not the track covered it.
- `gsummary` **counts** them and reports the count as the "NaN intervals" element, while the statistics themselves are computed over the non-`NaN` values only.
- `gdist`, `gquantiles` and `gscreen` **drop** them: `NaN` bins are not counted into any distribution bin, do not contribute to a percentile, and never satisfy a screening condition - including a condition that would be true of every real value.
- `gsegment` **spans** them: a `NaN` bin contributes no evidence to the test that places a boundary, but it still falls inside whichever segment surrounds it, so the returned segments tile the scope continuously rather than skipping the gaps.

So on 20 bins of which 7 are `NaN`, `gextract` returns 20 rows, `gsummary` reports 20 total and 7 `NaN`, and `gdist` counts 13; and on a 300 kb scope where 120 of 300 bins are `NaN`, `gsegment` still returns segments covering the full 300 kb.

The practical consequence is that `NaN` and zero are different, and collapsing them with `ifelse(is.na(x), 0, x)` turns "no data here" into a measured value of zero. Where that is genuinely what you want, note that it also changes every mean, quantile and distribution computed downstream.

See Also

gtrack.array.extract, gsample, gtrack.import, gtrack.array.import, glookup, gpartition, gdist

Examples

```
gdb.init_examples()

## get values of 'dense_track' for [0, 400), chrom 1
gextract("dense_track", gintervals(1, 0, 400))

## get values of 'rects_track' (a 2D track) for a 2D interval
gextract(
    "rects_track",
    gintervals.2d("chr1", 0, 4000, "chr2", 2000, 5000)
)
```

---

ggenome.implant *Implant donor sequences into a reference genome*

Description

Replaces specified intervals in a reference genome with donor DNA sequences and writes the result as a new FASTA file. Optionally creates a misha trackdb from the output.

Usage

```
ggenome.implant(
  intervals,
  donor,
  output,
  genome_fasta = NULL,
  create_trackdb = TRUE,
  trackdb_path = NULL,
  line_width = 80L,
  overwrite = FALSE
)
```

Arguments

intervals	A data.frame with chrom, start, end columns specifying the regions to replace. Coordinates are 0-based, half-open (standard misha convention).
donor	Either a character vector of DNA sequences (one per interval row), or a single string path to a misha database root from which sequences will be extracted at the same intervals.

output	Path for the output FASTA file.
genome_fasta	Path to the reference FASTA file to edit. If NULL, the current misha database is exported via gdb.export_fasta.
create_trackdb	Logical. If TRUE, creates a misha trackdb from the output FASTA using gdb.create.
trackdb_path	Path for the new trackdb. Defaults to <dirname(output)>/trackdb.
line_width	Integer. Number of bases per FASTA line. Default: 80.
overwrite	Logical. If TRUE, overwrite existing output file. Default: FALSE.

Details

This function fills the gap between manual sequence extraction and genome perturbation by providing a single-call interface for editing genomes.

The donor parameter controls where replacement sequences come from:

- If donor is a character vector with one sequence per interval row, those literal sequences are used directly.
- If donor is a single string pointing to an existing directory, it is treated as a misha database root. Sequences are extracted from that database at the same coordinates as intervals.

Perturbations are applied in reverse coordinate order within each chromosome so that earlier coordinates remain valid when later ones are replaced.

Value

Invisibly returns the output FASTA path.

See Also

ggenome.transplant, gdb.export_fasta, gseq.extract, gdb.create

Examples

```
gdb.init_examples()

# Export the example DB to a reference FASTA
ref_fasta <- tempfile(fileext = ".fa")
gdb.export_fasta(ref_fasta)

# Replace two regions with literal sequences
intervals <- data.frame(
    chrom = c("chr1", "chr1"),
    start = c(100, 200),
    end = c(110, 210)
)
donors <- c("AAAAAAAAAA", "CCCCCCCCCC")
out <- tempfile(fileext = ".fa")
trackdb <- tempfile()
ggenome.implant(intervals, donors,
    output = out,
```

```
    genome_fasta = ref_fasta,
    create_trackdb = TRUE,
    trackdb_path = trackdb
)

# Verify the implanted sequences via the new trackdb
gdb.init(trackdb)
gseq.extract(data.frame(chrom = "chr1", start = 100, end = 110))

# Clean up
unlink(c(ref_fasta, out, paste0(out, ".fai"), trackdb), recursive = TRUE)
```

`ggenome.transplant` *Transplant sequences from one genome into another*

Description

Sugar function that extracts sequences from a source genome at the given intervals and implants them into a target genome. Equivalent to calling `ggenome.implant` with `donor = source_genome` and `genome_fasta = target_genome`.

Usage

```
ggenome.transplant(
  intervals,
  source_genome,
  target_genome = NULL,
  output,
  create_trackdb = TRUE,
  trackdb_path = NULL,
  line_width = 80L,
  overwrite = FALSE
)
```

Arguments

`intervals`	A data.frame with `chrom`, `start`, `end` columns specifying the regions to transplant.
`source_genome`	Path to the misha database root containing the donor sequences.
`target_genome`	Path to the target reference FASTA file. If `NULL`, the current misha database is used.
`output`	Path for the output FASTA file.
`create_trackdb`	Logical. If `TRUE`, creates a misha trackdb.
`trackdb_path`	Path for the new trackdb.
`line_width`	Integer. Number of bases per FASTA line.
`overwrite`	Logical. If `TRUE`, overwrite existing output.

Value

Invisibly returns the output FASTA path.

See Also

ggenome.implant, gdb.export_fasta, gseq.extract

Examples

```
gdb.init_examples()

# Create a "donor" DB with different sequence (all T's)
donor_fasta <- tempfile(fileext = ".fa")
cat(">chr1\n", paste(rep("T", 500000), collapse = ""), "\n",
    ">chr2\n", paste(rep("T", 300000), collapse = ""), "\n",
    file = donor_fasta, sep = ""
)
donor_db <- tempfile()
gdb.create(donor_db, fasta = donor_fasta, verbose = FALSE)

# Export the current DB as the target FASTA
gdb.init_examples()
ref_fasta <- tempfile(fileext = ".fa")
gdb.export_fasta(ref_fasta)

# Transplant donor sequence into positions 100-200 of chr1
intervals <- data.frame(chrom = "chr1", start = 100, end = 200)
out <- tempfile(fileext = ".fa")
trackdb <- tempfile()
ggenome.transplant(intervals,
    source_genome = donor_db,
    target_genome = ref_fasta,
    output = out,
    create_trackdb = TRUE,
    trackdb_path = trackdb
)

# Verify: positions 100-200 should now be all T's
gdb.init(trackdb)
gseq.extract(data.frame(chrom = "chr1", start = 100, end = 200))

# Clean up
unlink(
    c(
        donor_fasta, donor_db, ref_fasta, out,
        paste0(out, ".fai"), trackdb
    ),
    recursive = TRUE
)
```

`gintervals` *Creates a set of 1D intervals*

Description

Creates a set of 1D intervals.

Usage

```
gintervals(chroms = NULL, starts = 0, ends = -1, strands = NULL)
```

Arguments

`chroms`	chromosomes - an array of strings with or without "chr" prefixes or an array of integers (like: '1' for "chr1")
`starts`	an array of start coordinates
`ends`	an array of end coordinates. If '-1' chromosome size is assumed.
`strands`	'NULL', a numeric vector of '-1', '0' or '1' values, or a character/factor vector with values "+", "-", ".", "*" or ""

Details

This function returns a set of one-dimensional intervals. The returned value can be used in all functions that accept 'intervals' argument.

One-dimensional intervals is a data frame whose first three columns are 'chrom', 'start' and 'end'. Each row of the data frame represents a genomic interval of the specified chromosome in the range of [start, end). Additional columns can be presented in 1D intervals object yet these columns must be added after the three obligatory ones.

If 'strands' argument is not 'NULL' an additional column "strand" is added to the intervals. The possible values of a strand can be '1' (plus strand), '-1' (minus strand) or '0' (unknown). Character values "+", "-", ".", "*" and "" (or factors with these levels) are also accepted and converted internally to '1', '-1' and '0' respectively.

Value

A data frame representing the intervals, sorted in the canonical chromosome order (which is not the order of the arguments). Beware of keeping per-interval data in a separate parallel vector across a `gintervals()` call: the rows are reordered and the vector is not. Put such data in a column of the resulting data frame instead.

See Also

`gintervals.2d`, `gintervals.force_range`

Examples

```
gdb.init_examples()

## the following 3 calls produce identical results
gintervals(1)
gintervals("1")
gintervals("chrX")

gintervals(1, 1000)
gintervals(c("chr2", "chrX"), 10, c(3000, 5000))
```

`gintervals.2d` *Creates a set of 2D intervals*

Description

Creates a set of 2D intervals.

Usage

```
gintervals.2d(
  chroms1 = NULL,
  starts1 = 0,
  ends1 = -1,
  chroms2 = NULL,
  starts2 = 0,
  ends2 = -1
)
```

Arguments

`chroms1`	chromosomes1 - an array of strings with or without "chr" prefixes or an array of integers (like: '1' for "chr1")
`starts1`	an array of start1 coordinates
`ends1`	an array of end1 coordinates. If '-1' chromosome size is assumed.
`chroms2`	chromosomes2 - an array of strings with or without "chr" prefixes or an array of integers (like: '1' for "chr1"). If 'NULL', 'chroms2' is assumed to be equal to 'chroms1'.
`starts2`	an array of start2 coordinates
`ends2`	an array of end2 coordinates. If '-1' chromosome size is assumed.

Details

This function returns a set of two-dimensional intervals. The returned value can be used in all functions that accept 'intervals' argument.

Two-dimensional intervals is a data frame whose first six columns are 'chrom1', 'start1', 'end1', 'chrom2', 'start2' and 'end2'. Each row of the data frame represents two genomic intervals from two chromosomes in the range of [start, end). Additional columns can be presented in 2D intervals object yet these columns must be added after the six obligatory ones.

Value

A data frame representing the intervals.

See Also

gintervals, gintervals.force_range

Examples

```
gdb.init_examples()

## the following 3 calls produce identical results
gintervals.2d(1)
gintervals.2d("1")
gintervals.2d("chrX")

gintervals.2d(1, 1000, 2000, "chrX", 400, 800)
gintervals.2d(c("chr2", "chrX"), 10, c(3000, 5000), 1)
```

---

gintervals.2d.all *Returns 2D intervals that cover the whole genome*

Description

Returns 2D intervals that cover the whole genome.

Usage

```
gintervals.2d.all()
```

Details

This function returns a set of two-dimensional intervals that cover the whole genome as it is defined by 'chrom_sizes.txt' file.

Value

A data frame representing the intervals.

See Also

gintervals.2d

---

gintervals.2d.band_intersect

Intersects two-dimensional intervals with a band

Description

Intersects two-dimensional intervals with a band.

Usage

```
gintervals.2d.band_intersect(
  intervals = NULL,
  band = NULL,
  intervals.set.out = NULL
)
```

Arguments

intervals	two-dimensional intervals
band	track expression band. If 'NULL' no band is used.
intervals.set.out	intervals set name where the function result is optionally outputted

Details

This function intersects each two-dimensional interval from 'intervals' with 'band'. If the intersection is not empty, the interval is shrunk to the minimal rectangle that contains the band and added to the return value.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a data frame representing the intervals.

See Also

gintervals.2d, gintervals.intersect

Examples

```
gdb.init_examples()
gintervals.2d.band_intersect(gintervals.2d(1), c(10000, 20000))
```

`gintervals.2d.convert_to_indexed`

Convert 2D interval set to indexed format

Description

Converts a per-chromosome interval set to indexed format (intervals2d.dat + intervals2d.idx) which reduces file descriptor usage.

Usage

```
gintervals.2d.convert_to_indexed(
  set.name = NULL,
  remove.old = FALSE,
  force = FALSE
)
```

Arguments

`set.name`	name of 2D interval set to convert
`remove.old`	if TRUE, removes old per-chromosome files after successful conversion
`force`	if TRUE, re-converts even if already in indexed format

Details

The indexed format stores all chromosome pairs in a single intervals2d.dat file with an intervals2d.idx index file. This dramatically reduces file descriptor usage, especially for genomes with many chromosomes (N*(N-1)/2 files to just 2).

Only non-empty pairs are stored in the index, avoiding O(N^2) space overhead.

The conversion process:

1. Scans directory for existing per-pair files
2. Creates temporary intervals2d.dat.tmp and intervals2d.idx.tmp files
3. Concatenates all per-pair files into intervals2d.dat.tmp
4. Builds index with pair offsets and checksums
5. Atomically renames temporary files to final names
6. Optionally removes old per-pair files

The indexed format is 100

Value

invisible NULL

Examples

```
## Not run:
# Convert a 2D interval set
gintervals.2d.convert_to_indexed("my_2d_intervals")

# Convert and remove old files
gintervals.2d.convert_to_indexed("my_2d_intervals", remove.old = TRUE)

# Force re-conversion
gintervals.2d.convert_to_indexed("my_2d_intervals", force = TRUE)

## End(Not run)
```

---

gintervals.2d.intersect

Intersects two sets of 2D intervals

Description

Intersects two sets of two-dimensional intervals.

Usage

```
gintervals.2d.intersect(intervals1 = NULL, intervals2 = NULL)
```

Arguments

`intervals1`, `intervals2`
: two-dimensional intervals (a data frame or the name of an intervals set)

Details

Returns the pairwise rectangle intersections between 'intervals1' and 'intervals2'. For every pair of rectangles that share the same (chrom1, chrom2) chromosome pair, the overlapping rectangle is computed by clipping each axis independently:

- start1 = max(start1 of the two rectangles), end1 = min(end1 ...)
- start2 = max(start2 of the two rectangles), end2 = min(end2 ...)

A rectangle is emitted only when it remains non-empty on both axes (start1 < end1 and start2 < end2). The result is not merged or canonicalized: because the union of two rectangles is not generally a rectangle, overlapping outputs are left as-is (unlike the 1D `gintervals.intersect`).

Value

A data frame with the six 2D interval columns, sorted as by `gintervals.2d`, or 'NULL' if the intersection is empty.

See Also

gintervals.2d, gintervals.2d.union, gintervals.intersect

Examples

```
gdb.init_examples()
a <- gintervals.2d("chr1", 0, 1000, "chr2", 0, 1000)
b <- gintervals.2d("chr1", 500, 1500, "chr2", 500, 1500)
gintervals.2d.intersect(a, b)
```

---

gintervals.2d.union *Unites two sets of 2D intervals*

Description

Unites two sets of two-dimensional intervals.

Usage

```
gintervals.2d.union(intervals1 = NULL, intervals2 = NULL)
```

Arguments

intervals1, intervals2
: two-dimensional intervals (a data frame or the name of an intervals set)

Details

Concatenates 'intervals1' and 'intervals2' and returns the combined set sorted as by gintervals.2d. Overlapping rectangles are not merged: because the union of two rectangles is not generally a rectangle, the result is simply the combined set (unlike the 1D gintervals.union).

Value

A data frame with the six 2D interval columns, sorted as by gintervals.2d.

See Also

gintervals.2d, gintervals.2d.intersect, gintervals.union

Examples

```
gdb.init_examples()
a <- gintervals.2d("chr1", 0, 1000, "chr2", 0, 1000)
b <- gintervals.2d("chr1", 500, 1500, "chr2", 500, 1500)
gintervals.2d.union(a, b)
```

gintervals.all *Returns 1D intervals that cover the whole genome*

Description

Returns 1D intervals that cover the whole genome.

Usage

```
gintervals.all()
```

Details

This function returns a set of one-dimensional intervals that cover the whole genome as it is defined by 'chrom_sizes.txt' file.

Value

A data frame representing the intervals.

See Also

gintervals

gintervals.annotate *Annotates 1D intervals using nearest neighbors*

Description

Annotates one-dimensional intervals by finding nearest neighbors in another set of intervals and adding selected columns from the neighbors to the original intervals.

Usage

```
gintervals.annotate(
  intervals,
  annotation_intervals,
  annotation_columns = NULL,
  column_names = NULL,
  dist_column = "dist",
  max_dist = Inf,
  na_value = NA,
  maxneighbors = 1,
  tie_method = c("first", "min.start", "min.end"),
  overwrite = FALSE,
  keep_order = TRUE,
```

```
  intervals.set.out = NULL,
  ...
)
```

Arguments

intervals	Intervals to annotate (1D).
annotation_intervals	Source intervals containing annotation data (1D).
annotation_columns	Character vector of column names to copy from annotation_intervals. If NULL (default), all non-basic columns are used, i.e. everything beyond the coordinate/strand columns among: chrom, start, end, chrom1, start1, end1, chrom2, start2, end2, strand.
column_names	Optional custom names for the annotation columns. If provided, must have the same length as annotation_columns. Defaults to using the original names.
dist_column	Name of the distance column to include. Use NULL to omit the distance column. Defaults to "dist".
max_dist	Maximum absolute distance. When finite, neighbors with |dist| > max_dist result in annotation columns being set to na_value for those rows, while the row itself is retained.
na_value	Value(s) to use for annotations when beyond max_dist or when no neighbor is found. Can be a single scalar recycled for all columns, or a named list/vector supplying per-column values matching column_names.
maxneighbors	Maximum number of neighbors per interval (duplicates intervals as needed). Defaults to 1.
tie_method	Tie-breaking when distances are equal: one of "first" (arbitrary but stable), "min.start" (smaller neighbor start first), or "min.end" (smaller neighbor end first). Applies when maxneighbors > 1.
overwrite	When FALSE (default), errors if selected annotation columns would overwrite existing columns in intervals. When TRUE, conflicting base columns are replaced by the annotation columns.
keep_order	If TRUE (default), preserves the original order of intervals rows in the output.
intervals.set.out	intervals set name where the function result is optionally outputted
...	Additional arguments forwarded to gintervals.neighbors (e.g., mindist, maxdist).

Details

The function wraps and extends gintervals.neighbors to provide convenient column selection/renaming, optional distance inclusion, distance thresholding with custom NA values, multiple neighbors per interval, and deterministic tie-breaking. Currently supports 1D intervals only.

- When annotation_columns = NULL, all non-basic columns present in annotation_intervals are included. - Setting dist_column = NULL omits the distance column. - If no neighbor is found for an interval, annotation columns are filled with na_value and the distance (when present) is

NA_real_. - Column name collisions are handled as follows: when overwrite=FALSE a clear error is emitted; when overwrite=TRUE, base columns with the same names are replaced by annotation columns.

Value

A data frame containing the original intervals plus the requested annotation columns (and optional distance column). If maxneighbors > 1, rows may be duplicated per input interval to accommodate multiple neighbors.

Examples

```
# Prepare toy data
intervs <- gintervals(1, c(1000, 5000), c(1100, 5050))
ann <- gintervals(1, c(900, 5400), c(950, 5500))
ann$remark <- c("a", "b")
ann$score <- c(10, 20)

# Basic usage with default columns (all non-basic columns)
gintervals.annotate(intervs, ann)

# Select specific columns, with custom names and distance column name
gintervals.annotate(
    intervs, ann,
    annotation_columns = c("remark"),
    column_names = c("ann_remark"),
    dist_column = "ann_dist"
)

# Distance threshold with scalar NA replacement
gintervals.annotate(
    intervs, ann,
    annotation_columns = c("remark"),
    max_dist = 200,
    na_value = "no_ann"
)

# Multiple neighbors with deterministic tie-breaking
nbrs <- gintervals.annotate(
    gintervals(1, 1000, 1100),
    {
        x <- gintervals(1, c(800, 1200), c(900, 1300))
        x$label <- c("left", "right")
        x
    },
    annotation_columns = "label",
    maxneighbors = 2,
    tie_method = "min.start"
)
nbrs

# Overwrite existing columns in the base intervals
```

```
intervs2 <- intervs
intervs2$remark <- c("orig1", "orig2")
gintervals.annotate(intervs2, ann, annotation_columns = "remark", overwrite = TRUE)
```

`gintervals.as_chain` *Transforms existing intervals to a chain format*

Description

Transforms existing intervals to a chain format by validating required columns and adding chain attributes.

Usage

```
gintervals.as_chain(
  intervals = NULL,
  src_overlap_policy = "error",
  tgt_overlap_policy = "auto",
  min_score = NULL
)
```

Arguments

`intervals`	a data frame with chain columns: chrom, start, end, strand, chromsrc, startsrc, endsrc, strandsrc, chain_id, score
`src_overlap_policy`	source overlap policy: "error", "keep", or "discard"
`tgt_overlap_policy`	target overlap policy: "error", "auto", "auto_first", "auto_longer", "auto_score", "discard", "keep", or "agg"
`min_score`	optional minimum alignment score threshold

Details

This function checks that the input intervals data frame has all the required columns for a chain format and adds the necessary attributes. A chain format requires both target coordinates (chrom, start, end, strand) and source coordinates (chromsrc, startsrc, endsrc, strandsrc), as well as chain_id and score columns.

Value

A data frame in chain format with chain attributes set

See Also

`gintervals.load_chain`, `gintervals.liftover`

Examples

```
gdb.init_examples()

# Create a chain from existing intervals
chain_data <- data.frame(
    chrom = "chr1",
    start = 1000,
    end = 2000,
    strand = 0,
    chromsrc = "chr1",
    startsrc = 5000,
    endsrc = 6000,
    strandsrc = 0,
    chain_id = 1L,
    score = 1000.0
)
chain <- gintervals.as_chain(chain_data)
```

---

gintervals.attr.export

Returns interval set attributes values

Description

Returns interval set attributes values.

Usage

```
gintervals.attr.export(intervals.set = NULL, attrs = NULL)
```

Arguments

intervals.set	a vector of interval set names or 'NULL'
attrs	a vector of attribute names or 'NULL'

Details

This function returns a data frame that contains interval set attribute values. Column names of the data frame consist of the attribute names, row names contain the interval set names.

The list of required interval sets is specified by 'intervals.set' argument. If 'intervals.set' is 'NULL' the attribute values of all existing interval sets are returned.

Likewise the list of required attributes is controlled by 'attrs' argument. If 'attrs' is 'NULL' all attribute values of the specified interval sets are returned. The columns are sorted then by "popularity" of an attribute, i.e. the number of interval sets containing this attribute. This sorting is not applied if 'attrs' is not 'NULL'.

Empty character string in a table cell marks a non-existing attribute.

Value

A data frame containing interval set attribute values.

See Also

gintervals.attr.import, gintervals.attr.get, gintervals.attr.set

Examples

```
gdb.init_examples()
gintervals.attr.set("annotations", "test_attr", "value1")
gintervals.attr.export()
gintervals.attr.export(intervals.set = "annotations")
gintervals.attr.export(attrs = "test_attr")
gintervals.attr.set("annotations", "test_attr", "")
```

---

gintervals.attr.get *Returns value of an interval set attribute*

Description

Returns value of an interval set attribute.

Usage

```
gintervals.attr.get(intervals.set = NULL, attr = NULL)
```

Arguments

intervals.set	interval set name
attr	attribute name

Details

This function returns the value of an interval set attribute. If the attribute does not exist an empty string is returned.

Value

Interval set attribute value (character string).

See Also

gintervals.attr.set, gintervals.attr.export, gintervals.attr.import

Examples

```
gdb.init_examples()
gintervals.attr.set("annotations", "test_attr", "value")
gintervals.attr.get("annotations", "test_attr")
gintervals.attr.set("annotations", "test_attr", "")
```

`gintervals.attr.import`

Imports interval set attributes values

Description

Imports interval set attributes values.

Usage

```
gintervals.attr.import(table = NULL, remove.others = FALSE)
```

Arguments

`table`	a data frame containing attribute values
`remove.others`	specifies what to do with the attributes that are not in the table

Details

This function imports attribute values contained in a data frame 'table'. The format of a table is similar to the one returned by 'gintervals.attr.export'. The values of the table must be character strings. Column names of the table should specify the attribute names, while row names should contain the interval set names.

The specified attributes of the specified interval sets are modified. If an attribute value is an empty string this attribute is removed from the interval set.

If 'remove.others' is 'TRUE' all attributes that do not appear in the table are removed, otherwise they are preserved unchanged.

Value

None.

See Also

`gintervals.attr.export`, `gintervals.attr.set`, `gintervals.attr.get`

Examples

```
gdb.init_examples()
t <- data.frame(
    myattr = "val1", row.names = "annotations",
    stringsAsFactors = FALSE
)
gintervals.attr.import(t)
gintervals.attr.export(attrs = "myattr")

# roll-back the changes
t$myattr <- ""
gintervals.attr.import(t)
```

`gintervals.attr.set` *Assigns value to an interval set attribute*

Description

Assigns value to an interval set attribute.

Usage

```
gintervals.attr.set(intervals.set = NULL, attr = NULL, value = NULL)
```

Arguments

`intervals.set`	interval set name
`attr`	attribute name
`value`	value (character string)

Details

This function creates an interval set attribute and assigns 'value' to it. If the attribute already exists its value is overwritten.

If 'value' is an empty string the attribute is removed.

Value

None.

See Also

`gintervals.attr.get`, `gintervals.attr.export`, `gintervals.attr.import`

Examples

```
gdb.init_examples()
gintervals.attr.set("annotations", "test_attr", "value")
gintervals.attr.get("annotations", "test_attr")
gintervals.attr.set("annotations", "test_attr", "")
```

`gintervals.canonic` *Converts intervals to canonic form*

Description

Converts intervals to canonic form.

Usage

```
gintervals.canonic(intervals = NULL, unify_touching_intervals = TRUE)
```

Arguments

`intervals`	intervals to be converted
`unify_touching_intervals`	if 'TRUE', touching one-dimensional intervals are unified

Details

This function converts 'intervals' into a "canonic" form: properly sorted with no overlaps. The result can be used later in the functions that require the intervals to be in canonic form. Use 'unify_touching_intervals' to control whether the intervals that touch each other (i.e. the end coordinate of one equals to the start coordinate of the other) are unified. 'unify_touching_intervals' is ignored if two-dimensional intervals are used.

Since 'gintervals.canonic' unifies overlapping or touching intervals, the number of the returned intervals might be less than the number of the original intervals. To allow the user to find the origin of the new interval 'mapping' attribute is attached to the result. It maps between the original intervals and the resulted intervals. Use 'attr(retv_of_gintervals.canonic, "mapping")' to retrieve the map.

Value

A data frame representing the canonic intervals and an attribute 'mapping' that maps the original intervals to the resulted ones.

See Also

gintervals, gintervals.2d

Examples

```
gdb.init_examples()

## Create intervals manually by using 'data.frame'.
## Note that we add an additional column 'data'.
## Return value:
##   chrom start   end data
## 1  chr1 11000 12000   10
## 2  chr1   100   200   20
## 3  chr1 10000 13000   30
## 4  chr1 10500 10600   40
intervs <- data.frame(
    chrom = "chr1",
    start = c(11000, 100, 10000, 10500),
    end = c(12000, 200, 13000, 10600),
    data = c(10, 20, 30, 40)
)

## Convert the intervals into the canonic form.
## The function discards any columns besides chrom, start and end.
## Return value:
##  chrom start   end
## 1  chr1   100   200
## 2  chr1 10000 13000
res <- gintervals.canonic(intervs)

## By inspecting mapping attribute we can see how the new
## intervals were created: "2 1 2 2" means that the first
## interval in the result was created from the second interval in
## the original set (we look for the indices in mapping where "1"
## appears). Likewise the second interval in the result was
## created from 3 intervals in the original set. Their indices are
## 1, 3 and 4 (once again we look for the indices in mapping where
## "2" appears).
## Return value:
## 2 1 2 2
attr(res, "mapping")

## Finally (and that is the most useful part of 'mapping'
## attribute): we add a new column 'data' to our result which is
## the mean value of the original data column. The trick is done
## using 'tapply' on par with 'mapping' attribute. For example,
## 20.00000 equals is a result of 'mean(intervs[2,]$data' while
## 26.66667 is a result of 'mean(intervs[c(1,3,4),]$data)'.
## 'res' after the following call:
##  chrom start   end     data
## 1  chr1   100   200 20.00000
## 2  chr1 10000 13000 26.66667
res$data <- tapply(intervs$data, attr(res, "mapping"), mean)
```

gintervals.chrom_sizes *Returns number of intervals per chromosome*

Description

Returns number of intervals per chromosome (or chromosome pair).

Usage

```
gintervals.chrom_sizes(intervals = NULL)
```

Arguments

intervals	intervals set

Details

This function returns number of intervals per chromosome (for 1D intervals) or chromosome pair (for 2D intervals).

Value

Data frame representing number of intervals per chromosome (for 1D intervals) or chromosome pair (for 2D intervals).

See Also

gintervals.load, gintervals.save, gintervals.exists, gintervals.ls, gintervals, gintervals.2d

Examples

```
gdb.init_examples()
gintervals.chrom_sizes("annotations")
```

gintervals.convert_to_indexed *Convert 1D interval set to indexed format*

Description

Converts a per-chromosome interval set to indexed format (intervals.dat + intervals.idx) which reduces file descriptor usage.

Usage

```
gintervals.convert_to_indexed(
  set.name = NULL,
  remove.old = FALSE,
  force = FALSE
)
```

Arguments

set.name	name of interval set to convert
remove.old	if TRUE, removes old per-chromosome files after successful conversion
force	if TRUE, re-converts even if already in indexed format

Details

The indexed format stores all chromosomes in a single intervals.dat file with an intervals.idx index file. This reduces file descriptor usage from N files (one per chromosome) to just 2 files.

The conversion process:

1. Creates temporary intervals.dat.tmp and intervals.idx.tmp files
2. Concatenates all per-chromosome files into intervals.dat.tmp
3. Builds index with offsets and checksums
4. Atomically renames temporary files to final names
5. Optionally removes old per-chromosome files

The indexed format is 100

Value

invisible NULL

See Also

gintervals.save, gintervals.load

Examples

```
## Not run:
# Convert an interval set
gintervals.convert_to_indexed("my_intervals")

# Convert and remove old files
gintervals.convert_to_indexed("my_intervals", remove.old = TRUE)

# Force re-conversion
gintervals.convert_to_indexed("my_intervals", force = TRUE)

## End(Not run)
```

---

gintervals.coverage_fraction

Calculate fraction of genomic space covered by intervals

Description

Returns the fraction of a genomic space that is covered by a set of intervals.

Usage

```
gintervals.coverage_fraction(intervals1 = NULL, intervals2 = NULL)
```

Arguments

intervals1	set of one-dimensional intervals (the covering set)
intervals2	set of one-dimensional intervals to be covered (default: NULL, meaning the entire genome)

Details

This function calculates what fraction of 'intervals2' is covered by 'intervals1'. If 'intervals2' is NULL, it calculates the fraction of the entire genome that is covered by 'intervals1'. Overlapping intervals in either set are automatically unified before calculation.

Value

A single numeric value between 0 and 1 representing the fraction of 'intervals2' (or the genome) covered by 'intervals1'.

See Also

gintervals, gintervals.intersect, gintervals.covered_bp, gintervals.all

Examples

```
gdb.init_examples()

# Create some intervals
intervs1 <- gscreen("dense_track > 0.15")
intervs2 <- gintervals(c("chr1", "chr2"), 0, c(100000, 100000))

# Calculate fraction of intervs2 covered by intervs1
gintervals.coverage_fraction(intervs1, intervs2)

# Calculate fraction of entire genome covered by intervs1
gintervals.coverage_fraction(intervs1)
```

---

gintervals.covered_bp *Calculate total base pairs covered by intervals*

Description

Returns the total number of base pairs covered by a set of intervals.

Usage

```
gintervals.covered_bp(intervals = NULL)
```

Arguments

intervals	set of one-dimensional intervals

Details

This function first canonicalizes the intervals to remove overlaps and touching intervals, then sums up the lengths of all resulting intervals. Overlapping intervals are counted only once.

Value

A single numeric value representing the total number of base pairs covered by the intervals.

See Also

gintervals, gintervals.canonic, gintervals.coverage_fraction

Examples

```
gdb.init_examples()

# Create some intervals
intervs <- gintervals(
    c("chr1", "chr1", "chr2"),
    c(100, 150, 1000),
    c(200, 250, 2000)
)

# Calculate total bp covered
# Note: intervals [100,200) and [150,250) overlap,
# so total is (200-100) + (250-150) + (2000-1000) = 100 + 100 + 1000 = 1200
# But after canonicalization: [100,250) + [1000,2000) = 150 + 1000 = 1150
gintervals.covered_bp(intervs)
```

---

gintervals.dataset *Returns the database/dataset path for interval sets*

Description

Returns the path of the database or dataset containing an interval set.

Usage

```
gintervals.dataset(intervals = NULL)
```

Arguments

intervals	interval set name or a vector of interval set names

Details

When datasets are loaded, interval sets can come from either the working database or from loaded datasets. This function returns the source path for each interval set.

Value

A character vector containing the database paths for each interval set. Returns NA for interval sets that don't exist in any connected database.

See Also

gintervals.dbs, gintervals.exists, gintervals.ls, gdataset.ls

Examples

```
gdb.init_examples()
gintervals.dataset("annotations1")
```

`gintervals.dbs` *Returns all database paths containing an interval set*

Description

Returns all database paths that contain a version of an interval set.

Usage

```
gintervals.dbs(intervals = NULL, dataframe = FALSE)
```

Arguments

`intervals`	interval set name
`dataframe`	return a data frame with columns `intervals` and `db`

Details

When datasets are loaded, an interval set may exist in multiple locations. This function computes on-demand and returns all such paths.

Value

A named character vector of database paths. If `dataframe` is TRUE, returns a data frame with columns `intervals` and `db`.

See Also

`gintervals.dataset`, `gintervals.ls`, `gdataset.ls`

Examples

```
gdb.init_examples()
gintervals.dbs("annotations1")
```

gintervals.diff *Calculates difference of two intervals sets*

Description

Returns difference of two sets of intervals.

Usage

```
gintervals.diff(intervals1 = NULL, intervals2 = NULL, intervals.set.out = NULL)
```

Arguments

intervals1, intervals2	set of one-dimensional intervals
intervals.set.out	intervals set name where the function result is optionally outputted

Details

This function returns a genomic space that is covered by 'intervals1' but not covered by 'intervals2'.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a data frame representing the intervals.

See Also

gintervals, gintervals.intersect, gintervals.union

Examples

```
gdb.init_examples()

intervs1 <- gscreen("dense_track > 0.15")
intervs2 <- gscreen("dense_track < 0.2")

## 'res3' equals to 'res4'
res3 <- gintervals.diff(intervs1, intervs2)
res4 <- gscreen("dense_track >= 0.2")
```

gintervals.exists *Tests for a named intervals set existence*

Description

Tests for a named intervals set existence.

Usage

```
gintervals.exists(intervals.set = NULL)
```

Arguments

intervals.set	name of an intervals set

Details

This function returns 'TRUE' if a named intervals set exists in Genomic Database.

Value

'TRUE' if a named intervals set exists. Otherwise 'FALSE'.

See Also

gintervals.ls, gintervals.load, gintervals.rm, gintervals.save, gintervals, gintervals.2d

Examples

```
gdb.init_examples()
gintervals.exists("annotations")
```

gintervals.force_range *Limits intervals to chromosomal range*

Description

Limits intervals to chromosomal range.

Usage

```
gintervals.force_range(intervals = NULL, intervals.set.out = NULL)
```

Arguments

intervals	intervals
intervals.set.out	intervals set name where the function result is optionally outputted

Details

This function enforces the intervals to be within the chromosomal range [0, chrom length) by altering the intervals' boundaries. Intervals that lay entirely outside of the chromosomal range are eliminated. The new intervals are returned.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a data frame representing the intervals.

See Also

gintervals, gintervals.2d, gintervals.canonic

Examples

```
gdb.init_examples()
intervs <- data.frame(
    chrom = "chr1",
    start = c(11000, -100, 10000, 10500),
    end = c(12000, 200, 13000000, 10600)
)
gintervals.force_range(intervs)
```

---

gintervals.from_mat — *Convert an interval-indexed matrix back to an intervals + values data.frame*

Description

Inverse of gintervals.to_mat. Recovers intervals from attr(mat, "intervals") if mat is an intervs_mat, or from the explicit intervals argument if mat is a plain matrix. Never parses rownames.

Usage

```
gintervals.from_mat(mat, intervals = NULL)
```

Arguments

mat	an intervs_mat (produced by gintervals.to_mat) or a plain matrix (then intervals must be supplied).
intervals	data.frame with chrom, start, end columns, required when mat is a plain matrix. Must satisfy nrow(intervals) == nrow(mat); alignment is strictly positional. Must NOT be supplied when mat is already an intervs_mat.

Value

A data.frame with chrom, start, end (plus intervalID if it was present in the original input), followed by the value columns.

See Also

gintervals.to_mat

Examples

```
df <- data.frame(
    chrom = c("chr1", "chr2"),
    start = c(100L, 200L),
    end   = c(200L, 400L),
    t1    = c(1.0, 2.0)
)
mat <- gintervals.to_mat(df)
identical(gintervals.from_mat(mat)$t1, df$t1)

# plain matrix path:
plain <- unclass(mat)
attr(plain, "intervals") <- NULL
gintervals.from_mat(plain, intervals = df[, c("chrom", "start", "end")])
```

---

gintervals.from_strings

Creates 1D intervals from coordinate strings

Description

Creates a set of 1D intervals by parsing UCSC-style coordinate strings.

Usage

```
gintervals.from_strings(regions = NULL)
```

Arguments

regions	a character vector of region strings. Accepted formats per element: "chrom", "chrom:start-end", "chrom:start..end", optionally followed by ":+" or ":-".

Details

Parses strings of the form "chrom:start-end" ("chrom:start..end" is also accepted) into a 1D intervals data frame. A chromosome-only string such as "chr1" expands to the full chromosome extent. An optional trailing strand suffix ":+" or ":-" adds a "strand" column.

Coordinates are zero-based and half-open, matching the convention used by gintervals.

Value

A data frame representing the intervals, sorted as by gintervals. A "strand" column is added only when at least one region specifies a strand.

See Also

gintervals, gintervals.2d

Examples

```
gdb.init_examples()
gintervals.from_strings("chr1:1000-2000")
gintervals.from_strings(c("chr1:1000-2000:+", "chr2:50-60:-"))
```

---

gintervals.import_bed *Import intervals from a BED file*

Description

Reads a BED file, plain or gzipped, and returns a misha 1D intervals data frame. Track/browser/comment header lines are skipped automatically. Chromosome names are normalized through the active database's CHROM_ALIAS mechanism (so chr1 <-> 1 works without explicit configuration).

Usage

```
gintervals.import_bed(file = NULL, name = TRUE, score = TRUE, strand = TRUE)
```

Arguments

file	path to a BED file (.bed or .bed.gz). Zip archives are not supported - unzip them first.
name	if TRUE and a 4th column exists, include it as name.
score	if TRUE and a 5th (numeric) column exists, include it as score.
strand	if TRUE and a 6th column exists, include it as strand (mapped to 1/-1/0).

Details

BED is already 0-based half-open, so coordinates are taken as-is.

Value

A 1D intervals data frame, sorted by chrom and start.

See Also

gintervals.import_gff, gintervals.import_vcf.

`gintervals.import_genes`

Imports genes and annotations from files

Description

Imports genes and annotations from files.

Usage

```
gintervals.import_genes(
  genes.file = NULL,
  annots.file = NULL,
  annots.names = NULL
)
```

Arguments

genes.file	name or URL of file that contains genes
annots.file	name of URL file that contains annotations. If 'NULL' no annotations are imported
annots.names	annotations names

Details

This function reads a definition of genes from 'genes.file' and returns four sets of intervals: TSS, exons, 3utr and 5utr. In addition to the regular intervals columns 'strand' column is added. It contains '1' values for '+' strands and '-1' values for '-' strands.

If annotation file 'annots.file' is given then annotations are attached too to the intervals. The names of the annotations as they would appear in the return value must be specified in 'annots.names' argument.

Both 'genes.file' and 'annots.file' can be either a file path or URL in a form of 'ftp://[address]/[file]'. Files that these arguments point to can be zipped or unzipped.

Examples of 'genes.file' and 'annots.file' can be found here:

ftp://hgdownload.soe.ucsc.edu/goldenPath/hg19/database/knownGene.txt.gz ftp://hgdownload.soe.ucsc.ed

If a few intervals overlap (for example: two TSS regions) they are all unified to an interval that covers the whole overlapping region. 'strand' value is set to '0' if two or more of the overlapping intervals have different strands. The annotations of the overlapping intervals are concatenated to a single character string separated by semicolons. Identical values of overlapping intervals' annotation are eliminated.

Value

A list of four intervals sets named 'tss', 'exons', 'utr3' and 'utr5'. 'strand' column and annotations are attached to the intevals.

See Also

gintervals, gdb.create

---

gintervals.import_gff *Import intervals from a GFF/GTF file*

Description

Reads a GFF3 or GTF file (optionally gzipped) and returns a misha 1D intervals data frame. GFF/GTF are 1-based and inclusive on both ends; coordinates are converted to 0-based half-open by subtracting 1 from start and leaving end as-is.

Usage

```
gintervals.import_gff(file = NULL, feature = NULL, strand = TRUE, attrs = TRUE)
```

Arguments

file	path to a GFF/GTF file.
feature	optional feature-type filter (column 3 of GFF). Pass a character vector to keep only those types (e.g. "exon", c("gene", "transcript")).
strand	if TRUE, include the strand column.
attrs	if TRUE, include the raw attributes string as column attrs. The attribute string is not parsed.

Details

Chromosome names are normalized through the active database's CHROM_ALIAS mechanism.

Value

A 1D intervals data frame with columns chrom, start, end, and optionally strand, type, source, score, attrs.

See Also

gintervals.import_bed, gintervals.import_vcf.

---

gintervals.import_vcf *Import intervals from a VCF file*

Description

Reads a VCF/VCF.gz file and returns a misha 1D intervals data frame with one row per record. VCF is 1-based; `start` is set to `POS - 1` and `end` is set to `POS - 1 + nchar(REF)`, yielding a 0-based half-open span covering the reference allele.

Usage

```
gintervals.import_vcf(file = NULL, info = TRUE)
```

Arguments

`file`	path to a VCF/VCF.gz file.
`info`	if `TRUE`, include the raw INFO column as `info`. The string is not parsed.

Details

Chromosome names are normalized through the active database's `CHROM_ALIAS` mechanism.

Multi-allelic records are kept as a single row; the `ALT` column contains the original comma-separated string.

Value

A 1D intervals data frame with columns chrom, start, end, and id, ref, alt, qual, filter, optionally info.

See Also

gintervals.import_bed, gintervals.import_gff.

`gintervals.intersect` *Calculates an intersection of two sets of intervals*

Description

Calculates an intersection of two sets of intervals.

Usage

```
gintervals.intersect(
  intervals1 = NULL,
  intervals2 = NULL,
  intervals.set.out = NULL
)
```

Arguments

`intervals1`, `intervals2`	set of intervals
`intervals.set.out`	intervals set name where the function result is optionally outputted

Details

This function returns intervals that represent a genomic space which is achieved by intersection of 'intervals1' and 'intervals2'.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a data frame representing the intersection of intervals.

See Also

`gintervals.2d.band_intersect`, `gintervals.diff`, `gintervals.union`, `gintervals`, `gintervals.2d`

Examples

```
gdb.init_examples()

intervs1 <- gscreen("dense_track > 0.15")
intervs2 <- gscreen("dense_track < 0.2")

## 'intervs3' and 'intervs4' are identical
intervs3 <- gintervals.intersect(intervs1, intervs2)
intervs4 <- gscreen("dense_track > 0.15 & dense_track < 0.2")
```

gintervals.is.bigset *Tests for big intervals set*

Description

Tests for big intervals set.

Usage

```
gintervals.is.bigset(intervals.set = NULL)
```

Arguments

intervals.set	name of an intervals set

Details

This function tests whether 'intervals.set' is a big intervals set. Intervals set is big if it is stored in big intervals set format and given the current limits it cannot be fully loaded into memory.

Memory limit is controlled by 'gmax.data.size' option (see: 'getOption("gmax.data.size")').

Value

'TRUE' if intervals set is big, otherwise 'FALSE'.

See Also

gintervals.load, gintervals.save, gintervals.exists, gintervals.ls

Examples

```
gdb.init_examples()
gintervals.is.bigset("annotations")
```

gintervals.liftover *Converts intervals from another assembly*

Description

Converts intervals from another assembly to the current one.

Usage

```
gintervals.liftover(
  intervals = NULL,
  chain = NULL,
  src_overlap_policy = "error",
  tgt_overlap_policy = "auto",
  min_score = NULL,
  include_metadata = FALSE,
  canonic = FALSE,
  value_col = NULL,
 multi_target_agg = c("mean", "median", "sum", "min", "max", "count", "first", "last",
    "nth", "max.coverage_len", "min.coverage_len", "max.coverage_frac",
    "min.coverage_frac"),
  params = NULL,
  na.rm = TRUE,
  min_n = NULL
)
```

Arguments

intervals	intervals from another assembly
chain	name of chain file or data frame as returned by 'gintervals.load_chain'
src_overlap_policy	policy for handling source overlaps: "error" (default), "keep", or "discard". "keep" allows one source interval to map to multiple target intervals, "discard" discards all source intervals that have overlaps and "error" throws an error if source overlaps are detected.
tgt_overlap_policy	policy for handling target overlaps. One of:

Policy	Description
error	Throws an error if any target overlaps are detected.
auto	Default. Alias for "auto_score".
auto_score	Resolves overlaps by segmenting the target region and selecting the best chain for each segment based c
auto_longer	Resolves overlaps by segmenting and selecting the chain with the longest span for each segment. Tie-br
auto_first	Resolves overlaps by segmenting and selecting the chain with the lowest chain_id for each segment.
keep	Preserves all overlapping intervals.
discard	Discards any chain interval that has a target overlap with another chain interval.
agg	Segments overlaps into smaller disjoint regions where each region contains all contributing chains, allow
best_source_cluster	Best source cluster strategy based on source overlap. When multiple chains map a source interval, cluste

min_score	optional minimum alignment score threshold. Chains with scores below this value are filtered out. Useful for excluding low-quality alignments.
include_metadata	logical; if TRUE, adds 'score' column to the output indicating the alignment score of the chain used for each mapping. Only applicable with "auto_score" or "auto" policy.

`canonic` logical; if TRUE, merges adjacent target intervals that originated from the same source interval (same intervalID) and same chain (same chain_id). This is useful when a source interval maps to multiple adjacent target blocks due to chain gaps.

`value_col` optional character string specifying the name of a numeric column in the intervals data frame to track through the liftover. When specified, this column's values are preserved in the output with the same column name. Use with multi_target_agg to aggregate values when multiple source intervals map to overlapping target regions.

`multi_target_agg` aggregation method to use when value_col is specified. One of: "mean", "median", "sum", "min", "max", "count", "first", "last", "nth", "max.coverage_len", "min.coverage_len", "max.coverage_frac", "min.coverage_frac". Default: "mean". Ignored when value_col is NULL.

`params` additional parameters for specific aggregation methods. Currently only used for "nth" aggregation, where it specifies which element to select (e.g., params = 2 for second element, or params = list(n = 2)).

`na.rm` logical; if TRUE (default), NA values are removed before aggregation. If FALSE, any NA in the values will cause the result to be NA. Only used when value_col is specified.

`min_n` optional minimum number of non-NA observations required for aggregation. If fewer observations are available, the result is NA. NULL (default) means no minimum. Only used when value_col is specified.

Details

This function converts 'intervals' from another assembly to the current one. Chain file instructs how the conversion of coordinates should be done. It can be either a name of a chain file or a data frame in the same format as returned by 'gintervals.load_chain' function.

The converted intervals are returned. An additional column named 'intervalID' is added to the resulted data frame. For each interval in the resulted intervals it indicates the index of the original interval.

Note: When passing a pre-loaded chain (data frame), overlap policies cannot be specified - they are taken from the chain's attributes that were set during loading. When passing a chain file path, policies can be specified and will be used for loading.

Value

A data frame representing the converted intervals. For 1D intervals, always includes 'intervalID' (index of original interval) and 'chain_id' (identifier of the chain that produced the mapping) columns. The chain_id column is essential for distinguishing results when a source interval maps to multiple target regions via different chains (duplications). When include_metadata=TRUE, also includes 'score' column. When value_col is specified, includes the value column with its original name.

See Also

`gintervals.load_chain`, `gtrack.liftover`, `gintervals`

Examples

```
gdb.init_examples()
chainfile <- paste(.misha$GROOT, "data/test.chain", sep = "/")
intervs <- data.frame(
    chrom = "chr25", start = c(0, 7000),
    end = c(6000, 20000)
)
# Liftover with default policies
gintervals.liftover(intervs, chainfile)

# Liftover keeping source overlaps (one source interval may map to multiple targets)
# gintervals.liftover(intervs, chainfile, src_overlap_policy = "keep")
```

---

gintervals.load *Loads a named intervals set*

Description

Loads a named intervals set.

Usage

```
gintervals.load(
  intervals.set = NULL,
  chrom = NULL,
  chrom1 = NULL,
  chrom2 = NULL
)
```

Arguments

intervals.set	name of an intervals set
chrom	chromosome for 1D intervals set
chrom1	first chromosome for 2D intervals set
chrom2	second chromosome for 2D intervals set

Details

This function loads and returns intervals stored in a named intervals set.

If intervals set contains 1D intervals and 'chrom' is not 'NULL' only the intervals of the given chromosome are returned.

Likewise if intervals set contains 2D intervals and 'chrom1', 'chrom2' are not 'NULL' only the intervals of the given pair of chromosomes are returned.

For big intervals sets 'chrom' parameter (1D case) / 'chrom1', 'chrom2' parameters (2D case) must be specified. In other words: big intervals sets can be loaded only by chromosome or chromosome pair.

Value

A data frame representing the intervals.

See Also

gintervals.save, gintervals.is.bigset, gintervals.exists, gintervals.ls, gintervals, gintervals.2d

Examples

```
gdb.init_examples()
gintervals.load("annotations")
```

---

gintervals.load_chain *Loads assembly conversion table from a chain file*

Description

Loads assembly conversion table from a chain file.

Usage

```
gintervals.load_chain(
  file = NULL,
  src_overlap_policy = "error",
  tgt_overlap_policy = "auto",
  src_groot = NULL,
  min_score = NULL
)
```

Arguments

file	name of chain file
src_overlap_policy	policy for handling source overlaps: "error" (default), "keep", or "discard". "keep" allows one source interval to map to multiple target intervals, "discard" discards all source intervals that have overlaps and "error" throws an error if source overlaps are detected.
tgt_overlap_policy	policy for handling target overlaps. One of:

Policy	Description
error	Throws an error if any target overlaps are detected.
auto	Default. Alias for "auto_score".
auto_score	Resolves overlaps by segmenting the target region and selecting the best chain for each segment based o

auto_longer	Resolves overlaps by segmenting and selecting the chain with the longest span for each segment. Tie-br
auto_first	Resolves overlaps by segmenting and selecting the chain with the lowest chain_id for each segment.
keep	Preserves all overlapping intervals.
discard	Discards any chain interval that has a target overlap with another chain interval.
agg	Segments overlaps into smaller disjoint regions where each region contains all contributing chains, allow
best_source_cluster	Best source cluster strategy based on source overlap. When multiple chains map a source interval, cluste

`src_groot`	optional path to source genome database for validating source chromosomes and coordinates. If provided, the function temporarily switches to this database to verify that all source chromosomes exist and coordinates are within bounds, then restores the original database.
`min_score`	optional minimum alignment score threshold. Chains with scores below this value are filtered out. Useful for excluding low-quality alignments.

Details

This function reads a file in 'chain' format and returns assembly conversion table that can be used in 'gtrack.liftover' and 'gintervals.liftover'.

Source overlaps occur when the same source genome position maps to multiple target genome positions. Target overlaps occur when multiple source positions map to overlapping regions in the target genome.

The 'src_overlap_policy' controls how source overlaps are handled:

- "error" (default): Throw an error if source overlaps are detected
- "keep": Keep all mappings, allowing one source to map to multiple targets
- "discard": Remove all chain intervals involved in source overlaps

The 'tgt_overlap_policy' controls how target overlaps are handled:

- "error": Throw an error if target overlaps are detected
- "auto" (default) / "auto_first": Keep the first overlapping chain (original file order) by trimming or discarding later overlaps while keeping source/target lengths consistent
- "auto_longer": Keep the longer overlapping chain and trim/drop the shorter ones
- "discard": Remove all chain intervals involved in target overlaps
- "keep": Allow target overlaps to remain untouched (liftover must be able to handle them)

Value

A data frame representing assembly conversion table with columns: chrom, start, end, strand, chromsrc, startsrc, endsrc, strandsrc, chain_id, score.

See Also

`gintervals.liftover`, `gtrack.liftover`

Examples

```
gdb.init_examples()
chainfile <- paste(.misha$GROOT, "data/test.chain", sep = "/")
# Load chain file with default policies
gintervals.load_chain(chainfile)
```

`gintervals.ls` *Returns a list of named intervals sets*

Description

Returns a list of named intervals sets in Genomic Database.

Usage

```
gintervals.ls(
  pattern = "",
  db = NULL,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE
)
```

Arguments

`pattern`, `ignore.case`, `perl`, `fixed`, `useBytes`	see 'grep'
`db`	optional database path to filter intervals. If specified, only interval sets from that database are returned.

Details

This function returns a list of named intervals sets that match the pattern (see 'grep'). If called without any arguments all named intervals sets are returned.

When multiple databases are connected, the 'db' parameter can be used to filter intervals to only those from a specific database.

Value

An array that contains the names of intervals sets.

See Also

`grep`, `gintervals.exists`, `gintervals.load`, `gintervals.save`, `gintervals.rm`, `gintervals`, `gintervals.2d`, `gintervals.dataset`

Examples

```
gdb.init_examples()
gintervals.ls()
gintervals.ls(pattern = "annot*")
```

`gintervals.mapply` *Applies a function to values of track expressions*

Description

Applies a function to values of track expressions for each interval.

Usage

```
gintervals.mapply(
  FUN = NULL,
  ...,
  intervals = NULL,
  enable.gapply.intervals = FALSE,
  iterator = NULL,
  band = NULL,
  intervals.set.out = NULL,
  colnames = "value"
)
```

Arguments

`FUN`	function to apply, found via 'match.fun'
`...`	track expressions whose values are used as arguments for 'FUN'
`intervals`	intervals for which track expressions are calculated
`enable.gapply.intervals`	if 'TRUE', then a variable 'GAPPLY.INTERVALS' is available
`iterator`	track expression iterator. If 'NULL' iterator is determined implicitly based on track expressions.
`band`	track expression band. If 'NULL' no band is used.
`intervals.set.out`	intervals set name where the function result is optionally outputted
`colnames`	name of the column that contains the return values of 'FUN'. Default is "value".

Details

This function evaluates track expressions for each interval from 'intervals'. The resulted vectors are passed then as arguments to 'FUN'.

If the intervals are one-dimensional and have an additional column named 'strand' whose value is '-1', the values of the track expression are placed to the vector in reverse order.

The current interval index (1-based) is stored in 'GAPPLY.INTERVID' variable that is available during the execution of 'gintervals.mapply'. There is no guarantee about the order in which the intervals are processed. Do not rely on any specific order and use 'GITERATOR.INTERVID' variable to detect the current interval id.

If 'enable.gapply.intervals' is 'TRUE', an additional variable 'GAPPLY.INTERVALS' is defined during the execution of 'gintervals.mapply'. This variable stores the current iterator intervals prior to track expression evaluation. Please note that setting 'enable.gapply.intervals' to 'TRUE' might severely affect the run-time of the function.

Note: all the changes made in R environment by 'FUN' will be void if multitasking mode is switched on. One should also refrain from performing any other operations in 'FUN' that might be not "thread-safe" such as updating files, etc. Please switch off multitasking ('options(gmultitasking = FALSE)') if you wish to perform such operations.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a data frame representing intervals with an additional column that contains the return values of 'FUN'. The name of this additional column is specified by the 'colnames' parameter.

See Also

mapply

Examples

```
gdb.init_examples()
gintervals.mapply(
    max, "dense_track",
    gintervals(c(1, 2), 0, 10000)
)
gintervals.mapply(
    function(x, y) {
        max(x + y)
    }, "dense_track",
    "sparse_track", gintervals(c(1, 2), 0, 10000),
    iterator = "sparse_track"
)
# Using custom column name
gintervals.mapply(
    max, "dense_track",
```

```
    gintervals(c(1, 2), 0, 10000),
    colnames = "max_value"
)
```

---

gintervals.mark_overlaps *Mark overlapping intervals with a group ID*

Description

Mark overlapping intervals with a group ID

Usage

```
gintervals.mark_overlaps(
  intervals,
  group_col = "overlap_group",
  unify_touching_intervals = TRUE
)
```

Arguments

intervals	intervals set
group_col	name of the column to store the overlap group IDs (default: "overlap_group")
unify_touching_intervals	if 'TRUE', touching one-dimensional intervals are unified

Value

The intervals set with an additional column containing group IDs from gintervals.canonic mapping. All overlapping intervals will have the same group ID.

Examples

```
gdb.init_examples()
# Create sample overlapping intervals
intervs <- data.frame(
    chrom = "chr1",
    start = c(11000, 100, 10000, 10500),
    end = c(12000, 200, 13000, 10600),
    data = c(10, 20, 30, 40)
)

# Mark overlapping intervals
intervs_marked <- gintervals.mark_overlaps(intervs)

# Use custom column name
intervs_marked <- gintervals.mark_overlaps(intervs, group_col = "my_groups")
```

`gintervals.neighbors` *Finds neighbors between two sets of intervals*

Description

For each interval in 'intervals1', finds the closest intervals from 'intervals2'. Distance directionality can be determined by either the strand of the target intervals (intervals2, default) or the query intervals (intervals1). When no strand column is present, all intervals are treated as positive strand (strand = 1).

Usage

```
gintervals.neighbors(
  intervals1 = NULL,
  intervals2 = NULL,
  maxneighbors = 1,
  mindist = -1e+09,
  maxdist = 1e+09,
  mindist1 = -1e+09,
  maxdist1 = 1e+09,
  mindist2 = -1e+09,
  maxdist2 = 1e+09,
  na.if.notfound = FALSE,
  use_intervals1_strand = FALSE,
  warn.ignored.strand = TRUE,
  intervals.set.out = NULL
)
```

Arguments

`intervals1`, `intervals2`
: intervals

`maxneighbors`
: maximal number of neighbors

`mindist`, `maxdist`
: distance range for 1D intervals

`mindist1`, `maxdist1`, `mindist2`, `maxdist2`
: distance range for 2D intervals

`na.if.notfound`
: if 'TRUE' return 'NA' interval if no matching neighbors were found, otherwise omit the interval in the answer

`use_intervals1_strand`
: if 'TRUE' use intervals1 strand column for distance directionality instead of intervals2 strand. If intervals1 has no strand column, all intervals are treated as positive strand (strand = 1). Invalid strand values (not -1 or 1) will cause an error.

`warn.ignored.strand`
: if 'TRUE' (default) show warning when 'intervals1' contains a strand column that will be ignored for distance calculation

`intervals.set.out`
: intervals set name where the function result is optionally outputted

Details

This function finds for each interval in 'intervals1' the closest 'maxneighbors' intervals from 'intervals2'.

For 1D intervals the distance must fall in the range of ['mindist', 'maxdist'].

Distance is defined as the number of base pairs between the the last base pair of the query interval and the first base pair of the target interval.

Strand handling: By default, distance directionality is determined by the 'strand' column in 'intervals2' (if present). If 'use_intervals1_strand' is TRUE, distance directionality is instead determined by the 'strand' column in 'intervals1'. This is particularly useful for TSS analysis where you want upstream/downstream distances relative to gene direction.

Distance calculation modes:

- **use_intervals1_strand = FALSE (default):** Uses intervals2 strand for directionality
- **use_intervals1_strand = TRUE:** Uses intervals1 strand for directionality

Important: When 'use_intervals1_strand = TRUE', distance signs are interpreted as:

- **+ strand queries:** Negative distances = upstream, Positive distances = downstream
- **- strand queries:** Negative distances = downstream, Positive distances = upstream

For 2D intervals two distances are calculated and returned for each axis. The distances must fall in the range of ['mindist1', 'maxdist1'] for axis 1 and ['mindist2', 'maxdist2'] for axis 2. For selecting the closest 'maxneighbors' intervals Manhattan distance is used (i.e. dist1+dist2).

Note: 'use_intervals1_strand' is not yet supported for 2D intervals.

The names of the returned columns are made unique using `make.unique(colnames(df), sep = "")`, assuming 'df' is the result.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a data frame containing the pairs of intervals from 'intervals1', intervals from 'intervals2' and an additional column named 'dist' ('dist1' and 'dist2' for 2D intervals) representing the distance between the corresponding intervals. The intervals from intervals2 would be changed to 'chrom1', 'start1', and 'end1' and for 2D intervals chrom11, start11, end11 and chrom22, start22, end22. If 'na.if.notfound' is 'TRUE', the data frame contains all the intervals from 'intervals1' including those for which no matching neighbor was found. For the latter intervals an 'NA' neighboring interval is stated. If 'na.if.notfound' is 'FALSE', the data frame contains only intervals from 'intervals1' for which matching neighbor(s) was found.

See Also

`gintervals`, `gintervals.neighbors.upstream`, `gintervals.neighbors.downstream`

Examples

```
gdb.init_examples()

# Basic intervals
intervs1 <- giterator.intervals("dense_track",
    gintervals(1, 0, 4000),
    iterator = 233
)
intervs2 <- giterator.intervals(
    "sparse_track",
    gintervals(1, 0, 2000)
)

# Original behavior - no strand considerations
gintervals.neighbors(intervs1, intervs2, 10,
    mindist = -300,
    maxdist = 500
)

# Add strand to intervals2 - affects distance directionality (original behavior)
intervs2$strand <- c(1, 1, -1, 1)
gintervals.neighbors(intervs1, intervs2, 10,
    mindist = -300,
    maxdist = 500
)

# TSS analysis example - use intervals1 (TSS) strand for directionality
tss <- data.frame(
    chrom = c("chr1", "chr1", "chr1"),
    start = c(1000, 2000, 3000),
    end = c(1001, 2001, 3001),
    strand = c(1, -1, 1), # +, -, +
    gene = c("GeneA", "GeneB", "GeneC")
)

features <- data.frame(
    chrom = "chr1",
    start = c(500, 800, 1200, 1800, 2200, 2800, 3200),
    end = c(600, 900, 1300, 1900, 2300, 2900, 3300),
    feature_id = paste0("F", 1:7)
)

# Use TSS strand for distance directionality
result <- gintervals.neighbors(tss, features,
    maxneighbors = 2,
    mindist = -1000, maxdist = 1000,
    use_intervals1_strand = TRUE
)

# Convenience functions for common TSS analysis
# Find upstream neighbors (negative distances for + strand genes)
```

```
upstream <- gintervals.neighbors.upstream(tss, features,
    maxneighbors = 2, maxdist = 1000
)

# Find downstream neighbors (positive distances for + strand genes)
downstream <- gintervals.neighbors.downstream(tss, features,
    maxneighbors = 2, maxdist = 1000
)

# Find both directions
both_directions <- gintervals.neighbors.directional(tss, features,
    maxneighbors_upstream = 1,
    maxneighbors_downstream = 1,
    maxdist = 1000
)
```

`gintervals.neighbors.upstream`

Directional neighbor finding functions

Description

These functions find neighbors using query strand directionality, where upstream/downstream directionality is determined by the strand of the query intervals rather than the target intervals. This is particularly useful for TSS analysis where you want distances relative to gene direction.

Usage

```
gintervals.neighbors.upstream(
  query_intervals,
  target_intervals,
  maxneighbors = 1,
  maxdist = 1e+09,
  ...
)

gintervals.neighbors.downstream(
  query_intervals,
  target_intervals,
  maxneighbors = 1,
  maxdist = 1e+09,
  ...
)

gintervals.neighbors.directional(
  query_intervals,
  target_intervals,
```

```
  maxneighbors_upstream = 1,
  maxneighbors_downstream = 1,
  maxdist = 1e+09,
  ...
)
```

Arguments

query_intervals	intervals with strand information (query intervals)
target_intervals	intervals to search for neighbors
maxneighbors	maximum number of neighbors per query interval (default: 1)
maxdist	maximum distance to search (default: 1e+09)
...	additional arguments passed to gintervals.neighbors
maxneighbors_upstream	maximum upstream neighbors per query interval (default: 1)
maxneighbors_downstream	maximum downstream neighbors per query interval (default: 1)

Details

Distance interpretation:

- **Positive strand queries:** upstream distances < 0, downstream distances > 0
- **Negative strand queries:** upstream distances > 0, downstream distances < 0

If no strand column is present, all intervals are treated as positive strand.

Value

gintervals.neighbors.upstream data frame of upstream neighbors

gintervals.neighbors.downstream data frame of downstream neighbors

gintervals.neighbors.directional list with 'upstream' and 'downstream' components

See Also

gintervals.neighbors

Examples

```
gdb.init_examples()

# Create TSS intervals with strand information
tss <- data.frame(
    chrom = c("chr1", "chr1", "chr1"),
    start = c(1000, 2000, 3000),
    end = c(1001, 2001, 3001),
```

```
    strand = c(1, -1, 1), # +, -, +
    gene = c("GeneA", "GeneB", "GeneC")
)

# Create regulatory features
features <- data.frame(
    chrom = "chr1",
    start = c(500, 800, 1200, 1800, 2200, 2800, 3200),
    end = c(600, 900, 1300, 1900, 2300, 2900, 3300),
    feature_id = paste0("F", 1:7)
)

# Find upstream neighbors (promoter analysis)
upstream <- gintervals.neighbors.upstream(tss, features,
    maxneighbors = 2, maxdist = 1000
)
print(upstream)

# Find downstream neighbors (gene body analysis)
downstream <- gintervals.neighbors.downstream(tss, features,
    maxneighbors = 2, maxdist = 5000
)
print(downstream)

# Find both directions in one call
both <- gintervals.neighbors.directional(tss, features,
    maxneighbors_upstream = 1,
    maxneighbors_downstream = 1,
    maxdist = 1000
)
print(both$upstream)
print(both$downstream)
```

`gintervals.normalize` *Normalize intervals to fixed or variable sizes*

Description

This function normalizes intervals by computing their centers and then expanding them to fixed or variable sizes, while ensuring they don't cross chromosome boundaries.

Usage

```
gintervals.normalize(intervals = NULL, size = NULL, intervals.set.out = NULL)
```

Arguments

`intervals`	intervals set
`size`	target size(s) for normalized intervals. Can be either:

- A single positive integer: all intervals normalized to this size
- A numeric vector: each interval normalized to its corresponding size. Vector length must exactly match the number of intervals, OR a single interval can be provided with multiple sizes to create multiple output intervals (one-to-many expansion).

`intervals.set.out` intervals set name where the function result is saved. If NULL, the result is returned to the user.

Value

Normalized intervals set with fixed or variable sizes, or NULL if result is saved to intervals.set.out

See Also

`gintervals.force_range`

Examples

```
gdb.init_examples()

# Single size (all intervals normalized to 500bp)
intervs <- gintervals(1, c(1000, 5000), c(2000, 6000))
gintervals.normalize(intervs, 500)

# Vector of sizes (each interval gets its own size)
intervs <- gintervals(1, c(1000, 3000, 5000), c(2000, 4000, 6000))
gintervals.normalize(intervs, c(500, 1000, 750))

# One-to-many: single interval with multiple sizes
interv <- gintervals(1, 1000, 2000)
gintervals.normalize(interv, c(500, 1000, 1500))
```

---

gintervals.path *Returns the path on disk of an interval set*

Description

Returns the path on disk of an interval set.

Usage

```
gintervals.path(intervals.set = NULL)
```

Arguments

`intervals.set` name of an interval set or a vector of interval set names

Details

This function returns the actual file system path where an interval set is stored. The function works with a single interval set name or a vector of names.

Value

A character vector containing the full paths to the interval sets on disk.

See Also

gintervals.exists, gintervals.ls, gtrack.path

Examples

```
gdb.init_examples()
gintervals.path("annotations")
gintervals.path(c("annotations", "coding"))
```

---

gintervals.quantiles *Calculates quantiles of a track expression for intervals*

Description

Calculates quantiles of a track expression for intervals.

Usage

```
gintervals.quantiles(
  expr = NULL,
  percentiles = 0.5,
  intervals = NULL,
  iterator = NULL,
  band = NULL,
  intervals.set.out = NULL
)
```

Arguments

expr	track expression for which quantiles are calculated
percentiles	an array of percentiles of quantiles in [0, 1] range
intervals	set of intervals
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expressions.
band	track expression band. If 'NULL' no band is used.
intervals.set.out	intervals set name where the function result is optionally outputted

Details

This function calculates quantiles of 'expr' for each interval in 'intervals'.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a set of intervals with additional columns representing quantiles for each percentile.

See Also

gquantiles, gbins.quantiles

Examples

```
gdb.init_examples()
intervs <- gintervals(c(1, 2), 0, 5000)
gintervals.quantiles("dense_track",
    percentiles = c(0.5, 0.3, 0.9), intervs
)
```

---

gintervals.random *Generate random genome intervals*

Description

Generate random genome intervals with a specified number of regions of a specified size. This function samples intervals uniformly across the genome, weighted by chromosome length.

Usage

```
gintervals.random(
  size,
  n,
  dist_from_edge = 3000000,
  chromosomes = NULL,
  filter = NULL
)
```

Arguments

size	The size of the intervals to generate (in base pairs)
n	The number of intervals to generate
dist_from_edge	The minimum distance from the edge of the chromosome for a region to start or end (default: 3e6)
chromosomes	The chromosomes to sample from (default: all chromosomes). Can be a character vector of chromosome names.
filter	A set of intervals to exclude from sampling (default: NULL). Generated intervals will not overlap with these regions.

Details

The function samples intervals randomly across the genome, with chromosomes weighted by their length. Each interval is guaranteed to:

- Be of the specified size
- Start and end at least dist_from_edge bases away from chromosome boundaries
- Fall entirely within a single chromosome
- Not overlap with any intervals in the filter (if provided)

When a filter is provided, the function pre-computes valid genome segments (regions not in the filter) and samples from these segments. Note that this can be slow when the filter contains many intervals.

The function uses R's random number generator, so set.seed() can be used for reproducibility.

This function is implemented in C++ for high performance and can generate millions of intervals quickly.

Value

A data.frame with columns chrom, start, and end representing genomic intervals

Examples

```
## Not run:
gdb.init_examples()

# Generate 1000 random intervals of 100bp
intervals <- gintervals.random(100, 1000)
head(intervals)

# Generate intervals only on chr1 and chr2
intervals <- gintervals.random(100, 1000, chromosomes = c("chr1", "chr2"))

# Generate intervals avoiding specific regions
filter_regions <- gintervals(c("chr1", "chr2"), c(1000, 5000), c(2000, 6000))
intervals <- gintervals.random(100, 1000, filter = filter_regions)

# Verify no overlaps with filter
```

```
overlaps <- gintervals.intersect(intervals, filter_regions)
nrow(overlaps) # Should be 0

# For reproducibility
set.seed(123)
intervals1 <- gintervals.random(100, 100)
set.seed(123)
intervals2 <- gintervals.random(100, 100)
identical(intervals1, intervals2) # TRUE

## End(Not run)
```

`gintervals.rbind` *Combines several sets of intervals*

Description

Combines several sets of intervals into one set.

Usage

```
gintervals.rbind(..., intervals.set.out = NULL)
```

Arguments

`...`	intervals sets to combine
`intervals.set.out`	intervals set name where the function result is optionally outputted
`intervals`	intervals set

Details

This function combines several intervals sets into one set. It works in a similar manner as 'rbind' yet it is faster. Also it supports intervals sets that are stored in files including the big intervals sets.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. If the format of the output intervals is set to be "big" (determined implicitly based on the result size and options), the order of the resulted intervals is altered as they are sorted by chromosome (or chromosomes pair - for 2D).

Value

If 'intervals.set.out' is 'NULL' a data frame combining intervals sets.

See Also

`gintervals`, `gintervals.2d`, `gintervals.canonic`

Examples

```
gdb.init_examples()

intervs1 <- gextract("sparse_track", gintervals(c(1, 2), 1000, 4000))
intervs2 <- gextract("sparse_track", gintervals(c(2, "X"), 2000, 5000))
gintervals.save("testintervs", intervs2)
gintervals.rbind(intervs1, "testintervs")
gintervals.rm("testintervs", force = TRUE)
```

---

gintervals.rm *Deletes a named intervals set*

Description

Deletes a named intervals set.

Usage

```
gintervals.rm(intervals.set = NULL, force = FALSE, db = NULL)
```

Arguments

intervals.set	name of an intervals set
force	if 'TRUE', suppresses user confirmation of a named intervals set removal
db	optional database path. When multiple databases are connected, this specifies which database to delete the intervals set from. If NULL (the default), the intervals set is deleted from the working database (GROOT).

Details

This function deletes a named intervals set from the Genomic Database. By default 'gintervals.rm' requires the user to interactively confirm the deletion. Set 'force' to 'TRUE' to suppress the user prompt.

Value

None.

See Also

gintervals.save, gintervals.exists, gintervals.ls, gintervals, gintervals.2d, gtrack.rm

Examples

```
gdb.init_examples()
intervs <- gintervals(c(1, 2))
gintervals.save("testintervs", intervs)
gintervals.ls()
gintervals.rm("testintervs", force = TRUE)
gintervals.ls()
```

`gintervals.save` *Creates a named intervals set*

Description

Saves intervals to a named intervals set.

Usage

```
gintervals.save(intervals.set.out = NULL, intervals = NULL)
```

Arguments

`intervals.set.out`	name of the new intervals set
`intervals`	intervals to save

Details

This function saves 'intervals' as a named intervals set.

Value

None.

See Also

`gintervals.rm`, `gintervals.load`, `gintervals.exists`, `gintervals.ls`, `gintervals`, `gintervals.2d`

Examples

```
gdb.init_examples()
intervs <- gintervals(c(1, 2))
gintervals.save("testintervs", intervs)
gintervals.ls()
gintervals.rm("testintervs", force = TRUE)
```

`gintervals.summary` *Calculates summary statistics of track expression for intervals*

Description

Calculates summary statistics of track expression for intervals.

Usage

```
gintervals.summary(
  expr = NULL,
  intervals = NULL,
  iterator = NULL,
  band = NULL,
  intervals.set.out = NULL
)
```

Arguments

`expr`	track expression
`intervals`	set of intervals
`iterator`	track expression iterator. If 'NULL' iterator is determined implicitly based on track expression.
`band`	track expression band. If 'NULL' no band is used.
`intervals.set.out`	intervals set name where the function result is optionally outputted

Details

This function returns summary statistics of a track expression for each interval 'intervals': total number of bins, total number of bins whose value is NaN, min, max, sum, mean and standard deviation of the values.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a set of intervals with additional columns representing summary statistics for each percentile and interval.

See Also

`gsummary`, `gbins.summary`

Examples

```
gdb.init_examples()
intervs <- gintervals(c(1, 2), 0, 5000)
gintervals.summary("dense_track", intervs)
```

`gintervals.to_mat` *Convert intervals + values data.frame to an interval-indexed matrix*

Description

Builds a numeric matrix of value columns whose rows are indexed by intervals. The intervals are carried in `attr(mat, "intervals")` as the authoritative identity; `rownames(mat)` are display-only (default `"chrom:start-end"`) and are NEVER parsed back by `gintervals.from_mat`. This avoids the round-trip corruption that occurs when chrom names contain underscores or other separators.

Usage

```
gintervals.to_mat(df, id_col = NULL, value_cols = NULL, labels = TRUE)
```

Arguments

`df`	data.frame with `chrom`, `start`, `end` and zero or more value columns. May contain an `intervalID` column (from `gextract`), which is kept in the attribute but excluded from the matrix.
`id_col`	optional column name in `df` whose values become rownames. If `NULL` (default), rownames are `"chrom:start-end"`.
`value_cols`	character vector of column names to use as matrix data. If `NULL` (default), auto-detect: all columns except `chrom`, `start`, `end`, `intervalID`. Auto-detect errors if any selected column is non-numeric; pass `value_cols` explicitly to override.
`labels`	if `TRUE` (default), set `rownames(mat)` to either `df[[id_col]]` (if `id_col` is supplied) or `"chrom:start-end"`. If `FALSE`, leave rownames `NULL` - useful in pipelines that don't need the display labels and would prefer to skip the construction cost on large inputs. When `FALSE`, the `id_col` argument is ignored.

Value

An `intervs_mat` object: a numeric matrix subclass with the intervals attached as `attr(., "intervals")`. Supports row/column subsetting (`[`) and `rbind()` while preserving the attribute.

See Also

`gintervals.from_mat`

Examples

```
df <- data.frame(
    chrom = c("chr1", "chr1", "chr2"),
    start = c(100L, 500L, 200L),
    end   = c(200L, 700L, 400L),
    t1    = c(1.5, 2.5, 3.5),
    t2    = c(10, 20, 30)
)
mat <- gintervals.to_mat(df)
rownames(mat)
# subset preserves intervals:
sub <- mat[c(1, 3), ]
attr(sub, "intervals")
# round-trip back to a data.frame:
gintervals.from_mat(sub)
```

---

gintervals.union *Calculates a union of two sets of intervals*

Description

Calculates a union of two sets of intervals.

Usage

```
gintervals.union(
  intervals1 = NULL,
  intervals2 = NULL,
  intervals.set.out = NULL
)
```

Arguments

`intervals1`, `intervals2`
: set of one-dimensional intervals

`intervals.set.out`
: intervals set name where the function result is optionally outputted

Details

This function returns intervals that represent a genomic space covered by either 'intervals1' or 'intervals2'.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a data frame representing the union of intervals.

See Also

gintervals.intersect, gintervals.diff, gintervals, gintervals.2d

Examples

```
gdb.init_examples()

intervs1 <- gscreen("dense_track > 0.15 & dense_track < 0.18")
intervs2 <- gscreen("dense_track >= 0.18 & dense_track < 0.2")

## 'intervs3' and 'intervs4' are identical
intervs3 <- gintervals.union(intervs1, intervs2)
intervs4 <- gscreen("dense_track > 0.15 & dense_track < 0.2")
```

---

gintervals.update *Updates a named intervals set*

Description

Updates a named intervals set.

Usage

```
gintervals.update(
  intervals.set = NULL,
  intervals = "",
  chrom = NULL,
  chrom1 = NULL,
  chrom2 = NULL
)
```

Arguments

intervals.set	name of an intervals set
intervals	intervals or 'NULL'
chrom	chromosome for 1D intervals set
chrom1	first chromosome for 2D intervals set
chrom2	second chromosome for 2D intervals set

Details

This function replaces all intervals of given chromosome (or chromosome pair) within 'intervals.set' with 'intervals'. Chromosome is specified by 'chrom' for 1D intervals set or 'chrom1', 'chrom2' for 2D intervals set.

If 'intervals' is 'NULL' all intervals of given chromosome are removed from 'intervals.set'.

Value

None.

See Also

gintervals.save, gintervals.load, gintervals.exists, gintervals.ls

Examples

```
gdb.init_examples()
intervs <- gscreen(
    "sparse_track > 0.2",
    gintervals(c(1, 2), 0, 10000)
)
gintervals.save("testintervs", intervs)
gintervals.load("testintervs")
gintervals.update("testintervs", intervs[intervs$chrom == "chr2", ][1:5, ], chrom = 2)
gintervals.load("testintervs")
gintervals.update("testintervs", NULL, chrom = 2)
gintervals.load("testintervs")
gintervals.rm("testintervs", force = TRUE)
```

---

giterator.cartesian_grid

Creates a cartesian-grid iterator

Description

Creates a cartesian grid two-dimensional iterator that can be used by any function that accepts an iterator argument.

Usage

```
giterator.cartesian_grid(
  intervals1 = NULL,
  expansion1 = NULL,
  intervals2 = NULL,
  expansion2 = NULL,
  min.band.idx = NULL,
  max.band.idx = NULL
)
```

Arguments

intervals1	one-dimensional intervals
expansion1	an array of integers that define expansion around intervals1 centers
intervals2	one-dimensional intervals. If 'NULL' then 'intervals2' is considered to be equal to 'intervals1'
expansion2	an array of integers that define expansion around intervals2 centers. If 'NULL' then 'expansion2' is considered to be equal to 'expansion1'
min.band.idx, max.band.idx	integers that limit iterator intervals to band

Details

This function creates and returns a cartesian grid two-dimensional iterator that can be used by any function that accepts an iterator argument.

Assume 'centers1' and 'centers2' to be the central points of each interval from 'intervals1' and 'intervals2', and 'C1', 'C2' to be two points from 'centers1', 'centers2' accordingly. Assume also that the values in 'expansion1' and 'expansion2' are unique and sorted.

'giterator.cartesian_grid' creates a set of all possible unique and non-overlapping two-dimensional intervals of form: '(chrom1, start1, end1, chrom2, start2, end2)'. Each '(chrom1, start1, end1)' is created by taking a point 'C1' - '(chrom1, coord1)' and converting it to 'start1' and 'end1' such that 'start1 == coord1+E1[i]', 'end1 == coord1+E1[i+1]', where 'E1[i]' is one of the sorted 'expansion1' values. Overlaps between rectangles or expansion beyond the limits of chromosome are avoided.

'min.band.idx' and 'max.band.idx' parameters control whether a pair of 'C1' and 'C2' is skipped or not. If both of these parameters are not 'NULL' AND if both 'C1' and 'C2' share the same chromosome AND the delta of indices of 'C1' and 'C2' ('C1 index - C2 index') lays within '[min.band.idx, max.band.idx]' range - only then the pair will be used to create the intervals. Otherwise 'C1-C2' pair is filtered out. Note: if 'min.band.idx' and 'max.band.idx' are not 'NULL', i.e. band indices filtering is applied, then 'intervals2' parameter must be set to 'NULL'.

Value

A list containing the definition of cartesian iterator.

See Also

giterator.intervals

Examples

```
gdb.init_examples()

intervs1 <- gintervals(
    c(1, 1, 2), c(100, 300, 200),
    c(300, 500, 300)
)
```

```
intervs2 <- gintervals(
    c(1, 2, 2), c(400, 1000, 3000),
    c(800, 2000, 4000)
)
itr <- giterator.cartesian_grid(
    intervs1, c(-20, 100), intervs2,
    c(-40, -10, 50)
)
giterator.intervals(iterator = itr)

itr <- giterator.cartesian_grid(intervs1, c(-20, 50, 100))
giterator.intervals(iterator = itr)

itr <- giterator.cartesian_grid(intervs1, c(-20, 50, 100),
    min.band.idx = -1,
    max.band.idx = 0
)
giterator.intervals(iterator = itr)
```

`giterator.intervals` *Returns iterator intervals*

Description

Returns iterator intervals given track expression, scope, iterator and band.

Usage

```
giterator.intervals(
  expr = NULL,
  intervals = .misha$ALLGENOME,
  iterator = NULL,
  band = NULL,
  intervals.set.out = NULL,
  interval_relative = FALSE,
  partial_bins = c("clip", "exact", "drop")
)
```

Arguments

`expr`	track expression
`intervals`	genomic scope
`iterator`	track expression iterator. If 'NULL' iterator is determined implicitly based on track expression.
`band`	track expression band. If 'NULL' no band is used.
`intervals.set.out`	intervals set name where the function result is optionally outputted

`interval_relative`
: if TRUE, and iterator is numeric, bins start at each interval's start position instead of chromosome position 0. Returns intervalID column. Default: FALSE.

`partial_bins`
: how to handle partial bins at interval boundaries when interval_relative is TRUE. One of "clip" (default, truncate last bin to interval boundary), "exact" or "drop" (only output full-size bins).

Details

This function returns a set of intervals used by the iterator intervals for the given track expression, genomic scope, iterator and band. Some functions accept an iterator without accepting a track expression (like 'gtrack.create_pwm_energy'). These functions generate the values for each iterator interval by themselves. Use set 'expr' to 'NULL' to simulate the work of these functions.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

When 'interval_relative' is TRUE, bins are aligned to each input interval's start position rather than chromosome position 0. This mode requires a numeric iterator (binsize) and returns an additional 'intervalID' column indicating which input interval spawned each bin.

Value

If 'intervals.set.out' is 'NULL' a data frame representing iterator intervals. When 'interval_relative' is TRUE, includes an 'intervalID' column.

See Also

giterator.cartesian_grid

Examples

```
gdb.init_examples()

## iterator is set implicitly to bin size of 'dense' track
giterator.intervals("dense_track", gintervals(1, 0, 200))

## iterator = 30
giterator.intervals("dense_track", gintervals(1, 0, 200), 30)

## iterator is an intervals set named 'annotations'
giterator.intervals("dense_track", .misha$ALLGENOME, "annotations")

## iterator is set implicitly to intervals of 'array_track' track
giterator.intervals("array_track", gintervals(1, 0, 200))

## iterator is a rectangle 100000 by 50000
giterator.intervals(
    "rects_track",
    gintervals.2d(chroms1 = 1, chroms2 = "chrX"),
    c(100000, 50000)
```

```
)

## interval_relative mode: bins aligned to each interval's start
intervs <- gintervals(1, c(100, 500), c(300, 700))
giterator.intervals(NULL, intervs, iterator = 50, interval_relative = TRUE)
```

---

glookup *Returns values from a lookup table based on track expression*

Description

Evaluates track expression and translates the values into bin indices that are used in turn to retrieve and return values from a lookup table.

Usage

```
glookup(
  lookup_table = NULL,
  ...,
  intervals = NULL,
  include.lowest = FALSE,
  force.binning = TRUE,
  iterator = NULL,
  band = NULL,
  intervals.set.out = NULL
)
```

Arguments

lookup_table	a multi-dimensional array containing the values that are returned by the function
...	pairs of 'expr', 'breaks' where 'expr' is a track expression and the breaks determine the bin
intervals	genomic scope for which the function is applied
include.lowest	if 'TRUE', the lowest value of the range determined by breaks is included
force.binning	if 'TRUE', the values smaller than the minimal break will be translated to index 1, and the values that exceed the maximal break will be translated to index N-1 where N is the number of breaks. If 'FALSE' the out-of-range values will produce NaN values.
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expressions.
band	track expression band. If 'NULL' no band is used.
intervals.set.out	intervals set name where the function result is optionally outputted

Details

This function evaluates the track expression for all iterator intervals and translates this value into an index based on the breaks. This index is then used to address the lookup table and return the according value. More than one 'expr'-'breaks' pair can be used. In that case 'lookup_table' is addressed in a multidimensional manner, i.e. 'lookup_table[i1, i2, ...]'.

The range of bins is determined by 'breaks' argument. For example: 'breaks = c(x1, x2, x3, x4)' represents three different intervals (bins): (x1, x2], (x2, x3], (x3, x4].

If 'include.lowest' is 'TRUE' then the lowest value is included in the first interval, i.e. in [x1, x2].

'force.binning' parameter controls what should be done when the value of 'expr' exceeds the range determined by 'breaks'. If 'force.binning' is 'TRUE' then values smaller than the minimal break will be translated to index 1, and the values exceeding the maximal break will be translated to index 'M-1' where 'M' is the number of breaks. If 'force.binning' is 'FALSE' the out-of-range values will produce 'NaN' values.

Regardless of 'force.binning' value if the value of 'expr' is 'NaN' then result is 'NaN' too.

The order inside the result might not be the same as the order of intervals. Use 'intervalID' column to refer to the index of the original interval from the supplied 'intervals'.

If 'intervals.set.out' is not 'NULL' the result (without 'intervalID' column) is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a set of intervals with additional 'value' and 'intervalID' columns.

See Also

gtrack.lookup, gextract, gpartition, gdist

Examples

```
gdb.init_examples()

## one-dimensional lookup table
breaks1 <- seq(0.1, 0.2, length.out = 6)
glookup(1:5, "dense_track", breaks1, gintervals(1, 0, 200))

## two-dimensional lookup table
t <- array(1:15, dim = c(5, 3))
breaks2 <- seq(0.31, 0.37, length.out = 4)
glookup(
    t, "dense_track", breaks1, "2 * dense_track", breaks2,
    gintervals(1, 0, 200)
)
```

gpartition *Partitions the values of track expression*

Description

Converts the values of track expression to intervals that match corresponding bin.

Usage

```
gpartition(
  expr = NULL,
  breaks = NULL,
  intervals = NULL,
  include.lowest = FALSE,
  iterator = NULL,
  band = NULL,
  intervals.set.out = NULL
)
```

Arguments

expr	track expression
breaks	breaks that determine the bin
intervals	genomic scope for which the function is applied
include.lowest	if 'TRUE', the lowest value of the range determined by breaks is included
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expression.
band	track expression band. If 'NULL' no band is used.
intervals.set.out	intervals set name where the function result is optionally outputted

Details

This function converts first the values of track expression into 1-based bin's index according 'breaks' argument. It returns then the intervals with the corresponding bin's index.

The range of bins is determined by 'breaks' argument. For example: 'breaks=c(x1, x2, x3, x4)' represents three different intervals (bins): (x1, x2], (x2, x3], (x3, x4].

If 'include.lowest' is 'TRUE' the the lowest value will be included in the first interval, i.e. in [x1, x2].

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a set of intervals with an additional column that indicates the corresponding bin index.

See Also

gscreen, gextract, glookup, gdist

Examples

```
gdb.init_examples()
breaks <- seq(0, 0.2, by = 0.05)
gpartition("dense_track", breaks, gintervals(1, 0, 5000))
```

---

gquantiles *Calculates quantiles of a track expression*

Description

Calculates the quantiles of a track expression for the given percentiles.

Usage

```
gquantiles(
  expr = NULL,
  percentiles = 0.5,
  intervals = get("ALLGENOME", envir = .misha),
  iterator = NULL,
  band = NULL
)
```

Arguments

expr	track expression
percentiles	an array of percentiles of quantiles in [0, 1] range
intervals	genomic scope for which the function is applied
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expression.
band	track expression band. If 'NULL' no band is used.

Details

This function calculates the quantiles for the given percentiles.

If data size exceeds the limit (see: 'getOption(gmax.data.size)'), the data is randomly sampled to fit the limit. A warning message is generated. Call `set.seed()` before this function to make the sample reproducible.

Note: this function is capable to run in multitasking mode. Sampling may vary according to the extent of multitasking. Since multitasking depends on the number of available CPU cores, running the function on two different machines might give different results. Please switch off multitasking if you want to achieve identical results on any machine. For more information regarding multitasking please refer "User Manual".

Value

An array that represent quantiles.

NaN values

A track expression evaluates to NaN wherever the iterator produces a bin the track has no data for. What happens next depends on the function:

- gextract **keeps** NaN rows, so the result has one row per iterator interval whether or not the track covered it.
- gsummary **counts** them and reports the count as the "NaN intervals" element, while the statistics themselves are computed over the non-NaN values only.
- gdist, gquantiles and gscreen **drop** them: NaN bins are not counted into any distribution bin, do not contribute to a percentile, and never satisfy a screening condition - including a condition that would be true of every real value.
- gsegment **spans** them: a NaN bin contributes no evidence to the test that places a boundary, but it still falls inside whichever segment surrounds it, so the returned segments tile the scope continuously rather than skipping the gaps.

So on 20 bins of which 7 are NaN, gextract returns 20 rows, gsummary reports 20 total and 7 NaN, and gdist counts 13; and on a 300 kb scope where 120 of 300 bins are NaN, gsegment still returns segments covering the full 300 kb.

The practical consequence is that NaN and zero are different, and collapsing them with ifelse(is.na(x), 0, x) turns "no data here" into a measured value of zero. Where that is genuinely what you want, note that it also changes every mean, quantile and distribution computed downstream.

See Also

gbins.quantiles, gintervals.quantiles, gdist

Examples

```
gdb.init_examples()
gquantiles("dense_track", c(0.1, 0.6, 0.8), gintervals(c(1, 2)))
```

---

grevcomp *Get reverse complement of DNA sequence*

Description

Takes a DNA sequence string and returns its reverse complement.

Usage

```
grevcomp(seq)
```

Arguments

seq	A character vector containing DNA sequences (using A,C,G,T). Ignores other characters and NA values.

Value

A character vector of the same length as the input, containing the reverse complement sequences

Examples

```
grevcomp("ACTG") # Returns "CAGT"
grevcomp(c("ACTG", "GGCC")) # Returns c("CAGT", "GGCC")
grevcomp(c("ACTG", NA, "GGCC")) # Returns c("CAGT", NA, "GGCC")
```

---

gsample *Returns samples from the values of track expression*

Description

Returns a sample of the specified size from the values of track expression.

Usage

```
gsample(expr = NULL, n = NULL, intervals = NULL, iterator = NULL, band = NULL)
```

Arguments

expr	track expression
n	a number of items to choose
intervals	genomic scope for which the function is applied
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expression.
band	track expression band. If 'NULL' no band is used.

Details

This function returns a sample of the specified size from the values of track expression. If 'n' is less than the total number of values, the data is randomly sampled. Call `set.seed()` before this function to make the sample reproducible.

If 'n' is higher than the total number of values, all values are returned (yet reshuffled).

Value

An array that represent quantiles.

See Also

gextract

Examples

```
gdb.init_examples()
gsample("sparse_track", 10)
```

---

gscreen *Finds intervals that match track expression*

Description

Finds all intervals where track expression is 'TRUE'.

Usage

```
gscreen(
  expr = NULL,
  intervals = NULL,
  iterator = NULL,
  band = NULL,
  intervals.set.out = NULL
)
```

Arguments

expr	logical track expression
intervals	genomic scope for which the function is applied
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expression.
band	track expression band. If 'NULL' no band is used.
intervals.set.out	intervals set name where the function result is optionally outputted

Details

This function finds all intervals where track expression's value is 'TRUE'.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a set of intervals that match track expression.

NaN values

A track expression evaluates to NaN wherever the iterator produces a bin the track has no data for. What happens next depends on the function:

- gextract **keeps** NaN rows, so the result has one row per iterator interval whether or not the track covered it.
- gsummary **counts** them and reports the count as the "NaN intervals" element, while the statistics themselves are computed over the non-NaN values only.
- gdist, gquantiles and gscreen **drop** them: NaN bins are not counted into any distribution bin, do not contribute to a percentile, and never satisfy a screening condition - including a condition that would be true of every real value.
- gsegment **spans** them: a NaN bin contributes no evidence to the test that places a boundary, but it still falls inside whichever segment surrounds it, so the returned segments tile the scope continuously rather than skipping the gaps.

So on 20 bins of which 7 are NaN, `gextract` returns 20 rows, `gsummary` reports 20 total and 7 NaN, and `gdist` counts 13; and on a 300 kb scope where 120 of 300 bins are NaN, `gsegment` still returns segments covering the full 300 kb.

The practical consequence is that NaN and zero are different, and collapsing them with `ifelse(is.na(x), 0, x)` turns "no data here" into a measured value of zero. Where that is genuinely what you want, note that it also changes every mean, quantile and distribution computed downstream.

See Also

gsegment, gextract

Examples

```
gdb.init_examples()
gscreen("dense_track > 0.2 & sparse_track < 0.4",
    iterator = "dense_track"
)
```

---

gsegment *Divides track expression into segments*

Description

Divides the values of track expression into segments by using Wilcoxon test.

Usage

```
gsegment(
  expr = NULL,
  minsegment = NULL,
  maxpval = 0.05,
  onetailed = TRUE,
  intervals = NULL,
  iterator = NULL,
  intervals.set.out = NULL
)
```

Arguments

`expr`	track expression
`minsegment`	minimal segment size
`maxpval`	maximal P-value that separates two adjacent segments
`onetailed`	if 'TRUE', Wilcoxon test is performed one tailed, otherwise two tailed
`intervals`	genomic scope for which the function is applied
`iterator`	track expression iterator of "fixed bin" type. If 'NULL' iterator is determined implicitly based on track expression.
`intervals.set.out`	intervals set name where the function result is optionally outputted

Details

This function divides the values of track expression into segments, where each segment size is at least of 'minsegment' size and the P-value of comparing the segment with the first 'minsegment' values from the next segment is at most 'maxpval'. Comparison is done using Wilcoxon (also known as Mann-Whitney) test.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a set of intervals where each interval represents a segment.

NaN values

A track expression evaluates to `NaN` wherever the iterator produces a bin the track has no data for. What happens next depends on the function:

- `gextract` **keeps** `NaN` rows, so the result has one row per iterator interval whether or not the track covered it.
- `gsummary` **counts** them and reports the count as the "NaN intervals" element, while the statistics themselves are computed over the non-`NaN` values only.

- gdist, gquantiles and gscreen **drop** them: NaN bins are not counted into any distribution bin, do not contribute to a percentile, and never satisfy a screening condition - including a condition that would be true of every real value.
- gsegment **spans** them: a NaN bin contributes no evidence to the test that places a boundary, but it still falls inside whichever segment surrounds it, so the returned segments tile the scope continuously rather than skipping the gaps.

So on 20 bins of which 7 are NaN, gextract returns 20 rows, gsummary reports 20 total and 7 NaN, and gdist counts 13; and on a 300 kb scope where 120 of 300 bins are NaN, gsegment still returns segments covering the full 300 kb.

The practical consequence is that NaN and zero are different, and collapsing them with ifelse(is.na(x), 0, x) turns "no data here" into a measured value of zero. Where that is genuinely what you want, note that it also changes every mean, quantile and distribution computed downstream.

See Also

gscreen, gwilcox

Examples

```
gdb.init_examples()
gsegment("dense_track", 5000, 0.0001)
```

---

gseq.comp *Complement DNA sequence*

Description

Takes a DNA sequence string and returns its complement (without reversing).

Usage

```
gseq.comp(seq)
```

Arguments

seq	A character vector containing DNA sequences (using A,C,G,T). Preserves case and handles NA values.

Value

A character vector of the same length as the input, containing the complemented sequences

See Also

gseq.revcomp, gseq.rev

Examples

```
gseq.comp("ACTG") # Returns "TGAC"
gseq.comp(c("ACTG", "GGCC")) # Returns c("TGAC", "CCGG")
gseq.comp(c("ACTG", NA, "GGCC")) # Returns c("TGAC", NA, "CCGG")
```

---

gseq.extract *Returns DNA sequences*

Description

Returns DNA sequences for given intervals

Usage

```
gseq.extract(intervals = NULL)
```

Arguments

intervals	intervals for which DNA sequence is returned

Details

This function returns an array of sequence strings for each interval from 'intervals'. If intervals contain an additional 'strand' column and its value is '-1', the reverse-complementary sequence is returned.

Each interval costs one disk read, so large interval sets are dominated by I/O latency rather than by the sequence data itself. When the sequence files are not in the page cache, the work is split across several processes (capped by 'gmax.processes', and in practice usually well below it). The decision is made by timing the first few reads and extrapolating over the number of read-ahead windows the intervals span: cached data, and interval sets dense enough that many intervals share a read, stay in a single process because distributing them would only add fork overhead.

Sets of fewer than 500 intervals are never distributed. Named intervals sets stored in the "big" format are always read sequentially. Distributing roughly doubles peak memory while the result is being assembled, so a call close to 'gmax.data.size' may need a lower limit or 'gmultitasking = FALSE'. Note also that this function forks: calling it from inside your own 'mclapply()' or similar multiplies the process count, as it does for the other multitasking misha functions.

Set 'gseq.extract.probe.usec' to override the per-interval microsecond threshold at which distributing kicks in; 0 always distributes and also requests the maximum number of processes, a large value never distributes. 'gmultitasking = FALSE' disables it entirely.

Value

An array of character strings representing DNA sequence.

See Also

gextract

Examples

```
gdb.init_examples()
intervs <- gintervals(c(1, 2), 10000, 10020)
gseq.extract(intervs)
```

---

gseq.kmer *Score DNA sequences with a k-mer over a region of interest*

Description

Counts exact matches of a k-mer in DNA sequences over a specified region of interest (ROI). The ROI is defined by start_pos and end_pos (1-based, inclusive), with optional extension controlled by extend.

Usage

```
gseq.kmer(
  seqs,
  kmer,
  mode = c("count", "frac"),
  strand = 0L,
  start_pos = NULL,
  end_pos = NULL,
  extend = FALSE,
  skip_gaps = TRUE,
  gap_chars = c("-", ".")
)
```

Arguments

seqs	character vector of DNA sequences (A/C/G/T/N; case-insensitive)
kmer	single character string containing the k-mer to search for (A/C/G/T only)
mode	character; one of "count" or "frac"
strand	integer; 1=forward, -1=reverse, 0=both strands (default: 0)
start_pos	integer or NULL; 1-based inclusive start of ROI (default: 1)
end_pos	integer or NULL; 1-based inclusive end of ROI (default: sequence length)
extend	logical or integer; extension of allowed window starts (default: FALSE)
skip_gaps	logical; if TRUE, treat gap characters as holes and skip them while scanning. Windows are k consecutive non-gap bases (default: TRUE)
gap_chars	character vector; which characters count as gaps (default: c("-", "."))

Details

This function counts k-mer occurrences in DNA sequences directly without requiring a genomics database. For detailed documentation on k-mer counting parameters, see `gvtrack.create` (functions "kmer.count" and "kmer.frac").

The ROI (region of interest) is defined by `start_pos` and `end_pos`. The `extend` parameter controls whether k-mer matches can extend beyond the ROI boundaries. For palindromic k-mers, use `strand=1` or `-1` to avoid double counting.

When `skip_gaps=TRUE`, characters specified in `gap_chars` are treated as gaps. Windows are defined as k consecutive non-gap bases. The `frac` denominator counts the number of possible logical starts (non-gap windows) in the region. `start_pos` and `end_pos` are interpreted as physical coordinates on the full sequence.

Value

Numeric vector with counts (for "count" mode) or fractions (for "frac" mode). Returns 0 when sequence is too short or ROI is invalid.

See Also

`gvtrack.create` for detailed k-mer parameter documentation

Examples

```
## Not run:
# Example sequences
seqs <- c("CGCGCGCGCG", "ATATATATAT", "ACGTACGTACGT")

# Count CG dinucleotides on both strands
gseq.kmer(seqs, "CG", mode = "count", strand = 0)

# Count on forward strand only
gseq.kmer(seqs, "CG", mode = "count", strand = 1)

# Get CG fraction
gseq.kmer(seqs, "CG", mode = "frac", strand = 0)

# Count in a specific region
gseq.kmer(seqs, "CG", mode = "count", start_pos = 2, end_pos = 8)

# Allow k-mer to extend beyond ROI boundaries
gseq.kmer(seqs, "CG", mode = "count", start_pos = 2, end_pos = 8, extend = TRUE)

# Calculate GC content by summing G and C fractions
g_frac <- gseq.kmer(seqs, "G", mode = "frac", strand = 1)
c_frac <- gseq.kmer(seqs, "C", mode = "frac", strand = 1)
gc_content <- g_frac + c_frac
gc_content

# Compare AT counts on different strands
at_forward <- gseq.kmer(seqs, "AT", mode = "count", strand = 1)
```

```
at_reverse <- gseq.kmer(seqs, "AT", mode = "count", strand = -1)
at_both <- gseq.kmer(seqs, "AT", mode = "count", strand = 0)
data.frame(forward = at_forward, reverse = at_reverse, both = at_both)

## End(Not run)
```

`gseq.kmer.dist` *Compute k-mer distribution in genomic intervals*

Description

Counts the occurrence of all k-mers (of size k) within the specified genomic intervals, optionally excluding masked regions.

Usage

```
gseq.kmer.dist(intervals, k = 6L, mask = NULL)
```

Arguments

`intervals`	Genomic intervals to analyze
`k`	Integer k-mer size (1-10). Default is 6.
`mask`	Optional intervals to exclude from counting. Positions within the mask will not contribute to k-mer counts.

Value

A data frame with columns:

kmer Character string representing the k-mer sequence

count Number of occurrences of this k-mer

Only k-mers with count > 0 are included. K-mers containing N bases are not counted.

See Also

`gseq.extract`, `gseq.kmer`

Examples

```
gdb.init_examples()

# Count all 6-mers in first 10kb of chr1
intervals <- data.frame(chrom = "chr1", start = 0, end = 10000)
kmer_dist <- gseq.kmer.dist(intervals, k = 6)
head(kmer_dist)

# Count dinucleotides
```

```
dinucs <- gseq.kmer.dist(intervals, k = 2)
dinucs

# Count with mask
mask <- data.frame(chrom = "chr1", start = 5000, end = 6000)
kmer_dist_masked <- gseq.kmer.dist(intervals, k = 6, mask = mask)
```

`gseq.pwm` *Score DNA sequences with a PWM over a region of interest*

Description

Scores full DNA sequences using a Position Weight Matrix (PWM) over a specified region of interest (ROI). The ROI is defined by `start_pos` and `end_pos` (1-based, inclusive), with optional extension controlled by `extend`. All reported positions are on the full input sequence.

Usage

```
gseq.pwm(
  seqs,
  pssm,
  mode = c("lse", "max", "pos", "count"),
  bidirect = TRUE,
  strand = 0L,
  score.thresh = NULL,
  start_pos = NULL,
  end_pos = NULL,
  extend = FALSE,
  spat.factor = NULL,
  spat.bin = 1L,
  spat.min = NULL,
  spat.max = NULL,
  return_strand = FALSE,
  skip_gaps = TRUE,
  gap_chars = c("-", "."),
  neutral_chars = c("N", "n", "*"),
  neutral_chars_policy = c("average", "log_quarter", "na"),
  prior = 0.01
)
```

Arguments

`seqs`	character vector of DNA sequences (A/C/G/T/N; case-insensitive)
`pssm`	numeric matrix or data frame with columns named A, C, G, T (additional columns are allowed and will be ignored)
`mode`	character; one of "lse", "max", "pos", or "count"

bidirect	logical; if TRUE, scans both strands (default: TRUE)
strand	integer; 1=forward, -1=reverse, 0=both strands (default: 0)
score.thresh	single number; windows scoring at or above this value are counted. Required when mode="count" and ignored otherwise. PWM scores are log-likelihoods, so the usable range depends on the PSSM, the prior and any spatial weights - calibrate with mode="max" before choosing one.
start_pos	integer or NULL; 1-based inclusive start of ROI (default: 1)
end_pos	integer or NULL; 1-based inclusive end of ROI (default: sequence length)
extend	logical or integer; extension of allowed window starts (default: FALSE)
spat.factor	numeric vector; spatial weighting factors (optional)
spat.bin	integer; bin size for spatial weighting
spat.min	numeric; start of scanning window
spat.max	numeric; end of scanning window
return_strand	logical; if TRUE and mode="pos", returns data.frame with pos and strand columns
skip_gaps	logical; if TRUE, treat gap characters as holes and skip them while scanning. Windows are w consecutive non-gap bases (default: TRUE)
gap_chars	character vector; which characters count as gaps (default: c("-", "."))
neutral_chars	character vector; bases treated as unknown and scored with the average log probability per position (default: c("N", "n", "*"))
neutral_chars_policy	character string; how to treat neutral characters. One of "average" (default; use the column's mean log-probability), "log_quarter" (always use log(1/4)), or "na" (return NA when a neutral character is encountered in the scanning window).
prior	numeric; pseudocount added to frequencies (default: 0.01). Set to 0 for no pseudocounts.

Details

This function scores DNA sequences directly without requiring a genomics database. For detailed documentation on PWM scoring modes, parameters, and spatial weighting, see gvtrack.create (functions "pwm", "pwm.max", "pwm.max.pos", "pwm.count").

The ROI (region of interest) is defined by start_pos and end_pos. The extend parameter controls whether motif matches can extend beyond the ROI boundaries.

When skip_gaps=TRUE, characters specified in gap_chars are treated as gaps. Windows are defined as w consecutive non-gap bases. All positions (pos) are reported as 1-based indices on the original full sequence (including gaps). start_pos and end_pos are interpreted as physical coordinates on the full sequence.

Neutral characters (neutral_chars, default c("N", "n", "*")) are treated as unknown bases in both orientations. Each neutral contributes the mean log-probability of the corresponding PSSM column, yielding identical penalties on forward and reverse strands without hard-coded background scores. In mode = "max" the reported value is the single best strand score after applying any spatial weights; forward and reverse contributions are not aggregated. This matches the default behavior of the PWM virtual tracks (pwm.max, pwm.max.pos, etc.).

Value

Numeric vector (for "lse"/"max"/"count" modes), integer vector (for "pos" mode), or data.frame with pos and strand columns (for "pos" mode with return_strand=TRUE). Returns NA when no valid windows exist.

See Also

gvtrack.create for detailed PWM parameter documentation

Examples

```
## Not run:
# Create a PSSM (position-specific scoring matrix) with frequency values
pssm <- matrix(
    c(
        0.7, 0.1, 0.1, 0.1, # Position 1: mostly A
        0.1, 0.7, 0.1, 0.1, # Position 2: mostly C
        0.1, 0.1, 0.7, 0.1, # Position 3: mostly G
        0.1, 0.1, 0.1, 0.7 # Position 4: mostly T
    ),
    ncol = 4, byrow = TRUE
)
colnames(pssm) <- c("A", "C", "G", "T")

# Example sequences
seqs <- c("ACGTACGTACGT", "GGGGACGTCCCC", "TTTTTTTTTTT")

# Score sequences using log-sum-exp (default mode)
gseq.pwm(seqs, pssm, mode = "lse")

# Get maximum score
gseq.pwm(seqs, pssm, mode = "max")

# Find position of best match
gseq.pwm(seqs, pssm, mode = "pos")

# Find position with strand information
gseq.pwm(seqs, pssm, mode = "pos", bidirect = TRUE, return_strand = TRUE)

# Count matches above threshold. score.thresh is mandatory in "count" mode
# and has no default: PWM scores are log-likelihoods, so calibrate it
# against the score range of your own PSSM.
range(gseq.pwm(seqs, pssm, mode = "max"))
gseq.pwm(seqs, pssm, mode = "count", score.thresh = -3)

# Score only a region of interest
gseq.pwm(seqs, pssm, mode = "max", start_pos = 3, end_pos = 10)

# Allow matches to extend beyond ROI boundaries
gseq.pwm(seqs, pssm, mode = "count", score.thresh = -3, start_pos = 5, end_pos = 8, extend = TRUE)

# Spatial weighting example: higher weight in the center
```

```
spatial_weights <- c(0.5, 1.0, 2.0, 1.0, 0.5)
gseq.pwm(seqs, pssm,
    mode = "lse",
    spat.factor = spatial_weights,
    spat.bin = 2
)

## End(Not run)
```

`gseq.pwm_edits` *Show optimal edits to reach a PWM score threshold*

Description

For each input sequence (or genomic interval), finds the optimal motif window and the specific base changes needed to reach `score.thresh`. Returns a long-format data frame with one row per edit.

Usage

```
gseq.pwm_edits(
  seqs,
  pssm,
  score.thresh,
  max_edits = NULL,
  max_indels = NULL,
  bidirect = TRUE,
  prior = 0.01,
  score.min = NULL,
  score.max = NULL,
  extend = TRUE,
  strand = 1L,
  direction = "above"
)
```

Arguments

`seqs`	character vector of DNA sequences, OR a data frame of genomic intervals (with columns chrom, start, end). When intervals are provided, sequences are extracted automatically via `gseq.extract`.
`pssm`	numeric matrix or data frame with columns A, C, G, T. Each row is a motif position.
`score.thresh`	single number; target PWM log-likelihood score to reach.
`max_edits`	integer or NULL; maximum number of edits to search. NULL means no cap. Default NULL.

max_indels	integer or NULL; maximum number of insertions and deletions allowed (default NULL, substitutions only). When > 0, a banded Needleman-Wunsch DP is used. Edits are reported with edit_type "sub", "ins", or "del".
bidirect	logical; scan both strands? Default TRUE.
prior	numeric; pseudocount for PSSM frequencies. Default 0.01.
score.min	numeric or NULL; skip windows with PWM score below this. Default NULL (no filter).
score.max	numeric or NULL; skip windows with PWM score above this. Default NULL (no filter).
extend	logical or integer; extend sequence for boundary motifs. Default TRUE.
strand	integer; which strand to scan when bidirect=FALSE. 1=forward, -1=reverse. Default 1.
direction	character; direction of the edit distance query. "above" (default) finds minimum edits to bring score above score.thresh; "below" finds minimum edits to bring score below score.thresh.

Details

This is an investigation tool: use it on a small set of positions (e.g., from gscreen) to see what mutations would activate latent binding sites.

Value

A data frame (long format) with one row per edit, containing columns:

seq_idx Index into input sequences/intervals (1-based)

strand +1 (forward) or -1 (reverse strand)

window_start 1-based position of optimal window within sequence

score_before PWM score before edits

score_after PWM score after all edits

n_edits Total number of edits needed

edit_num Which edit this row represents (1, 2, ...)

motif_col 1-based position within the motif where the edit occurs

ref Current base at this position

alt Suggested replacement base

gain Score improvement from this individual edit

window_seq Motif-length sequence at the optimal window (as seen by PSSM, reverse-complemented if on reverse strand)

mutated_seq Same sequence with all edits applied

When intervals are provided, additional columns chrom, start, end are included.

Sequences already above the threshold produce a single row with n_edits = 0. Unreachable sequences are omitted from the output.

See Also

gseq.pwm, gvtrack.create

Examples

```
gdb.init_examples()

# Simple PSSM
pssm <- matrix(c(1, 0, 0, 0, 0, 1, 0, 0),
    nrow = 2,
    dimnames = list(NULL, c("A", "C", "G", "T"))
)

# What edits are needed?
gseq.pwm_edits("CCGTACGT", pssm, score.thresh = -0.5, prior = 0)
```

---

gseq.read_homer *Read motifs from a HOMER motif format file*

Description

Parses a HOMER `.motif` format file and returns a named list of position probability matrices (PPM). Each matrix has rows corresponding to motif positions and columns `A`, `C`, `G`, `T`. The returned matrices are directly usable with gseq.pwm.

Usage

```
gseq.read_homer(file)
```

Arguments

`file`	character(1) path to a HOMER motif file (`.motif`).

Value

A named list of numeric matrices. Each matrix has columns `A`, `C`, `G`, `T` and one row per motif position. List names are derived from the consensus sequence. Each matrix carries the following attributes:

name Motif name / description from the header

consensus Consensus sequence from the header

log_odds_threshold Detection threshold (numeric)

log_p_value Log p-value (numeric)

w Motif width (integer)

source `"homer"`

See Also

Other motif functions: gseq.read_jaspar(), gseq.read_meme()

Examples

```
## Not run:
motifs <- gseq.read_homer("known_motifs.motif")
names(motifs)
m <- motifs[[1]]
head(m)
attr(m, "consensus")

## End(Not run)
```

---

gseq.read_jaspar *Read motifs from a JASPAR PFM format file*

Description

Parses a JASPAR Position Frequency Matrix (PFM) file and returns a named list of position probability matrices (PPM). Supports both the standard JASPAR header format (>ID NAME followed by labeled rows) and the simple 4-row PFM format. Counts are converted to probabilities by dividing each column by its column sum.

Usage

```
gseq.read_jaspar(file)
```

Arguments

file	character(1) path to a JASPAR format file (.jaspar, .pfm, .txt).

Value

A named list of numeric matrices. Each matrix has columns A, C, G, T and one row per motif position. List names are motif identifiers. Each matrix carries the following attributes:

name Motif name from the header line

w Motif width (integer)

nsites Total counts per position (numeric; NA for simple-format files)

format Sub-format detected: "jaspar" or "simple"

See Also

Other motif functions: gseq.read_homer(), gseq.read_meme()

Examples

```
## Not run:
motifs <- gseq.read_jaspar("JASPAR2024_CORE.jaspar")
names(motifs)
m <- motifs[[1]]
head(m)

## End(Not run)
```

`gseq.read_meme` *Read motifs from a MEME minimal motif format file*

Description

Parses a MEME minimal motif format file and returns a named list of position probability matrices (PPM). Each matrix has rows corresponding to motif positions and columns `A`, `C`, `G`, `T`. The returned matrices are directly usable with `gseq.pwm`.

Usage

```
gseq.read_meme(file)
```

Arguments

`file`	character(1) path to a MEME format file (`.meme`, `.txt`).

Value

A named list of numeric matrices. Each matrix has columns `A`, `C`, `G`, `T` and one row per motif position. List names are motif identifiers. Each matrix carries the following attributes:

name Motif name / alternate ID (second token on the MOTIF line)

alength Alphabet length (integer, typically 4)

w Motif width (integer, number of positions)

nsites Number of sites used to build the matrix (numeric; `NA` if absent)

E E-value (numeric; `NA` if absent)

url URL string if present, otherwise `NA`

strand Strand specification from the file header (e.g. `"+ -"`)

background Named numeric vector of background frequencies (`c(A=..., C=..., G=..., T=...)`), or `NULL` if absent

See Also

Other motif functions: `gseq.read_homer()`, `gseq.read_jaspar()`

Examples

```
## Not run:
motifs <- gseq.read_meme("JASPAR2024_CORE_vertebrates.meme")
names(motifs)
m <- motifs[[1]]
head(m)
attr(m, "name")
attr(m, "nsites")

## End(Not run)
```

---

gseq.rev *Reverse DNA sequence*

Description

Takes a DNA sequence string and returns its reverse (without complementing).

Usage

```
gseq.rev(seq)
```

Arguments

seq	A character vector containing DNA sequences. Preserves case and handles NA values.

Value

A character vector of the same length as the input, containing the reversed sequences

See Also

gseq.revcomp, gseq.comp

Examples

```
gseq.rev("ACTG") # Returns "GTCA"
gseq.rev(c("ACTG", "GGCC")) # Returns c("GTCA", "CCGG")
gseq.rev(c("ACTG", NA, "GGCC")) # Returns c("GTCA", NA, "CCGG")
```

gseq.revcomp *Get reverse complement of DNA sequence*

Description

Alias for grevcomp. Takes a DNA sequence string and returns its reverse complement.

Usage

```
gseq.revcomp(seq)
```

Arguments

seq	A character vector containing DNA sequences (using A,C,G,T). Ignores other characters and NA values.

Value

A character vector of the same length as the input, containing the reverse complement sequences

See Also

grevcomp, gseq.rev, gseq.comp

Examples

```
gseq.revcomp("ACTG") # Returns "CAGT"
gseq.revcomp(c("ACTG", "GGCC")) # Returns c("CAGT", "GGCC")
```

gsummary *Calculates summary statistics of track expression*

Description

Calculates summary statistics of track expression.

Usage

```
gsummary(expr = NULL, intervals = NULL, iterator = NULL, band = NULL)
```

Arguments

expr	track expression
intervals	genomic scope for which the function is applied
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expression.
band	track expression band. If 'NULL' no band is used.

Details

This function returns summary statistics of a track expression: total number of bins, total number of bins whose value is NaN, min, max, sum, mean and standard deviation of the values.

Value

An array that represents summary statistics.

NaN values

A track expression evaluates to NaN wherever the iterator produces a bin the track has no data for. What happens next depends on the function:

- gextract **keeps** NaN rows, so the result has one row per iterator interval whether or not the track covered it.
- gsummary **counts** them and reports the count as the "NaN intervals" element, while the statistics themselves are computed over the non-NaN values only.
- gdist, gquantiles and gscreen **drop** them: NaN bins are not counted into any distribution bin, do not contribute to a percentile, and never satisfy a screening condition - including a condition that would be true of every real value.
- gsegment **spans** them: a NaN bin contributes no evidence to the test that places a boundary, but it still falls inside whichever segment surrounds it, so the returned segments tile the scope continuously rather than skipping the gaps.

So on 20 bins of which 7 are NaN, gextract returns 20 rows, gsummary reports 20 total and 7 NaN, and gdist counts 13; and on a 300 kb scope where 120 of 300 bins are NaN, gsegment still returns segments covering the full 300 kb.

The practical consequence is that NaN and zero are different, and collapsing them with `ifelse(is.na(x), 0, x)` turns "no data here" into a measured value of zero. Where that is genuinely what you want, note that it also changes every mean, quantile and distribution computed downstream.

See Also

gintervals.summary, gbins.summary

Examples

```
gdb.init_examples()
gsummary("rects_track")
```

gsynth.bin_map *Create a bin mapping from value-based merge specifications*

Description

Converts value-based bin merge specifications into a bin_map named vector that can be used with gsynth.train. This allows you to specify merges using actual track values rather than bin indices.

Usage

```
gsynth.bin_map(breaks, merge_ranges = NULL)
```

Arguments

breaks	Numeric vector of bin boundaries (same as used in gsynth.train)
merge_ranges	List of merge specifications. Each specification is a named list with:

from Numeric vector of length 2 c(min, max) defining the source value range to merge. Use -Inf or Inf for open-ended ranges. Can also be a single number (shorthand for c(value, Inf)).

to Numeric vector of length 2 c(min, max) defining the target bin that source bins should map to. Must match an existing bin defined by breaks.

Value

A named vector (bin_map) compatible with bin_map parameter in gsynth.train. The names are source bin indices (1-based), and values are target bin indices (1-based).

See Also

gsynth.train, gsynth.cell_merge, gsynth.sample

Examples

```
# Define breaks for GC content [0, 1] in 0.025 increments
breaks <- seq(0, 1, 0.025)

# Merge all GC content above 70% (0.7) into the bin (0.675, 0.7]
bin_map <- gsynth.bin_map(
    breaks = breaks,
    merge_ranges = list(
        list(from = 0.7, to = c(0.675, 0.7))
    )
)

# Multiple merges: merge low GC (< 0.3) and high GC (> 0.7) into middle bins
bin_map2 <- gsynth.bin_map(
    breaks = breaks,
    merge_ranges = list(
```

```
        list(from = c(-Inf, 0.3), to = c(0.4, 0.425)), # low GC -> (0.4, 0.425]
        list(from = 0.7, to = c(0.675, 0.7)) # high GC -> (0.675, 0.7]
    )
)
```

`gsynth.cell_merge` *Resolve a cell-level merge specification into flat bin indices*

Description

Unlike `gsynth.bin_map`, which merges bins independently along each dimension, `gsynth.cell_merge` resolves per-*joint-cell* redirects: each entry redirects one specific training cell (identified by its per-dimension values) to another specific training cell. This lets callers redirect arbitrary cells whose Cartesian position cannot be expressed via per-axis merges (e.g., "cell (GC=0.725, CG=0.05) –> cell (GC=0.70, CG=0.08)").

Usage

```
gsynth.cell_merge(model, cell_merge, bin_merge = NULL)
```

Arguments

`model`	A `gsynth.model` object from `gsynth.train`.
`cell_merge`	A list of redirect specifications. Each entry is a named list with: **from** Numeric vector of length `n_dims`; one representative value per dimension that identifies the *source* cell. **to** Numeric vector of length `n_dims`; one representative value per dimension that identifies the *target* cell. A single data frame with columns `from_1`, `from_2`, ..., `to_1`, `to_2`,... is also accepted and converted internally.
`bin_merge`	Optional sampling-time bin merge specification (same format as in `gsynth.sample`). When supplied, source and target per-dimension bin indices are remapped through the resulting per-axis maps before being combined into flat indices, so that cell values reference post-bin_merge cells.

Details

This function is primarily a utility for inspecting / debugging what `gsynth.sample` will do when invoked with the `cell_merge` argument. It returns a data frame describing, for every entry, the resolved source and target cells in both per-dimension-bin and flat-bin space.

Value

A data frame with one row per `cell_merge` entry and columns:

- `from_<d>`, `to_<d>` for each dimension `d`: 1-based bin index after any bin_merge remapping.
- `source_flat`, `target_flat`: 1-based flat bin index into `model$model_data$cdf`.

See Also

gsynth.bin_map, gsynth.sample

Examples

```
## Not run:
# Resolve a redirect table before handing it to gsynth.sample:
redirects <- list(
    list(from = c(0.725, 0.05), to = c(0.70, 0.08)),
    list(from = c(0.75, 0.06), to = c(0.70, 0.08))
)
resolved <- gsynth.cell_merge(model, redirects)
print(resolved)

gsynth.sample(model, "out.fa",
    output_format = "fasta",
    cell_merge = redirects
)

## End(Not run)
```

---

gsynth.convert *Convert a legacy RDS gsynth model to .gsm format*

Description

Reads a gsynth.model from a legacy RDS file and saves it in the cross-platform .gsm format.

Usage

```
gsynth.convert(input_file, output_file, compress = FALSE)
```

Arguments

input_file	Path to the legacy RDS model file
output_file	Path for the output .gsm model (directory or zip)
compress	Logical. If TRUE, save as a ZIP archive. If FALSE (default), save as a directory.

Value

Invisibly returns the output file path.

See Also

gsynth.save, gsynth.load

`gsynth.forbid_kmer` *Forbid a k-mer pattern in a trained gsynth model*

Description

Returns a new `gsynth.model` whose samples are guaranteed not to contain `pattern` as a substring (subject to the seeding caveat below). Analytically equivalent to rejection sampling the output, implemented by zeroing every transition that would produce the pattern and renormalizing per state-row.

Usage

```
gsynth.forbid_kmer(model, pattern, check = TRUE)
```

Arguments

`model`	A `gsynth.model` from `gsynth.train`.
`pattern`	Character scalar, uppercase DNA (`ACGT` only), with `nchar(pattern)` <= `model$k + 1`. Patterns longer than one transition cannot be forbidden locally and error.
`check`	Logical. If `TRUE` (default), print a short summary of how many transitions and how many bins were affected.

Details

Useful for building CpG-null, motif-null, or repeat-class-null synthetic backgrounds from a standard `gsynth.train()` model without retraining.

Seeding caveat. `gsynth.sample` initializes the first `k` bases of each sampling interval by uniform random draw, so those seed bases may themselves contain `pattern`. If the seed lands on a state k-mer that already contains `pattern` as a substring, every possible next base would extend that occurrence and thus be forbidden; such "trapped" states fall back to uniform sampling (not the forbid'd CDF) until the pattern slides out of the state window. The guarantee applies to the Markov-sampled bases downstream of the trap-escape window, not to the first few bases of the interval. Expected residual per interval is small but nonzero; for strict pattern-free output, pass `mask_copy` to `gsynth.sample` to seed from a known pattern-free reference, or scrub residuals after sampling.

Value

A new `gsynth.model` with modified `model_data$counts` and `model_data$cdf`. The original model is not mutated.

See Also

`gsynth.sample`, `gsynth.train`

Examples

```
## Not run:
# CpG-null synthetic background: train on the genome, then forbid CG.
model <- gsynth.train(
    list(expr = "gc_vt", breaks = seq(0, 1, 0.05)),
    intervals = gintervals.all(),
    iterator = 200
)
model_no_cg <- gsynth.forbid_kmer(model, "CG")
seqs <- gsynth.sample(model_no_cg,
    output_format = "vector",
    intervals = some_regions, seed = 42
)

# Motif-null background: forbid a 4-mer TF consensus substring.
model_no_ebox <- gsynth.forbid_kmer(model, "CACG")

## End(Not run)
```

---

gsynth.load *Load a gsynth.model from disk*

Description

Loads a previously saved Markov model. Auto-detects the format:

- If `file` is a directory, reads the .gsm directory format
- If `file` is a file, tries ZIP .gsm format first, then falls back to legacy RDS format

Usage

```
gsynth.load(file)
```

Arguments

`file`	Path to the saved model (directory, .gsm zip, or legacy .rds)

Value

A gsynth.model object

See Also

`gsynth.save`, `gsynth.train`, `gsynth.convert`

`gsynth.random` *Generate random genome sequences*

Description

Generates random DNA sequences based on nucleotide probabilities without using a trained Markov model. Each nucleotide is sampled independently according to the specified probabilities.

Usage

```
gsynth.random(
  intervals = NULL,
  output_path = NULL,
  output_format = c("misha", "fasta", "vector"),
  nuc_probs = c(A = 0.25, C = 0.25, G = 0.25, T = 0.25),
  mask_copy = NULL,
  preserve_n = TRUE,
  seed = NULL,
  n_samples = 1,
  iterator = 1
)
```

Arguments

`intervals`	Genomic intervals to sample. If NULL, uses all chromosomes.
`output_path`	Path to the output file (ignored when output_format = "vector")
`output_format`	Output format: • "misha": .seq binary format (default) • "fasta": FASTA text format • "vector": Return sequences as a character vector (does not write to file)
`nuc_probs`	Nucleotide probabilities. Can be specified as: • A named vector: `c(A = 0.3, C = 0.2, G = 0.2, T = 0.3)` • An unnamed vector in A, C, G, T order: `c(0.3, 0.2, 0.2, 0.3)` Probabilities are automatically normalized to sum to 1. Default is uniform (0.25 each).
`mask_copy`	Optional intervals to copy from the original genome instead of random sampling. Use this to preserve specific regions exactly as they appear in the reference.
`preserve_n`	Logical; default `TRUE`. When `TRUE`, positions whose original reference is `N` (or `n`) are written to the output verbatim rather than filled with a random ACGT base. Same semantics as in `gsynth.sample`; `mask_copy` intervals take precedence.
`seed`	Random seed for reproducibility. If NULL, uses current random state.
`n_samples`	Number of samples to generate per interval. Default is 1.
`iterator`	Iterator for position resolution. Default is 1 (base-pair resolution). Larger values may speed up processing but are typically not needed for random sampling.

Details

Unlike gsynth.sample which uses a trained Markov model to generate sequences that preserve k-mer statistics, gsynth.random generates purely random sequences where each nucleotide is sampled independently. This is useful for generating baseline random sequences or sequences with specific GC content.

Nucleotide ordering: When using an unnamed vector for nuc_probs, the order is A, C, G, T. Named vectors can be in any order.

Value

When output_format is "misha" or "fasta", returns invisible NULL and writes the random sequences to output_path. When output_format is "vector", returns a character vector of sequences (length = n_intervals * n_samples).

See Also

gsynth.sample, gsynth.train

Examples

```
gdb.init_examples()

# Generate random sequences with uniform nucleotide probabilities
seqs <- gsynth.random(
    intervals = gintervals(1, 0, 1000),
    output_format = "vector",
    seed = 42
)

# Generate GC-rich sequences (60% GC)
gc_rich <- gsynth.random(
    intervals = gintervals(1, 0, 1000),
    output_format = "vector",
    nuc_probs = c(A = 0.2, C = 0.3, G = 0.3, T = 0.2),
    seed = 42
)

# Generate AT-rich sequences
at_rich <- gsynth.random(
    intervals = gintervals(1, 0, 1000),
    output_format = "vector",
    nuc_probs = c(A = 0.35, C = 0.15, G = 0.15, T = 0.35),
    seed = 42
)
```

`gsynth.replace_kmer` *Iteratively replace a k-mer in the genome*

Description

Performs an iterative replacement of a `target` k-mer with a `replacement` sequence. This is useful for creating synthetic genomes with specific motifs removed (e.g., creating a CpG-null genome by iteratively swapping CG to GC).

Usage

```
gsynth.replace_kmer(
  target,
  replacement,
  output_path = NULL,
  output_format = c("misha", "fasta", "vector"),
  intervals = NULL,
  check_composition = TRUE
)
```

Arguments

`target`	The k-mer sequence to remove (e.g., "CG").
`replacement`	The replacement sequence (e.g., "GC").
`output_path`	Path to the output file (ignored when output_format = "vector").
`output_format`	Output format: • "misha": .seq binary format (default) • "fasta": FASTA text format • "vector": Return sequences as a character vector (does not write to file)
`intervals`	Genomic intervals to process. If NULL, uses all chromosomes.
`check_composition`	Logical. If TRUE (default), ensures target and replacement have the same nucleotide composition (preserving exact base counts).

Details

Bubble Sort / Iterative Logic: The function scans the sequence and replaces occurrences of `target` with `replacement`. If a replacement creates a new instance of `target` (e.g., removing "CG" with "GC" in the sequence "CCG" -> "CGC"), the new instance is also replaced. This continues until the sequence is free of the `target` k-mer.

When `target` and `replacement` are permutations of each other (e.g., "CG" and "GC"), this acts as a "bubble sort" of nucleotides, moving bases locally without altering the total GC content or base counts of the genome.

Value

When output_format is "misha" or "fasta", returns invisible NULL and writes to output_path. When output_format is "vector", returns a character vector of modified sequences.

Examples

```
## Not run:
# Robust removal of all CpG dinucleotides (preserving GC%)
gsynth.replace_kmer(
    target = "CG",
    replacement = "GC",
    output_path = "genome_no_cpg.seq",
    output_format = "misha"
)

## End(Not run)
```

`gsynth.sample` *Sample a synthetic genome from a trained Markov model*

Description

Generates a synthetic genome by sampling from a trained stratified Markov model. The generated genome preserves the k-mer statistics of the original genome within each stratification bin.

Usage

```
gsynth.sample(
  model,
  output_path = NULL,
  output_format = c("misha", "fasta", "vector"),
  mask_copy = NULL,
  preserve_n = TRUE,
  seed = NULL,
  intervals = NULL,
  n_samples = 1,
  bin_merge = NULL,
  cell_merge = NULL
)
```

Arguments

`model`	A gsynth.model object from `gsynth.train`
`output_path`	Path to the output file (ignored when output_format = "vector")
`output_format`	Output format: • "misha": .seq binary format (default) • "fasta": FASTA text format

- "vector": Return sequences as a character vector (does not write to file)

mask_copy — Optional intervals to copy from the original genome instead of sampling. Use this to preserve specific regions (e.g., repeats, regulatory elements) exactly as they appear in the reference. Regions not in mask_copy will be sampled using the Markov model. Note: mask_copy intervals should be non-overlapping and sorted by start position within each chromosome. Overlapping intervals may result in only the first overlapping region being copied, with subsequent overlaps skipped due to cursor advancement during sequential processing.

preserve_n — Logical; default TRUE. When TRUE, positions whose original reference is N (or n) are written to the output verbatim instead of being filled in with a sampled ACGT base. Case is preserved (so soft-masked n round-trips). mask_copy regions take precedence: inside a mask_copy interval the original byte is copied regardless. Set to FALSE to restore the pre-5.6.22 behavior of treating every position as something to sample.

seed — Random seed for reproducibility. If NULL, uses current random state.

intervals — Genomic intervals to sample. If NULL, uses all chromosomes.

n_samples — Number of samples to generate per interval. Default is 1. When n_samples > 1 and output_format = "fasta", headers include "_sampleN". When output_format = "vector", returns n_samples * n_intervals sequences.

bin_merge — Optional list of bin merge specifications to apply during sampling, one per dimension (length must equal model$n_dims). Each element should be:

- A list of merge specifications (same format as in gsynth.train: each spec is list(from = ..., to = ...))
- Or NULL to use the bin mapping from training for that dimension

This allows merging sparse bins at sampling time without re-training. Example for a 2D model:

```
bin_merge = list(
  # Dimension 1: merge bins with values >= 0.8 to bin [0.7, 0.8)
  list(list(from = c(0.8, Inf), to = c(0.7, 0.8))),
  # Dimension 2: use training-time bin_map (no override)
  NULL
)
```

cell_merge — Optional per-*joint-cell* redirect table, applied after bin_merge and after any sparse-bin uniform fallback. Each entry redirects one specific training cell to another specific training cell whose Cartesian position cannot be expressed as a per-axis merge. Format: a list where each element is list(from = c(v1, v2, ...), to = c(v1, v2, ...)), with one representative value per dimension; or a data frame with columns from_1, from_2, ..., to_1, to_2, ... Internally resolved via gsynth.cell_merge; at sampling time each source cell's CDF is replaced with the target cell's CDF (a pointer-level copy inside cdf_list — no matrix duplication and no change to the C++ hot path).

Details

FASTA index (.fai): When `output_format = "fasta"`, the function also writes a samtools-compatible `.fai` file alongside the FASTA (byte offsets are tracked during the write loop, so this is effectively free). The index is suitable for any downstream tool that expects a samtools-indexed reference.

N bases during sampling: By default (`preserve_n = TRUE`) positions whose original reference is N are copied verbatim into the output, so gaps and centromeres remain N rather than being fabricated as ACGT. Where the sampler still has to fill an N-adjacent k-mer context (the first k bp inside an interval, or any k-mer window containing a preserved N), it falls back to uniform random base selection until a valid context is re-established. Similarly, if a bin has no learned statistics (sparse bin with NA CDF), uniform sampling is used for that position. Pass `preserve_n = FALSE` to recover the pre-5.6.22 behavior of sampling every position regardless of the reference.

Sparse bins: If the model has sparse bins (from `min_obs` during training), a warning is issued when sampling regions that fall into these bins. Consider using `bin_merge` to redirect sparse bins to well-populated ones.

Value

When output_format is "misha" or "fasta", returns invisible NULL and writes the synthetic genome to output_path. When output_format is "vector", returns a character vector of sequences (length = n_intervals * n_samples).

See Also

gsynth.train, gsynth.save

Examples

```
gdb.init_examples()

# Create virtual tracks
gvtrack.create("g_frac", NULL, "kmer.frac", kmer = "G")
gvtrack.create("c_frac", NULL, "kmer.frac", kmer = "C")
gvtrack.create("cg_frac", NULL, "kmer.frac", kmer = "CG")
gvtrack.create("masked_frac", NULL, "masked.frac")

# Define repeat mask (regions to preserve from original)
repeats <- gscreen("masked_frac > 0.5",
    intervals = gintervals.all(),
    iterator = 100
)

# Train model (excluding repeats from training)
model <- gsynth.train(
    list(expr = "g_frac + c_frac", breaks = seq(0, 1, 0.025)),
    list(expr = "cg_frac", breaks = c(0, 0.01, 0.02, 0.03, 0.04, 0.2)),
    mask = repeats,
    iterator = 200,
    min_obs = 1000
)
```

```
# Sample with mask_copy to preserve repeats from original genome
temp_dir <- tempdir()
synthetic_genome_file <- file.path(temp_dir, "synthetic_genome.fa")
gsynth.sample(model, synthetic_genome_file,
    output_format = "fasta",
    mask_copy = repeats,
    seed = 60427,
    bin_merge = list(
        list(list(from = 0.7, to = c(0.675, 0.7))),
        list(list(from = 0.04, to = c(0.03, 0.04)))
    )
)
```

`gsynth.save` *Save a gsynth.model to disk in .gsm format*

Description

Saves a trained Markov model in the cross-platform .gsm format, which consists of a metadata YAML file and raw binary arrays for counts and CDFs. The .gsm format can be stored as a directory (default) or a ZIP archive.

Usage

```
gsynth.save(model, file, compress = FALSE)
```

Arguments

`model`	A gsynth.model object from `gsynth.train`
`file`	Path to save the model (directory or .zip file)
`compress`	Logical. If `TRUE`, save as a ZIP archive. If `FALSE` (default), save as a directory.

Value

Invisibly returns the file path.

See Also

`gsynth.load`, `gsynth.train`, `gsynth.convert`

`gsynth.score` *Score the genome under a trained gsynth model*

Description

Writes a misha fixed-bin dense track whose value at each output bin is the summed natural-log conditional probability of the reference sequence under the trained Markov model:

$$T(a) = \sum_{i=a}^{a+r-1} \log P_M(\text{seq}[i] \mid \text{seq}[i-k..i-1], b(i))$$

where $b(i)$ is the model's stratum bin (constant within each `model$iterator`-bp window). The first k bp of every chromosome are NA (no upstream context available); under default policies, any output bin containing an NA per-bp contribution is NA.

Usage

```
gsynth.score(
  model,
  track,
  description = NULL,
  intervals = NULL,
  mask = NULL,
  resolution = NULL,
  sparse_policy = c("NA", "uniform"),
  n_policy = c("NA", "uniform"),
  overwrite = FALSE
)
```

Arguments

`model`	A `gsynth.model` object (from `gsynth.train`).
`track`	Name of the misha track to create.
`description`	Optional track description.
`intervals`	Intervals to score. Defaults to `gintervals.all()`. Best results when interval starts are aligned to multiples of `model$iterator`; otherwise the first stratum window is shorter than `model$iterator` and its bin label may differ from training.
`mask`	Optional intervals to NA-out in the output (e.g. repeats). Intersects with `intervals` per bp; every output bin containing a masked bp becomes `NaN`.
`resolution`	Output bin size in bp. Defaults to `model$iterator`. `1` produces a per-bp track; any positive integer is allowed.
`sparse_policy`	How to score positions whose stratum bin is sparse in the model: `"NA"` (default) propagates NA; `"uniform"` contributes `log(1/4)` per bp.

n_policy	How to score positions whose k-mer *context* contains an N: "NA" (default) or "uniform" (log(1/4) per bp). The predicted base itself is always NA when N — the model has no $\log P$ for non-ACGT bases.
overwrite	If TRUE, replace an existing track of the same name.

Details

Two models scored against the same reference give a windowed log Bayes factor as a track expression:

```
gsynth.score(genome_model, "genome_score")
gsynth.score(cre_model,    "cre_score")
gextract("cre_score - genome_score", iterator=1000, ...)
```

Value

Invisibly NULL. Side effect: creates the named misha track. Output bins are written as NaN where the sum is undefined (out of intervals, or any per-bp NA).

See Also

gsynth.train, gsynth.sample

---

gsynth.train *Train a stratified Markov model from genome sequences*

Description

Computes a Markov model of order k (default 5) optionally stratified by bins of one or more track expressions (e.g., GC content and CG dinucleotide frequency). This model can be used to generate synthetic genomes that preserve the k-mer statistics of the original genome within each stratification bin. When called with no dimension specifications, trains a single unstratified model.

Usage

```
gsynth.train(
  ...,
  mask = NULL,
  intervals = NULL,
  iterator = NULL,
  pseudocount = 1,
  min_obs = 0,
  k = 5L,
  prior = "marginal"
)
```

Arguments

`...`	Zero or more dimension specifications. Each specification is a list containing: **expr** Track expression for this dimension (required) **breaks** Numeric vector of bin boundaries for this dimension (required) **bin_merge** Optional list of merge specifications for merging sparse bins. Each specification is a named list with 'from' and 'to' elements. If no dimensions are provided, trains an unstratified model with a single bin.
`mask`	Optional intervals to exclude from training. Regions in the mask will not contribute to k-mer counts. Can be computed using `gscreen()`.
`intervals`	Genomic intervals to process. If NULL, uses all chromosomes.
`iterator`	Iterator for track evaluation, determines the resolution at which track values are computed.
`pseudocount`	Total Dirichlet concentration alpha used in the smoothed posterior `P(a|c,b) = (N + alpha * pi_a(b)) / (sum_a N + alpha)`. Default is 1.
`min_obs`	Minimum number of observations ((k+1)-mers) required per bin. Bins with fewer observations will be marked as NA (not learned) and a warning will be issued. Default is 0 (no minimum). During sampling, NA bins will fall back to uniform sampling unless merged via `bin_merge`.
`k`	Integer Markov order (1–10). Default is 5, which models 6-mer (context of length 5 plus the emitted base) transition probabilities. Higher values capture longer-range sequence dependencies but require exponentially more memory (4^k context states).
`prior`	Per-base Dirichlet prior $\pi_a(b)$. One of: • `"marginal"` (default): per-bin empirical base composition learned from the trainer's own counts (post bin-merge). Bins with zero observations fall back to uniform with a warning. • `"global"`: a single base composition pooled over all bins, broadcast to every bin. • `NULL` or `"uniform"`: uniform prior (1/4 per base) – the pre-5.6.21 fallback. • Length-4 numeric (optionally named A, C, G, T): user-supplied global π, broadcast. • `n_bins x 4` numeric matrix: user-supplied per-bin π. Together with `pseudocount`, this defines the Dirichlet posterior. To reproduce the pre-5.6.21 Laplace-add-one behavior, pass `prior = NULL, pseudocount = 4`.

Details

Strand symmetry: The training process counts both the forward strand (k+1)-mer and its reverse complement for each position, ensuring strand-symmetric transition probabilities. This means the reported total_kmers is approximately double the number of genomic positions processed.

N bases: Positions where the (k+1)-mer contains any N (unknown) bases are skipped during training and counted in `total_n`. The model only learns from valid A/C/G/T sequences.

Value

A `gsynth.model` object containing:

k Markov order used for training

num_kmers Number of context states (4^k)

n_dims Number of stratification dimensions

dim_specs List of dimension specifications (expr, breaks, num_bins, bin_map)

dim_sizes Vector of bin counts per dimension

total_bins Total number of bins (product of dim_sizes)

total_kmers Total number of valid (k+1)-mers counted

per_bin_kmers Number of (k+1)-mers counted per bin

total_masked Number of positions skipped due to mask

total_n Number of positions skipped due to N bases

model_data Internal model data (counts and CDFs)

See Also

`gsynth.sample`, `gsynth.save`, `gsynth.load`, `gsynth.bin_map`

Examples

```
gdb.init_examples()

# Create virtual tracks for stratification
gvtrack.create("g_frac", NULL, "kmer.frac", kmer = "G")
gvtrack.create("c_frac", NULL, "kmer.frac", kmer = "C")
gvtrack.create("cg_frac", NULL, "kmer.frac", kmer = "CG")
gvtrack.create("masked_frac", NULL, "masked.frac")

# Define repeat mask
repeats <- gscreen("masked_frac > 0.5",
    intervals = gintervals.all(),
    iterator = 100
)

# Train unstratified model (no stratification)
model_0d <- gsynth.train(
    mask = repeats,
    intervals = gintervals.all(),
    iterator = 200
)

# Train model with 2D stratification (GC content and CG dinucleotide)
model <- gsynth.train(
    list(
        expr = "g_frac + c_frac",
        breaks = seq(0, 1, 0.025),
        bin_merge = list(list(from = 0.7, to = c(0.675, 0.7)))
```

```
    ),
    list(
        expr = "cg_frac",
        breaks = c(0, 0.01, 0.02, 0.03, 0.04, 0.2),
        bin_merge = list(list(from = 0.04, to = c(0.03, 0.04)))
    ),
    mask = repeats,
    intervals = gintervals.all(),
    iterator = 200
)
```

`gtrack.2d.convert_to_indexed` *Convert 2D track to indexed format*

Description

Converts a per-chromosome-pair 2D track (rectangles or points) to indexed format (track.dat + track.idx). This reduces file descriptor usage from O(N^2) to O(1), which is especially beneficial for genomes with many contigs.

Usage

```
gtrack.2d.convert_to_indexed(track = NULL, remove.old = FALSE, force = FALSE)
```

Arguments

`track`	track name to convert
`remove.old`	Logical. If TRUE, removes old per-chromosome-pair files after successful conversion. Default: FALSE.
`force`	Logical. If TRUE, re-converts even if already in indexed format. Default: FALSE.

Value

None.

See Also

`gtrack.2d.create`, `gtrack.2d.import`, `gtrack.2d.import_contacts`, `gtrack.convert_to_indexed`, `gdb.convert_to_indexed`

Examples

```
## Not run:
# Convert a 2D track to indexed format
gtrack.2d.convert_to_indexed("my_2d_track")

# Convert and remove old per-pair files
gtrack.2d.convert_to_indexed("my_2d_track", remove.old = TRUE)

# Force re-conversion
gtrack.2d.convert_to_indexed("my_2d_track", force = TRUE)

## End(Not run)
```

---

gtrack.2d.create *Creates a 'Rectangles' track from intervals and values*

Description

Creates a 'Rectangles' track from intervals and values.

Usage

```
gtrack.2d.create(
  track = NULL,
  description = NULL,
  intervals = NULL,
  values = NULL
)
```

Arguments

track	track name
description	a character string description
intervals	a set of two-dimensional intervals
values	an array of numeric values - one for each interval

Details

This function creates a new 'Rectangles' (two-dimensional) track with values at given intervals. 'description' is added as a track attribute.

When multiple databases are connected via gsetroot, the track is created in the current working directory (.misha$GWD), which defaults to the last connected database. Use gdir.cd with an absolute path to change where new tracks are created.

Value

None.

See Also

gtrack.create, gtrack.create_sparse, gtrack.smooth, gtrack.modify, gtrack.rm, gtrack.info, gdir.create, gtrack.attr.get

Examples

```
gdb.init_examples()
intervs1 <- gintervals.2d(
    1, (1:4) * 200, (1:4) * 200 + 100,
    1, (1:4) * 300, (1:4) * 300 + 200
)
intervs2 <- gintervals.2d(
    "X", (7:10) * 100, (7:10) * 100 + 50,
    2, (1:4) * 200, (1:4) * 200 + 130
)
intervs <- rbind(intervs1, intervs2)
gtrack.2d.create(
    "test_rects", "Test 2d track", intervs,
    runif(dim(intervs)[1], 1, 100)
)
gextract("test_rects", .misha$ALLGENOME)
gtrack.rm("test_rects", force = TRUE)
```

---

gtrack.2d.import *Creates a 2D track from tab-delimited file*

Description

Creates a 2D track from tab-delimited file(s).

Usage

```
gtrack.2d.import(track = NULL, description = NULL, file = NULL)
```

Arguments

track	track name
description	a character string description
file	vector of file paths

Details

This function creates a 2D track track from one or more tab-delimited files. Each file must start with a header describing the columns. The first 6 columns must have the following names: 'chrom1', 'start1', 'end1', 'chrom2', 'start2', 'end2'. The last column is designated for the value and it may

have an arbitrary name. The header is followed by a list of intervals and a value for each interval. Overlapping intervals are forbidden.

One can learn about the format of the tab-delimited file by running 'gextract' function on a 2D track with a 'file' parameter set to the name of the file.

If all the imported intervals represent a point (i.e. end == start + 1) a 'Points' track is created otherwise it is a 'Rectangles' track.

'description' is added as a track attribute.

Note: temporary files are created in the directory of the track during the run of the function. A few of them need to be kept simultaneously open. If the number of chromosomes and / or intervals is particularly high, a few thousands files might be needed to be opened simultaneously. Some operating systems limit the number of open files per user, in which case the function might fail with "Too many open files" or similar error. The workaround could be:

1. Increase the limit of simultaneously opened files (the way varies depending on your operating system). 2. Increase the value of 'gmax.data.size' option. Higher values of 'gmax.data.size' option will increased memory usage of the function but create fewer temporary files.

Value

None.

See Also

gtrack.rm, gtrack.info, gdir.create

`gtrack.2d.import_contacts` *Creates a track from a file of inter-genomic contacts*

Description

Creates a track from a file of inter-genomic contacts.

Usage

```
gtrack.2d.import_contacts(
  track = NULL,
  description = NULL,
  contacts = NULL,
  fends = NULL,
  allow.duplicates = TRUE
)
```

Arguments

track	track name
description	a character string description
contacts	vector of contacts files
fends	name of fragment ends file
allow.duplicates	if 'TRUE' duplicated contacts are allowed

Details

This function creates a 'Points' (two-dimensional) track from contacts files. If 'allow.duplicates' is 'TRUE' duplicated contacts are allowed and summed up, otherwise an error is reported.

Contacts (coord1, coord2) within the same chromosome are automatically doubled to include also '(coord2, coord1)' unless 'coord1' equals to 'coord2'.

Contacts may come in one or more files.

If 'fends' is 'NULL' contacts file is expected to be in "intervals-value" tab-separated format. The file starts with a header defining the column names. The first 6 columns must have the following names: 'chrom1', 'start1', 'end1', 'chrom2', 'start2', 'end2'. The last column is designated for the value and it may have an arbitrary name. The header is followed by a list of intervals and a value for each interval. An interval of form (chrom1, start1, end1, chrom2, start2, end2) is added as a point (X, Y) to the resulted track where X = (start1 + end1) / 2 and Y = (start2 + end2) / 2.

One can see an example of "intervals-value" format by running 'gextract' function on a 2D track with a 'file' parameter set to the name of the file.

If 'fends' is not 'NULL' contacts file is expected to be in "fends-value" tab-separated format. It should start with a header containing at least 3 column names 'fend1', 'fend2' and 'count' in arbitrary order followed by lines each defining a contact between two fragment ends.

COLUMN	VALUE	DESCRIPTION
fend1	Integer	ID of the first fragment end
fend2	Integer	ID of the second fragment end
count	Numeric	Value associated with the contact

A fragment ends file is also in tab-separated format. It should start with a header containing at least 3 column names 'fend', 'chr' and 'coord' in arbitrary order followed by lines each defining a single fragment end.

COLUMN	VALUE	DESCRIPTION
fend	Unique integer	ID of the fragment end
chr	Chromosome name	Can be specified with or without "chr" prefix, like: "X" or "chrX"
coord	Integer	Coordinate

'description' is added as a track attribute.

Note: temporary files are created in the directory of the track during the run of the function. A few of them need to be kept simultaneously open. If the number of chromosomes and / or contacts is particularly high, a few thousands files might be needed to be opened simultaneously. Some operating systems limit the number of open files per user, in which case the function might fail with "Too many open files" or similar error. The workaround could be:

1. Increase the limit of simultaneously opened files (the way varies depending on your operating system). 2. Increase the value of 'gmax.data.size' option. Higher values of 'gmax.data.size' option will increased memory usage of the function but create fewer temporary files.

Value

None.

See Also

gtrack.2d.import, gtrack.rm, gtrack.info, gdir.create

---

gtrack.array.extract *Returns values from 'Array' track*

Description

Returns values from 'Array' track.

Usage

```
gtrack.array.extract(
  track = NULL,
  slice = NULL,
  intervals = NULL,
  file = NULL,
  intervals.set.out = NULL
)
```

Arguments

track	track name
slice	a vector of column names or column indices or 'NULL'
intervals	genomic scope for which the function is applied
file	file name where the function result is to be saved. If 'NULL' result is returned to the user.
intervals.set.out	intervals set name where the function result is optionally outputted

Details

This function returns the column values of an 'Array' track in the genomic scope specified by 'intervals'. 'slice' parameter determines which columns should appear in the result. The columns can be indicated by their names or their indices. If 'slice' is 'NULL' the values of all track columns are returned.

The order inside the result might not be the same as the order of intervals. An additional column 'intervalID' is added to the return value. Use this column to refer to the index of the original interval from the supplied 'intervals'.

If 'file' parameter is not 'NULL' the result is saved to a tab-delimited text file (without 'intervalID' column) rather than returned to the user. This can be especially useful when the result is too big to fit into the physical memory. The resulted file can be used as an input for 'gtrack.array.import' function.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Similarly to 'file' parameter 'intervals.set.out' can be useful to overcome the limits of the physical memory.

Value

If 'file' and 'intervals.set.out' are 'NULL' a set of intervals with additional columns for 'Array' track column values and 'intervalID'.

See Also

gextract, gtrack.array.get_colnames, gtrack.array.import

Examples

```
gdb.init_examples()
gtrack.array.extract(
    "array_track", c("col3", "col5"),
    gintervals(1, 0, 2000)
)
```

---

gtrack.array.get_colnames

Returns column names of array track

Description

Returns column names of array track.

Usage

```
gtrack.array.get_colnames(track = NULL)
```

Arguments

track	track name

Details

This function returns the column names of an array track.

Value

A character vector with column names.

See Also

gtrack.array.set_colnames, gtrack.array.extract, gvtrack.array.slice, gtrack.info

Examples

```
gtrack.array.get_colnames("array_track")
```

---

gtrack.array.import *Creates an array track from array tracks or files*

Description

Creates an array track from array tracks or files.

Usage

```
gtrack.array.import(track = NULL, description = NULL, ...)
```

Arguments

track	name of the newly created track
description	a character string description
...	array track or name of a tab-delimited file

Details

This function creates a new 'Array' track from one or more "sources". Each source can be either another 'Array' track or a tab-delimited file that contains one-dimensional intervals and column values that should be added to the newly created track. One can learn about the exact format of the file by running 'gtrack.array.extract' or 'gextract' functions with a 'file' parameter and inspecting the output file.

There might be more than one source used to create the new track. In that case the new track will contain the columns from all the sources. The equally named columns are merged. Intervals that

appear in one source but not in the other are added and the values for the missing columns are set to NaN. Intervals with all NaN values are not added. Partial overlaps between two intervals from different sources are forbidden.

'description' is added as a track attribute.

Value

None.

See Also

gextract, gtrack.array.extract, gtrack.array.set_colnames, gtrack.rm, gtrack.info, gdir.create

Examples

```
f1 <- tempfile()
gextract("sparse_track", gintervals(1, 5000, 20000), file = f1)
f2 <- tempfile()
gtrack.array.extract("array_track", c("col2", "col3", "col4"),
    gintervals(1, 0, 20000),
    file = f2
)
f3 <- tempfile()
gtrack.array.extract("array_track", c("col1", "col3"),
    gintervals(1, 0, 20000),
    file = f3
)

gtrack.array.import("test_track1", "Test array track 1", f1, f2)
gtrack.array.extract("test_track1", NULL, .misha$ALLGENOME)

gtrack.array.import(
    "test_track2", "Test array track 2",
    "test_track1", f3
)
gtrack.array.extract("test_track2", NULL, .misha$ALLGENOME)

gtrack.rm("test_track1", TRUE)
gtrack.rm("test_track2", TRUE)
unlink(c(f1, f2, f3))
```

---

gtrack.array.set_colnames

Sets column names of array track

Description

Sets column names of array track.

Usage

```
gtrack.array.set_colnames(track = NULL, names = NULL)
```

Arguments

track	track name
names	vector of column names

Details

This sets the column names of an array track.

Value

None.

See Also

gtrack.array.get_colnames, gtrack.array.extract, gvtrack.array.slice, gtrack.info

Examples

```
old.names <- gtrack.array.get_colnames("array_track")
new.names <- paste("modified", old.names, sep = "_")
gtrack.array.set_colnames("array_track", new.names)
gtrack.array.get_colnames("array_track")
gtrack.array.set_colnames("array_track", old.names)
gtrack.array.get_colnames("array_track")
```

---

gtrack.attr.export *Returns track attributes values*

Description

Returns track attributes values.

Usage

```
gtrack.attr.export(tracks = NULL, attrs = NULL)
```

Arguments

tracks	a vector of track names or 'NULL'
attrs	a vector of attribute names or 'NULL'

Details

This function returns a data frame that contains track attributes values. Column names of the data frame consist of the attribute names, row names contain the track names.

The list of required tracks is specified by 'tracks' argument. If 'tracks' is 'NULL' the attribute values of all existing tracks are returned.

Likewise the list of required attributes is controlled by 'attrs' argument. If 'attrs' is 'NULL' all attribute values of the specified tracks are returned. The columns are also sorted then by "popularity" of an attribute, i.e. the number of tracks containing this attribute. This sorting is not applied if 'attrs' is not 'NULL'.

Empty character string in a table cell marks a non-existing attribute.

Value

A data frame containing track attributes values.

See Also

gtrack.attr.import, gtrack.attr.get, gtrack.attr.set

Examples

```
gdb.init_examples()
gtrack.attr.export()
gtrack.attr.export(tracks = c("sparse_track", "dense_track"))
gtrack.attr.export(attrs = "created.by")
```

---

gtrack.attr.get *Returns value of a track attribute*

Description

Returns value of a track attribute.

Usage

```
gtrack.attr.get(track = NULL, attr = NULL)
```

Arguments

track	track name
attr	attribute name

Details

This function returns the value of a track attribute. If the attribute does not exist an empty sting is returned.

Value

Track attribute value.

See Also

gtrack.attr.import, gtrack.attr.set

Examples

```
gdb.init_examples()
gtrack.attr.set("sparse_track", "test_attr", "value")
gtrack.attr.get("sparse_track", "test_attr")
gtrack.attr.set("sparse_track", "test_attr", "")
```

---

gtrack.attr.import *Imports track attributes values*

Description

Imports track attributes values.

Usage

```
gtrack.attr.import(table = NULL, remove.others = FALSE)
```

Arguments

table	a data frame containing attribute values
remove.others	specifies what to do with the attributes that are not in the table

Details

This function makes imports attribute values contained in a data frame 'table'. The format of a table is similar to the one returned by 'gtrack.attr.export'. The values of the table must be character strings. Column names of the table should specify the attribute names, while row names should contain the track names.

The specified attributes of the specified tracks are modified. If an attribute value is an empty string this attribute is removed from the track.

If 'remove.others' is 'TRUE' all non-readonly attributes that do not appear in the table are removed, otherwise they are preserved unchanged.

Error is reported on an attempt to modify a value of a read-only attribute.

Value

None.

See Also

gtrack.attr.import, gtrack.attr.set, gtrack.attr.get, gdb.get_readonly_attrs

Examples

```
gdb.init_examples()
t <- gtrack.attr.export()
t$newattr <- as.character(1:dim(t)[1])
gtrack.attr.import(t)
gtrack.attr.export(attrs = "newattr")

# roll-back the changes
t$newattr <- ""
gtrack.attr.import(t)
```

---

gtrack.attr.set *Assigns value to a track attribute*

Description

Assigns value to a track attribute.

Usage

```
gtrack.attr.set(track = NULL, attr = NULL, value = NULL)
```

Arguments

track	track name
attr	attribute name
value	value

Details

This function creates a track attribute and assigns 'value' to it. If the attribute already exists its value is overwritten.

If 'value' is an empty string the attribute is removed.

Error is reported on an attempt to modify a value of a read-only attribute.

Value

None.

See Also

gtrack.attr.get, gtrack.attr.import, gtrack.var.set, gdb.get_readonly_attrs

Examples

```
gdb.init_examples()
gtrack.attr.set("sparse_track", "test_attr", "value")
gtrack.attr.get("sparse_track", "test_attr")
gtrack.attr.set("sparse_track", "test_attr", "")
```

---

gtrack.convert *Converts a track to the most current format*

Description

Converts a track (if needed) to the most current format.

Usage

```
gtrack.convert(src.track = NULL, tgt.track = NULL)
```

Arguments

src.track	source track name
tgt.track	target track name. If 'NULL' the source track is overwritten.

Details

This function converts a track to the most current format. It should be used if a track created by an old version of the library cannot be read anymore by the newer version. The old track is given by 'src.track'. After conversion a new track 'tgt.track' is created. If 'tgt.track' is 'NULL' the source track is overwritten.

Value

None

See Also

gtrack.create, gtrack.2d.create, gtrack.create_sparse

---

gtrack.convert_to_indexed

Convert a track to indexed format

Description

Converts a per-chromosome track to indexed format (track.dat + track.idx).

Usage

```
gtrack.convert_to_indexed(track = NULL)
```

Arguments

track	track name to convert

Details

This function converts a track from the per-chromosome file format to single-file indexed format. The indexed format dramatically reduces file descriptor usage for genomes with many contigs and provides better performance for parallel access.

The function performs the following steps:

1. Validates that all per-chromosome files have consistent metadata
2. Creates track.dat by concatenating all per-chromosome files
3. Creates track.idx with offset/length information for each chromosome
4. Uses atomic operations (fsync + rename) to ensure data integrity
5. Removes the old per-chromosome files after successful conversion

Value

None

See Also

gtrack.create, gtrack.create_sparse, gtrack.create_dense

Examples

```
## Not run:
# Convert a track to indexed format
gtrack.convert_to_indexed("my_track")

## End(Not run)
```

---

gtrack.copy *Copies one or more tracks*

Description

Creates a copy of an existing track, optionally to a different database. Transparently handles format mismatches (per-chromosome vs indexed) and chromosome-order differences between source and destination databases.

Usage

```
gtrack.copy(src = NULL, dest = NULL, db = NULL, overwrite = FALSE)
```

Arguments

src	source track name(s). Either a single name or a character vector of names.
dest	destination name. If src is a single name, this is the destination track name (defaults to src). If src is a vector, dest is treated as a namespace prefix (e.g. "ns" produces "ns.track1", "ns.track2", ...). NULL keeps each track's name.
db	destination database root. Must be the current GROOT or a member of GDATASETS. NULL means the current working directory db.
overwrite	if TRUE, replace an existing destination track.

Details

Chromosomes that exist in the source database but not in the destination are dropped with a warning.

For 2D tracks (rectangles, points), cross-database copy requires identical chromosome order in source and destination. Format conversion (per-chromosome to indexed) is supported only when the destination is the active database.

Value

invisibly, the character vector of created track names.

See Also

gtrack.mv, gtrack.rm

Examples

```
gdb.init_examples()
gtrack.copy("dense_track", "dense_track_copy")
gtrack.exists("dense_track_copy")
gtrack.rm("dense_track_copy", force = TRUE)
```

---

gtrack.create *Creates a track from a track expression*

Description

Creates a track from a track expression.

Usage

```
gtrack.create(
  track = NULL,
  description = NULL,
  expr = NULL,
  iterator = NULL,
  band = NULL
)
```

Arguments

track	track name
description	a character string description
expr	track expression
iterator	track expression iterator. If 'NULL' iterator is determined implicitly based on track expression.
band	track expression band. If 'NULL' no band is used.

Details

This function creates a new track named track. The values of the track are determined by evaluation of 'expr' - a numeric track expression. The type of the new track is determined by the type of the iterator. 'Fixed bin', 'Sparse' or 'Rectangles' track can be created accordingly. 'description' is added as a track attribute.

When multiple databases are connected via gsetroot, the track is created in the current working directory (.misha$GWD), which defaults to the last connected database. Use gdir.cd with an absolute path to change where new tracks are created.

Value

None.

See Also

gtrack.2d.create, gtrack.create_sparse, gtrack.smooth, gtrack.modify, gtrack.rm, gtrack.info, gdir.create

Examples

```
gdb.init_examples()

## Creates a new track that is a sum of values from 'dense' and
## 2 * non-nan values of 'sparse' track. The new track type is
## Dense with a bin size that equals to '70'.
gtrack.create("mixed_track", "Test track",
    "dense_track +
              replace(sparse_track, is.nan(sparse_track), 0) * 2",
    iterator = 70
)
gtrack.info("mixed_track")
gtrack.rm("mixed_track", force = TRUE)
```

---

gtrack.create_dense *Creates a 'Dense' track from intervals and values*

Description

Creates a 'Dense' track from intervals and values.

Usage

```
gtrack.create_dense(
  track = NULL,
  description = NULL,
  intervals = NULL,
  values = NULL,
  binsize = NULL,
  defval = NaN,
  func = "weighted.mean"
)
```

Arguments

track	track name
description	a character string description
intervals	a set of one-dimensional intervals
values	an array of numeric values - one for each interval
binsize	bin size of the newly created 'Dense' track
defval	default track value for genomic regions not covered by the intervals
func	per-bin aggregation function over intervals overlapping each bin. One of: "weighted.mean" (default) sum(v_i * ov_i) / sum(ov_i) - byte-identical to the historical behavior. "weighted.sum" sum(v_i * ov_i) - coverage-weighted integral over the bin. "max" max(v_i) over intervals touching the bin (unweighted). "min" min(v_i) over intervals touching the bin (unweighted). "median" overlap-weighted (lower) median by coverage mass. "count" number of intervals touching the bin. Empty bin = 0. "coverage" sum(v_i * ov_i) / binsize - average per-base signal in the bin. With values = rep(1, n) this is a ChIP-seq-style pileup track (mean overlapping intervals per base). For weighted.mean, weighted.sum, max, min, median, and coverage, uncovered bases in a bin act as a synthetic contribution with value defval (overlap = uncovered_bases), included iff defval is not NaN. count ignores defval.

Details

This function creates a new 'Dense' track with values at given intervals. 'description' is added as a track attribute.

Value

None.

See Also

gtrack.create_sparse, gtrack.import, gtrack.modify, gtrack.rm, gtrack.info

Examples

```
gdb.init_examples()
intervs <- gintervals.load("annotations")
gtrack.create_dense(
    "test_dense", "Test dense track", intervs,
    1:dim(intervs)[1], 50, 0
)
gextract("test_dense", .misha$ALLGENOME)
gtrack.rm("test_dense", force = TRUE)
```

gtrack.create_dirs *Create directories needed for track creation*

Description

This function creates the directories needed for track creation. For example, if the track name is 'proj.sample.my_track', this function creates the directories 'proj' and 'sample'. Use this function with caution - a long track name may create a deep directory structure.

Usage

```
gtrack.create_dirs(track, mode = "0777")
```

Arguments

track	name of the track
mode	see 'dir.create'

Value

None.

Examples

```
gdb.init_examples()

# This creates the directories 'proj' and 'sample'
gtrack.create_dirs("proj.sample.my_track")
```

gtrack.create_pwm_energy *Creates a new track from PSSM energy function*

Description

Creates a new track from PSSM energy function.

Usage

```
gtrack.create_pwm_energy(
  track = NULL,
  description = NULL,
  pssmset = NULL,
  pssmid = NULL,
  prior = NULL,
  iterator = NULL
)
```

Arguments

track	track name
description	a character string description
pssmset	name of PSSM set: 'pssmset.key' and 'pssmset.data' must be presented in 'GROOT/pssms' directory
pssmid	PSSM id
prior	prior
iterator	track expression iterator for the newly created track

Details

This function creates a new track with values of a PSSM energy function. PSSM parameters (nucleotide probability per position and pluralization) are determined by 'pssmset' key and data files ('pssmset.key' and 'pssmset.data'). These two files must be located in 'GROOT/pssms' directory. The type of the created track is determined by the type of the iterator. 'description' is added as a track attribute.

Value

None.

See Also

gtrack.create, gtrack.2d.create, gtrack.create_sparse, gtrack.smooth, gtrack.modify, gtrack.rm, gtrack.info, gdir.create

Examples

```
gdb.init_examples()
gtrack.create_pwm_energy("pwm_energy_track", "Test track", "pssm",
    3, 0.01,
    iterator = 100
)
gextract("pwm_energy_track", gintervals(1, 0, 1000))
```

---

gtrack.create_sparse *Creates a 'Sparse' track from intervals and values*

Description

Creates a 'Sparse' track from intervals and values.

Usage

```
gtrack.create_sparse(
  track = NULL,
  description = NULL,
  intervals = NULL,
  values = NULL
)
```

Arguments

track	track name
description	a character string description
intervals	a set of one-dimensional intervals
values	an array of numeric values - one for each interval, in the same order as the rows of intervals. If NULL, the value column of intervals is used.

Details

This function creates a new 'Sparse' track with values at given intervals. 'description' is added as a track attribute.

When multiple databases are connected via gsetroot, the track is created in the current working directory (.misha$GWD), which defaults to the last connected database. Use gdir.cd with an absolute path to change where new tracks are created.

values is matched to intervals row by row, in the order the intervals are passed; intervals need not be sorted. Note however that gintervals returns its result sorted in the canonical chromosome order, so building intervals with gintervals() while keeping values in the original order will bind values to the wrong intervals. Keep the values in a value column of the intervals data frame and omit values to make such a mismatch impossible.

Value

None.

See Also

gtrack.create, gtrack.2d.create, gtrack.smooth, gtrack.modify, gtrack.rm, gtrack.info, gdir.create

Examples

```
gdb.init_examples()
intervs <- gintervals.load("annotations")
gtrack.create_sparse(
    "test_sparse", "Test track", intervs,
    1:dim(intervs)[1]
)
gextract("test_sparse", .misha$ALLGENOME)
```

```
gtrack.rm("test_sparse", force = TRUE)
```

`gtrack.dataset` *Returns the database/dataset path for a track*

Description

Returns the path of the database or dataset containing a track.

Usage

```
gtrack.dataset(track = NULL)
```

Arguments

`track`	track name or a vector of track names

Details

When datasets are loaded, tracks can come from either the working database or from loaded datasets. This function returns the source path for each track.

Value

Character vector of database/dataset paths. Returns NA for non-existent tracks.

See Also

`gtrack.dbs`, `gtrack.exists`, `gtrack.ls`, `gdataset.ls`

Examples

```
gdb.init_examples()
gtrack.dataset("dense_track")
```

gtrack.dbs *Returns the database paths that contain track(s)*

Description

Returns all database paths that contain a version of a track.

Usage

```
gtrack.dbs(track = NULL, dataframe = FALSE)
```

Arguments

track	track name or a vector of track names
dataframe	return a data frame with columns track and db instead of a named character vector.

Details

When datasets are loaded, a track may exist in multiple locations (working database and/or datasets). This function computes on-demand and returns all such paths, which is useful for debugging when using force=TRUE with gdataset.load().

Value

A named character vector of database paths for each track. If dataframe is TRUE, returns a data frame with columns track and db, with multiple rows per track when it appears in multiple databases.

See Also

gtrack.dataset, gtrack.exists, gtrack.ls, gdataset.ls

Examples

```
gdb.init_examples()
gtrack.dbs("dense_track")
gtrack.dbs(gtrack.ls(), dataframe = TRUE)
```

gtrack.exists *Tests for a track existence*

Description

Tests for a track existence.

Usage

```
gtrack.exists(track = NULL)
```

Arguments

track	track name

Details

This function returns 'TRUE' if a track exists in Genomic Database.

Value

'TRUE' if a track exists. Otherwise 'FALSE'.

See Also

gtrack.ls, gtrack.info, gtrack.create, gtrack.rm

Examples

```
gdb.init_examples()
gtrack.exists("dense_track")
```

gtrack.export_bedgraph *Export a track to bedGraph format*

Description

Exports a track or track expression to a UCSC bedGraph file.

Usage

```
gtrack.export_bedgraph(
  track,
  file,
  intervals = NULL,
  iterator = NULL,
  name = NULL
)
```

Arguments

track	track name or track expression (character string)
file	output file path. If it ends in .gz, output is gzip-compressed.
intervals	genomic intervals to export. If NULL (default), the entire genome (.misha$ALLGENOME) is used.
iterator	iterator bin size. If NULL (default), the iterator is determined automatically from the track expression.
name	track name for the bedGraph header line. If NULL (default), uses the track parameter value.

Details

This function evaluates a track expression over the specified genomic intervals and writes the result in standard bedGraph format (4-column, tab-separated: chrom, start, end, value). NaN values are omitted from the output.

The function supports physical tracks, virtual tracks, and arbitrary track expressions (e.g. "dense_track * 2"). 2D tracks are not supported.

If the output file path ends in .gz, the output is gzip-compressed.

Value

NULL (invisible). Called for its side effect of writing a file.

See Also

gextract, gtrack.export_bigwig, gtrack.info

Examples

```
## Not run:
gdb.init_examples()

# Export a dense track
gtrack.export_bedgraph("dense_track", "/tmp/dense.bedgraph")

# Export with specific intervals
intervs <- gintervals(1, 0, 1000)
gtrack.export_bedgraph("dense_track", "/tmp/dense_chr1.bedgraph",
```

```
    intervals = intervs
)

# Export a track expression
gtrack.export_bedgraph("dense_track * 2", "/tmp/scaled.bedgraph",
    iterator = 100
)

# Export compressed
gtrack.export_bedgraph("dense_track", "/tmp/dense.bedgraph.gz")

## End(Not run)
```

`gtrack.export_bigwig` *Export a track to BigWig format*

Description

Exports a track or track expression to BigWig format by first creating a temporary bedGraph file and then converting it using `bedGraphToBigWig` (or `wigToBigWig` as a fallback).

Usage

```
gtrack.export_bigwig(track, file, intervals = NULL, iterator = NULL)
```

Arguments

`track`	track name or track expression (character string)
`file`	output file path (typically ending in `.bw` or `.bigwig`).
`intervals`	genomic intervals to export. If `NULL` (default), the entire genome (`.misha$ALLGENOME`) is used.
`iterator`	iterator bin size. If `NULL` (default), the iterator is determined automatically from the track expression.

Details

This function requires the UCSC `bedGraphToBigWig` utility to be installed and available on the system PATH, or bundled with the misha package. If not found, the function will raise an error with installation instructions.

Value

`NULL` (invisible). Called for its side effect of writing a file.

See Also

`gextract`, `gtrack.export_bedgraph`, `gtrack.info`

Examples

```
## Not run:
gdb.init_examples()

# Export to BigWig (requires bedGraphToBigWig)
gtrack.export_bigwig("dense_track", "/tmp/dense.bw")

# With specific region
gtrack.export_bigwig("dense_track", "/tmp/dense_chr1.bw",
    intervals = gintervals(1, 0, 1e6)
)

## End(Not run)
```

---

gtrack.import *Creates a track from WIG / BigWig / BedGraph / BED / tab-delimited file*

Description

Creates a track from WIG / BigWig / BedGraph / BED / tab-delimited file

Usage

```
gtrack.import(
  track = NULL,
  description = NULL,
  file = NULL,
  binsize = NULL,
  defval = NaN,
  attrs = NULL
)
```

Arguments

track	track name
description	a character string description
file	file path
binsize	bin size of the newly created 'Dense' track or '0' for a 'Sparse' track
defval	default track value
attrs	a named vector or list of attributes to be set on the track after import

Details

This function creates a track from WIG / BigWig / BedGraph / tab-delimited file. Zipped files are supported (file name must have '.gz' or '.zip' suffix).

Tab-delimited files must start with a header line with the following column names (tab-separated): 'chrom', 'start', 'end', and exactly one value column name (e.g. 'value'). Each subsequent line provides a single interval: - chrom: chromosome name (e.g. 'chr1') - start: 0-based start coordinate (inclusive) - end: 0-based end coordinate (exclusive) - value: numeric value (floating point allowed); exactly one value column is supported

Columns must be separated by tabs. Coordinates must refer to chromosomes existing in the current genome. Missing values can be specified as 'NaN'.

Chromosome names are matched against the genome database through the chromosome alias chain (`chr1 <-> 1`, `M <-> MT`, and `<groot>/chrom_aliases.tsv` if present). If the file names chromosomes and none of them can be matched, 'gtrack.import' fails instead of creating an empty track; a file with no chromosome records at all is not an error and still imports as an empty track. If only some of the names can be matched, the rest of the file is imported and the unmatched names are reported: a message for contigs the database does not have (unplaced scaffolds, patches), a warning when the unmatched name looks like a primary chromosome, since that suggests a naming mismatch.

BED files (.bed/.bed.gz/.bed.zip) are also supported. If the BED 'score' column (5th column) exists and is numeric, it is used as the interval value; otherwise a constant value of 1 is used. For BED inputs, 'binsize' controls the output type: if 'binsize' is 0 the track is 'Sparse'; otherwise the track is 'Dense' with bin-averaged values based on overlaps with BED intervals (and 'defval' for regions not covered).

If 'binsize' is 0 the resulted track is created in 'Sparse' format. Otherwise the 'Dense' format is chosen with a bin size equal to 'binsize'. The values that were not defined in input file file are substituted by 'defval' value.

'description' is added as a track attribute.

When multiple databases are connected via `gsetroot`, the track is created in the current working directory (.misha$GWD), which defaults to the last connected database. Use `gdir.cd` with an absolute path to change where new tracks are created.

Value

None.

See Also

`gtrack.import_set`, `gtrack.rm`, `gtrack.info`, `gdir.create`, `gextract`

Examples

```
gdb.init_examples()

# Create a simple WIG file for demonstration
temp_file <- tempfile(fileext = ".wig")
```

```
writeLines(c(
    "track type=wiggle_0 name=\"example track\"",
    "fixedStep chrom=chr1 start=1 step=1",
    "1.5",
    "2.0",
    "1.8",
    "3.2"
), temp_file)

# Basic import
gtrack.import("example_track", "Example track from WIG file",
    temp_file,
    binsize = 1
)
gtrack.info("example_track")
gtrack.rm("example_track", force = TRUE)

# Import with custom attributes
attrs <- c("author" = "researcher", "version" = "1.0", "experiment" = "test")
gtrack.import("example_track_with_attrs", "Example track with attributes",
    temp_file,
    binsize = 1, attrs = attrs
)

# Check that attributes were set
gtrack.attr.get("example_track_with_attrs", "author")
gtrack.attr.get("example_track_with_attrs", "version")
gtrack.attr.get("example_track_with_attrs", "experiment")

# Clean up
gtrack.rm("example_track_with_attrs", force = TRUE)
```

`gtrack.import_mappedseq` *Creates a track from a file of mapped sequences*

Description

Creates a track from a file of mapped sequences.

Usage

```
gtrack.import_mappedseq(
  track = NULL,
  description = NULL,
  file = NULL,
  pileup = 0,
  binsize = -1,
```

```
  cols.order = c(9, 11, 13, 14),
  remove.dups = TRUE
)
```

Arguments

track	track name
description	a character string description
file	path to the mapped sequences file (see Description for accepted formats)
pileup	interval expansion
binsize	bin size of a dense track
cols.order	order of sequence, chromosome, coordinate and strand columns in mapped sequences file or NULL if SAM file is used. For BAM input the only accepted values are 'NULL' or omission; an explicit non-NULL 'cols.order' with BAM input is an error.
remove.dups	if 'TRUE' the duplicated coordinates are counted only once.

Details

This function creates a track from a file of mapped sequences. The file can be in SAM format, in a general TAB delimited text format where each line describes a single read, in gzipped variants of either ('.sam.gz', '.tsv.gz'), or in BAM format (auto-detected by bgzip magic; requires 'samtools' on 'PATH').

For a SAM file 'cols.order' must be set to 'NULL'. For BAM input the default 'cols.order = c(9, 11, 13, 14)' is treated as SAM mode because 'samtools view' emits SAM-format payload; passing a non-NULL 'cols.order' explicitly with BAM input is an error.

For a general TAB delimited text format the following columns must be presented in the file: sequence, chromosome, coordinate and strand. The position of these columns should be specified in 'cols.order' argument. The default value of 'cols.order' is an array of (9, 11, 13, 14) meaning that sequence is expected to be found at column number 9, chromosome - at column 11, coordinate - at column 13 and strand - at column 14. The column indices are 1-based, i.e. the first column is referenced by 1. Chromosome needs a prefix 'chr' e.g. 'chr1'. Valid strand values are '+' or 'F' for forward strand and '-' or 'R' for the reverse strand.

Each read at given coordinate can be "expanded" to cover an interval rather than a single point. The length of the interval is controlled by 'pileup' argument. The direction of expansion depends on the strand value. If 'pileup' is '0', no expansion is performed and the read is converted to a single point. The track is created in sparse format. If 'pileup' is greater than zero, the output track is in dense format. 'binsize' controls the bin size of the dense track.

If 'remove.dups' is 'TRUE' the duplicated coordinates are counted only once.

'description' is added as a track attribute.

'gtrack.import_mappedseq' returns the statistics of the conversion process.

Value

A list of conversion process statistics.

See Also

gtrack.rm, gtrack.info, gdir.create

---

gtrack.import_set *Creates one or more tracks from multiple WIG / BigWig / BedGraph / tab-delimited files on disk or FTP*

Description

Creates one or more tracks from WIG / BigWig / BedGraph / tab-delimited files on disk or FTP.

Usage

```
gtrack.import_set(
  description = NULL,
  path = NULL,
  binsize = NULL,
  track.prefix = NULL,
  defval = NaN
)
```

Arguments

description	a character string description
path	file path or URL (may contain wildcards)
binsize	bin size of the newly created 'Dense' track or '0' for a 'Sparse' track
track.prefix	prefix for a track name
defval	default track value

Details

This function is similar to 'gtrack.import' however unlike the latter it can create multiple tracks. Additionally the files can be fetched from an FTP server.

The files are expected to be in WIG / BigWig / BedGraph / tab-delimited formats. One can learn about the format of the tab-delimited file by running 'gextract' function with a 'file' parameter set to the name of the file. Zipped files are supported (file name must have '.gz' or '.zip' suffix).

Files are specified by 'path' argument. 'path' can be also a URL of an FTP server in the form of 'ftp://[address]/[files]'. If 'path' is a URL, the files are first downloaded from FTP server to a temporary directory and then imported to tracks. The temporary directory is created at 'GROOT/downloads'.

Regardless whether 'path' is file path or to a URL, it can contain wildcards. Hence multiple files can be imported (and downloaded) at once.

If 'binsize' is 0 the resulted tracks are created in 'Sparse' format. Otherwise the 'Dense' format is chosen with a bin size equal to 'binsize'. The values that were not defined in input file file are substituted by 'defval' value.

The name of a each created track is of '[track.prefix][filename]' form, where 'filename' is the name of the WIG file. For example, if 'track.prefix' equals to "wigs."" and an input file name is 'mydata', a track named 'wigs.mydata' is created. If 'track.prefix' is 'NULL' no prefix is appended to the name of the created track.

Existing tracks are not overwritten and no new directories are automatically created.

'description' is added to the created tracks as an attribute.

'gtrack.import_set' does not stop if an error occurs while importing a file. It rather continues importing the rest of the files.

'gtrack.import_set' returns the names of the files that were successfully imported and those that failed.

Value

Names of files that were successfully imported and those that failed.

See Also

gtrack.import, gwget, gtrack.rm, gtrack.info, gdir.create, gextract

---

`gtrack.info` *Returns information about a track*

Description

Returns information about a track.

Usage

```
gtrack.info(track = NULL, validate = FALSE)
```

Arguments

track	track name
validate	if TRUE, validates the track index file integrity (for indexed tracks). Default: FALSE

Details

Returns information about the track (type, dimensions, size in bytes, etc.). The fields in the returned value vary depending on the type of the track.

Value

A list that contains track properties

See Also

gtrack.exists, gtrack.ls

Examples

```
gdb.init_examples()
gtrack.info("dense_track")
gtrack.info("rects_track")
```

---

gtrack.liftover *Imports a track from another assembly*

Description

Imports a track from another assembly.

Usage

```
gtrack.liftover(
  track = NULL,
  description = NULL,
  src.track.dir = NULL,
  chain = NULL,
  src_overlap_policy = "error",
  tgt_overlap_policy = "auto",
 multi_target_agg = c("mean", "median", "sum", "min", "max", "count", "first", "last",
    "nth", "max.coverage_len", "min.coverage_len", "max.coverage_frac",
    "min.coverage_frac"),
  params = NULL,
  na.rm = TRUE,
  min_n = NULL,
  min_score = NULL
)
```

Arguments

track	name of a created track
description	a character string description
src.track.dir	path to the directory of the source track
chain	name of chain file or data frame as returned by 'gintervals.load_chain'
src_overlap_policy	policy for handling source overlaps: "error" (default), "keep", or "discard". "keep" allows one source interval to map to multiple target intervals, "discard" discards all source intervals that have overlaps and "error" throws an error if source overlaps are detected.

`tgt_overlap_policy`
: policy for handling target overlaps. One of:

Policy	Description
error	Throws an error if any target overlaps are detected.
auto	Default. Alias for "auto_score".
auto_score	Resolves overlaps by segmenting the target region and selecting the best chain for each segment based c
auto_longer	Resolves overlaps by segmenting and selecting the chain with the longest span for each segment. Tie-br
auto_first	Resolves overlaps by segmenting and selecting the chain with the lowest chain_id for each segment.
keep	Preserves all overlapping intervals.
discard	Discards any chain interval that has a target overlap with another chain interval.
agg	Segments overlaps into smaller disjoint regions where each region contains all contributing chains, allow
best_source_cluster	Best source cluster strategy based on source overlap. When multiple chains map a source interval, cluste

`multi_target_agg`
: aggregation/selection policy for contributors that land on the same target locus. When multiple source intervals map to overlapping regions in the target genome (after applying tgt_overlap_policy), their values must be combined into a single value.

`params`
: additional parameters for aggregation (e.g., for "nth" aggregation)

`na.rm`
: logical indicating whether NA values should be removed before aggregation (default: TRUE)

`min_n`
: minimum number of non-NA values required for aggregation. If fewer values are available, the result will be NA.

`min_score`
: optional minimum alignment score threshold. Chains with scores below this value are filtered out. Useful for excluding low-quality alignments.

Details

This function imports a track located in 'src.track.dir' of another assembly to the current database. Chain file instructs how the conversion of coordinates should be done. It can be either a name of a chain file or a data frame in the same format as returned by 'gintervals.load_chain' function. The name of the newly created track is specified by 'track' argument and 'description' is added as a track attribute.

Note: When passing a pre-loaded chain (data frame), overlap policies cannot be specified - they are taken from the chain's attributes that were set during loading. When passing a chain file path, policies can be specified and will be used for loading. Aggregation parameters (multi_target_agg, params, na.rm, min_n) can always be specified regardless of chain type.

Value

None.

Note

Terminology note for UCSC chain format users: In the UCSC chain format specification, the fields prefixed with 't' (tName, tStart, tEnd, etc.) are called "target" or "reference", while fields prefixed

with 'q' (qName, qStart, qEnd, etc.) are called "query". However, misha uses reversed terminology: UCSC's "target/reference" corresponds to misha's "source" (chromsrc, startsrc, endsrc), and UCSC's "query" corresponds to misha's "target" (chrom, start, end).

See Also

gintervals.load_chain, gintervals.liftover

`gtrack.lookup` *Creates a new track from a lookup table based on track expression*

Description

Evaluates track expression and translates the values into bin indices that are used in turn to retrieve values from a lookup table and create a track.

Usage

```
gtrack.lookup(
  track = NULL,
  description = NULL,
  lookup_table = NULL,
  ...,
  include.lowest = FALSE,
  force.binning = TRUE,
  iterator = NULL,
  band = NULL
)
```

Arguments

`track`	track name
`description`	a character string description
`lookup_table`	a multi-dimensional array containing the values that are returned by the function
`...`	pairs of track expressions and breaks
`include.lowest`	if 'TRUE', the lowest value of the range determined by breaks is included
`force.binning`	if 'TRUE', the values smaller than the minimal break will be translated to index 1, and the values that exceed the maximal break will be translated to index N-1 where N is the number of breaks. If 'FALSE' the out-of-range values will produce NaN values.
`iterator`	track expression iterator. If 'NULL' iterator is determined implicitly based on track expressions.
`band`	track expression band. If 'NULL' no band is used.

Details

This function evaluates the track expression for all iterator intervals and translates this value into an index based on the breaks. This index is then used to address the lookup table and create with its values a new track. More than one 'expr'-'breaks' pair can be used. In that case 'lookup_table' is addressed in a multidimensional manner, i.e. 'lookup_table[i1, i2, ...]'.

The range of bins is determined by 'breaks' argument. For example: 'breaks = c(x1, x2, x3, x4)' represents three different intervals (bins): (x1, x2], (x2, x3], (x3, x4].

If 'include.lowest' is 'TRUE' the the lowest value is included in the first interval, i.e. in [x1, x2].

'force.binning' parameter controls what should be done when the value of 'expr' exceeds the range determined by 'breaks'. If 'force.binning' is 'TRUE' then values smaller than the minimal break will be translated to index 1, and the values exceeding the maximal break will be translated to index 'M-1' where 'M' is the number of breaks. If 'force.binning' is 'FALSE' the out-of-range values will produce 'NaN' values.

Regardless of 'force.binning' value if the value of 'expr' is 'NaN' then the value in the track would be 'NaN' too.

'description' is added as a track attribute.

Value

None.

See Also

glookup, gtrack.2d.create, gtrack.create_sparse, gtrack.smooth, gtrack.modify, gtrack.rm, gtrack.info, gdir.create

Examples

```
gdb.init_examples()

## one-dimensional example
breaks1 <- seq(0.1, 0.2, length.out = 6)
gtrack.lookup(
    "lookup_track", "Test track", 1:5, "dense_track",
    breaks1
)
gtrack.rm("lookup_track", force = TRUE)

## two-dimensional example
t <- array(1:15, dim = c(5, 3))
breaks2 <- seq(0.31, 0.37, length.out = 4)
gtrack.lookup(
    "lookup_track", "Test track", t, "dense_track",
    breaks1, "2 * dense_track", breaks2
)
gtrack.rm("lookup_track", force = TRUE)
```

`gtrack.ls` *Returns a list of track names*

Description

Returns a list of track names in Genomic Database.

Usage

```
gtrack.ls(
  ...,
  pattern = NULL,
  db = NULL,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE
)
```

Arguments

`...`	additional arguments of either form 'pattern' (matched against track names) or 'attribute = pattern' (matched against a track attribute value)
`pattern`	optional pattern to match against track names, equivalent to passing it positionally (e.g. `gtrack.ls("den*")` and `gtrack.ls(pattern = "den*")` are the same). Note that this means a track attribute literally named "pattern" can no longer be filtered by name via `...`; use `gtrack.ls()` followed by `gtrack.attr.get` instead.
`db`	optional database path to filter tracks. If specified, only tracks from that database are returned.
`ignore.case`, `perl`, `fixed`, `useBytes`	see 'grep'

Details

This function returns a list of tracks whose name or track attribute value match a pattern (see 'grep'). If called without any arguments all tracks are returned.

If pattern is specified without a track attribute (i.e. in the form of 'pattern', either positionally or via the 'pattern' argument) then filtering is applied to the track names. If pattern is supplied with a track attribute (i.e. in the form of 'name = pattern') then track attribute is matched against the pattern.

Multiple patterns are applied one after another. The resulted list of tracks should match all the patterns.

When multiple databases are connected, the 'db' parameter can be used to filter tracks to only those from a specific database.

Value

An array that contains the names of tracks that match the supplied patterns.

See Also

grep, gtrack.exists, gtrack.create, gtrack.rm, gtrack.dataset

Examples

```
gdb.init_examples()

# get all track names
gtrack.ls()

# get track names that match the pattern "den*"
gtrack.ls("den*")

# equivalent, using the 'pattern' argument
gtrack.ls(pattern = "den*")

# get track names whose "created.by" attribute match the pattern
# "create_sparse"
gtrack.ls(created.by = "create_sparse")

# get track names whose names match the pattern "den*" and whose
# "created.by" attribute match the pattern "track"
gtrack.ls("den*", created.by = "track")
```

---

gtrack.modify *Modifies track contents*

Description

Modifies 'Dense' track contents.

Usage

```
gtrack.modify(track = NULL, expr = NULL, intervals = NULL)
```

Arguments

track	track name
expr	track expression
intervals	genomic scope for which track is modified

Details

This function modifies the contents of a 'Dense' track by the values of 'expr'. 'intervals' argument controls which portion of the track is modified. The iterator policy is set internally to the bin size of the track.

The new values are written to a staging copy of the track's data files and swapped in only once the whole modification has succeeded, so an interrupt or an error leaves the track exactly as it was. The staging copy needs as much free space as the part of the track being rewritten, which on a database in indexed format is the track's whole data file.

Value

None.

See Also

gtrack.create, gtrack.rm

Examples

```
gdb.init_examples()
intervs <- gintervals(1, 300, 800)
gextract("dense_track", intervs)
gtrack.modify("dense_track", "dense_track * 2", intervs)
gextract("dense_track", intervs)
gtrack.modify("dense_track", "dense_track / 2", intervs)
```

---

gtrack.mv *Renames or moves a track*

Description

Renames a track or moves it to a different namespace within the same database.

Usage

```
gtrack.mv(src = NULL, dest = NULL)
```

Arguments

src	source track name
dest	destination track name

Details

This function renames a track or moves it to a different namespace (directory) within the same database. The track cannot be moved to a different database. Use gtrack.copy followed by gtrack.rm if you need to move a track between databases.

Value

None.

See Also

gtrack.copy, gtrack.rm, gtrack.exists, gtrack.ls

Examples

```
gdb.init_examples()
gtrack.create_sparse("test_track", "Test", gintervals(1, 0, 100), 1)
gtrack.mv("test_track", "renamed_track")
gtrack.exists("renamed_track")
gtrack.rm("renamed_track", force = TRUE)
```

---

gtrack.path *Returns the path on disk of a track*

Description

Returns the path on disk of a track.

Usage

```
gtrack.path(track = NULL)
```

Arguments

track	track name or a vector of track names

Details

This function returns the actual file system path where a track is stored. The function works with a single track name or a vector of track names.

Value

A character vector containing the full paths to the tracks on disk.

See Also

gtrack.exists, gtrack.ls, gintervals.path

Examples

```
gdb.init_examples()
gtrack.path("dense_track")
gtrack.path(c("dense_track", "sparse_track"))
```

---

gtrack.rm *Deletes a track*

Description

Deletes a track.

Usage

```
gtrack.rm(track = NULL, force = FALSE, db = NULL)
```

Arguments

track	track name
force	if 'TRUE', suppresses user confirmation of a named track removal
db	optional database path to delete the track from when multiple databases are connected

Details

This function deletes a track from the Genomic Database. By default 'gtrack.rm' requires the user to interactively confirm the deletion. Set 'force' to 'TRUE' to suppress the user prompt.

Value

None.

See Also

gtrack.exists, gtrack.ls, gtrack.create, gtrack.2d.create, gtrack.create_sparse, gtrack.smooth

Examples

```
gdb.init_examples()
gtrack.create("new_track", "Test track", "2 * dense_track")
gtrack.exists("new_track")
gtrack.rm("new_track", force = TRUE)
gtrack.exists("new_track")
```

`gtrack.smooth` *Creates a new track from smoothed values of track expression*

Description

Creates a new track from smoothed values of track expression.

Usage

```
gtrack.smooth(
  track = NULL,
  description = NULL,
  expr = NULL,
  winsize = NULL,
  weight_thr = 0,
  smooth_nans = FALSE,
  alg = "LINEAR_RAMP",
  iterator = NULL
)
```

Arguments

`track`	track name
`description`	a character string description
`expr`	track expression
`winsize`	size of smoothing window
`weight_thr`	smoothing weight threshold
`smooth_nans`	if 'FALSE' track value is always set to 'NaN' if central window value is 'NaN', otherwise it is calculated from the rest of non 'NaN' values
`alg`	smoothing algorithm - "MEAN" or "LINEAR_RAMP"
`iterator`	track expression iterator of 'Fixed bin' type

Details

This function creates a new 'Dense' track named 'track'. The values of the track are results of smoothing the values of 'expr'.

Each track value at coordinate 'C' is determined by smoothing non 'NaN' values of 'expr' over the window around 'C'. The window size is controlled by 'winsize' and is given in coordinate units (not in number of bins), defining the total regions to be considered when smoothing (on both sides of the central point). Two different algorithms can be used for smoothing:

"MEAN" - an arithmetic average.

"LINEAR_RAMP" - a weighted arithmetic average, where the weights linearly decrease as the distance from the center of the window increases.

'weight_thr' determines the function behavior when some of the values in the window are missing or 'NaN' (missing values may occur at the edges of each chromosome when the window covers an area beyond chromosome boundaries). 'weight_thr' sets the weight sum threshold below which smoothing algorithm returns 'NaN' rather than a smoothing value based on non 'NaN' values in the window.

'smooth_nans' controls what would be the smoothed value if the central value in the window is 'NaN'. If 'smooth_nans' is 'FALSE' then the smoothed value is set to 'NaN' regardless of 'weight_thr' parameter. Otherwise it is calculated normally.

'description' is added as a track attribute.

Iterator policy must be of "fixed bin" type.

Value

None.

See Also

gtrack.create, gtrack.2d.create, gtrack.create_sparse, gtrack.modify, gtrack.rm, gtrack.info, gdir.create

Examples

```
gdb.init_examples()
gtrack.smooth("smoothed_track", "Test track", "dense_track", 500)
gextract("dense_track", "smoothed_track", gintervals(1, 0, 1000))
gtrack.rm("smoothed_track", force = TRUE)
```

---

gtrack.var.get *Returns value of a track variable*

Description

Returns value of a track variable.

Usage

```
gtrack.var.get(track = NULL, var = NULL)
```

Arguments

track	track name
var	track variable name

Details

This function returns the value of a track variable. If the variable does not exist an error is reported.

Value

Track variable value.

See Also

gtrack.var.set, gtrack.var.ls, gtrack.var.rm

Examples

```
gdb.init_examples()
gtrack.var.set("sparse_track", "test_var", 1:10)
gtrack.var.get("sparse_track", "test_var")
gtrack.var.rm("sparse_track", "test_var")
```

---

gtrack.var.ls *Returns a list of track variables for a track*

Description

Returns a list of track variables for a track.

Usage

```
gtrack.var.ls(
  track = NULL,
  pattern = "",
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE
)
```

Arguments

track	track name
pattern, ignore.case, perl, fixed, useBytes	see 'grep'

Details

This function returns a list of track variables of a track that match the pattern (see 'grep'). If called without any arguments all track variables of a track are returned.

Value

An array that contains the names of track variables.

See Also

grep, gtrack.var.get, gtrack.var.set, gtrack.var.rm

Examples

```
gdb.init_examples()
gtrack.var.ls("sparse_track")
gtrack.var.set("sparse_track", "test_var1", 1:10)
gtrack.var.set("sparse_track", "test_var2", "v")
gtrack.var.ls("sparse_track")
gtrack.var.ls("sparse_track", pattern = "2")
gtrack.var.rm("sparse_track", "test_var1")
gtrack.var.rm("sparse_track", "test_var2")
```

---

gtrack.var.rm *Deletes a track variable*

Description

Deletes a track variable.

Usage

```
gtrack.var.rm(track = NULL, var = NULL)
```

Arguments

track	track name
var	track variable name

Details

This function deletes a track variable.

Value

None.

See Also

gtrack.var.get, gtrack.var.set, gtrack.var.ls

Examples

```
gdb.init_examples()
gtrack.var.set("sparse_track", "test_var1", 1:10)
gtrack.var.set("sparse_track", "test_var2", "v")
gtrack.var.ls("sparse_track")
gtrack.var.rm("sparse_track", "test_var1")
gtrack.var.rm("sparse_track", "test_var2")
gtrack.var.ls("sparse_track")
```

---

gtrack.var.set *Assigns value to a track variable*

Description

Assigns value to a track variable.

Usage

```
gtrack.var.set(track = NULL, var = NULL, value = NULL)
```

Arguments

track	track name
var	track variable name
value	value

Details

This function creates a track variable and assigns 'value' to it. If the track variable already exists its value is overwritten.

Value

None.

See Also

gtrack.var.get, gtrack.var.ls, gtrack.var.rm

Examples

```
gdb.init_examples()
gtrack.var.set("sparse_track", "test_var", 1:10)
gtrack.var.get("sparse_track", "test_var")
gtrack.var.rm("sparse_track", "test_var")
```

`gvtrack.array.slice` *Defines rules for a single value calculation of a virtual 'Array' track*

Description

Defines how a single value within an interval is achieved for a virtual track based on 'Array' track.

Usage

```
gvtrack.array.slice(vtrack = NULL, slice = NULL, func = "avg", params = NULL)
```

Arguments

`vtrack`	virtual track name
`slice`	a vector of column names or column indices or 'NULL'
`func, params`	see below

Details

A track (regular or virtual) used in a track expression is expected to return one value for each track interval. 'Array' tracks store multiple values per interval (one for each 'column') and hence if used in a track expression one must define the way of how a single value should be deduced from several ones.

By default if an 'Array' track is used in a track expressions, its interval value would be the average of all column values that are not NaN. 'gvtrack.array.slice' allows to select specific columns and to specify the function applied to their values.

'slice' parameter allows to choose the columns. Columns can be indicated by their names or their indices. If 'slice' is 'NULL' the non-NaN values of all track columns are used.

'func' parameter determines the function applied to the columns' values. Use the following table for a reference of all valid functions and parameters combinations:

func = "avg", params = NULL
Average of columns' values.

func = "max", params = NULL
Maximum of columns' values.

func = "min", params = NULL
Minimum of columns' values.

func = "stdev", params = NULL
Unbiased standard deviation of columns' values.

func = "sum", params = NULL
Sum of columns' values.

func = "quantile", params = [Percentile in the range of [0, 1]]
Quantile of columns' values.

Value

None.

See Also

gvtrack.create, gtrack.array.get_colnames, gtrack.array.extract

Examples

```
gdb.init_examples()
gvtrack.create("vtrack1", "array_track")
gvtrack.array.slice("vtrack1", c("col2", "col4"), "max")
gextract("vtrack1", gintervals(1, 0, 1000))
```

---

gvtrack.clear *Deletes all virtual tracks*

Description

Deletes all virtual tracks of the current working directory.

Usage

```
gvtrack.clear()
```

Details

This function removes every virtual track defined for the current working directory in one call. After it returns gvtrack.ls reports no virtual tracks. It is a convenient way to reset virtual-track state between analyses. Use gvtrack.rm to remove a single virtual track.

Value

None.

See Also

gvtrack.rm, gvtrack.ls, gvtrack.create

Examples

```
gdb.init_examples()
gvtrack.create("vtrack1", "dense_track", "max")
gvtrack.create("vtrack2", "dense_track", "avg")
gvtrack.ls()
gvtrack.clear()
gvtrack.ls()
```

`gvtrack.create` *Creates a new virtual track*

Description

Creates a new virtual track.

Usage

```
gvtrack.create(
  vtrack = NULL,
  src = NULL,
  func = NULL,
  params = NULL,
  dim = NULL,
  sshift = NULL,
  eshift = NULL,
  filter = NULL,
  ...
)
```

Arguments

`vtrack`	virtual track name
`src`	source (track/intervals). NULL for PWM functions. For value-based tracks, provide a data frame with columns `chrom`, `start`, `end`, and one numeric value column. The data frame functions as an in-memory sparse track and supports all track-based summarizer functions. Intervals must not overlap.
`func`	function name (see above)
`params`	function parameters (see above)
`dim`	use 'NULL' or '0' for 1D iterators. '1' converts 2D iterator to (chrom1, start1, end1) , '2' converts 2D iterator to (chrom2, start2, end2)
`sshift`	shift of 'start' coordinate
`eshift`	shift of 'end' coordinate
`filter`	genomic mask to apply. Can be:

- A data.frame with columns 'chrom', 'start', 'end' (intervals to mask)
- A character string naming an intervals set
- A character string naming a track (must be intervals-type track)
- A list of any combination of the above (all will be unified)
- NULL to clear the filter

... additional named parameters for functions that accept them this way instead of `params` - currently the PWM, k-mer and edit-distance families, and `neighbor.count`. Use one style or the other: supplying `params` together with named `...` arguments in the same call is ambiguous (the named arguments would be ignored) and raises an error. Parameter names are validated against the accepted names for `func`, whether they arrive through `params` or through `...`; an unrecognised name, or any name passed via `...` for a `func` that takes no additional named parameters, raises an error naming the offending argument and the accepted ones. `masked.count` / `masked.frac` are the exception: they warn and ignore the extra arguments.

Details

This function creates a new virtual track named 'vtrack' with the given source, function and parameters. 'src' can be either a track, intervals (1D or 2D), or a data frame with intervals and a numeric value column (value-based track). The tables below summarize the supported combinations.

Value-based tracks Value-based tracks are data frames containing genomic intervals with associated numeric values. They function as in-memory sparse tracks without requiring track creation in the database. To create a value-based track, provide a data frame with columns `chrom`, `start`, `end`, and one numeric value column (any name is acceptable). Value-based tracks support all track-based summarizer functions (e.g., `avg`, `min`, `max`, `sum`, `lse`, `stddev`, `quantile`, `nearest`, `exists`, `size`, `first`, `last`, `sample`, and position functions), but do not support overlapping intervals. They behave like sparse tracks in aggregation: values are aggregated using count-based averaging (each interval contributes equally regardless of length), not coverage-based averaging.

Track-based summarizers

Source	func	params	Description
Track	avg	NULL	Average track value in the iterator interval.
Track (1D)	exists	vals (optional)	Returns 1 if any value exists (or specific vals if p
Track (1D)	first	NULL	First value in the iterator interval.
Track (1D)	last	NULL	Last value in the iterator interval.
Track	max	NULL	Maximum track value in the iterator interval.
Track	min	NULL	Minimum track value in the iterator interval.
Dense / Sparse / Array track	nearest	NULL	Average value inside the iterator; for sparse track
Track (1D)	sample	NULL	Uniformly sampled source value from the iterato
Track (1D)	size	NULL	Number of non-NaN values in the iterator interva
Dense / Sparse / Array track	stddev	NULL	Unbiased standard deviation of values in the itera
Dense / Sparse / Array track	sum	NULL	Sum of values in the iterator interval.
Dense / Sparse / Array track	lse	NULL	Log-sum-exp of track values in the iterator interv
Dense / Sparse / Array track	quantile	Percentile in [0, 1]	Quantile of values in the iterator interval.
Dense track	global.percentile	NULL	Percentile of the interval average relative to the f
Dense track	global.percentile.max	NULL	Percentile of the interval maximum relative to th

Dense track	global.percentile.min	NULL	Percentile of the interval minimum relative to the

Track position summarizers

Source	func	params	Description
Track (1D)	first.pos.abs	NULL	Absolute genomic coordinate of the first value.
Track (1D)	first.pos.relative	NULL	Zero-based position (relative to interval start) of the first value.
Track (1D)	last.pos.abs	NULL	Absolute genomic coordinate of the last value.
Track (1D)	last.pos.relative	NULL	Zero-based position (relative to interval start) of the last value.
Track (1D)	max.pos.abs	NULL	Absolute genomic coordinate of the maximum value inside the iterator interval.
Track (1D)	max.pos.relative	NULL	Zero-based position (relative to interval start) of the maximum value.
Track (1D)	min.pos.abs	NULL	Absolute genomic coordinate of the minimum value inside the iterator interval.
Track (1D)	min.pos.relative	NULL	Zero-based position (relative to interval start) of the minimum value.
Track (1D)	sample.pos.abs	NULL	Absolute genomic coordinate of a uniformly sampled value.
Track (1D)	sample.pos.relative	NULL	Zero-based position (relative to interval start) of a uniformly sampled value.

For `max.pos.relative`, `min.pos.relative`, `first.pos.relative`, `last.pos.relative`, `sample.pos.relative`, iterator modifiers (including `sshift` / `eshift` and 1D projections generated via `gvtrack.iterator`) are applied before the position is reported. In other words, the returned coordinate is always 0-based and measured from the start of the iterator interval after all modifier adjustments.

Interval-based summarizers

Source	func	params	Description
1D intervals	distance	Minimal distance from center (default 0)	Signed distance using normalized formula when i
1D intervals	distance.center	NULL	Distance from iterator center to the closest interva
1D intervals	distance.edge	NULL	Edge-to-edge distance from iterator interval to clo
1D intervals	coverage	NULL	Fraction of iterator length covered by source inter
1D intervals	neighbor.count	Max distance (>= 0)	Number of source intervals whose edge-to-edge d

2D track summarizers

Source	func	params	Description
2D track	area	NULL	Area covered by intersections of track rectangles with the iterator interval.
2D track	weighted.sum	NULL	Weighted sum of values where each weight equals the intersection area.

Motif (PWM) summarizers

Source	func	Key params	Description
NULL (sequence)	pwm	pssm, bidirect, prior, extend, spat_*	Log-sum-exp scor
NULL (sequence)	pwm.max	pssm, bidirect, prior, extend, spat_*	Maximum log-like
NULL (sequence)	pwm.max.pos	pssm, bidirect, prior, extend, spat_*	1-based position o
NULL (sequence)	pwm.count	pssm, score.thresh (required), bidirect, prior, extend, strand, spat_*	Count of anchors s

Edit distance summarizers

Source	func	Key params
NULL (sequence)	pwm.edit_distance	pssm, score.thresh, max_edits, max_indels, score.min, score.max, direction, bi
NULL (sequence)	pwm.edit_distance.pos	pssm, score.thresh, max_edits, max_indels, score.min, score.max, direction, bi
NULL (sequence)	pwm.max.edit_distance	pssm, score.thresh, max_edits, max_indels, score.min, score.max, direction, bi

Mutation count summarizers

Source	func	Key params
NULL (sequence)	pwm.n_mutations	pssm, score.thresh, score.min, score.max, direction, bidirect, prior, extend, strand

LSE edit distance summarizers

Source	func	Key params
NULL (sequence)	pwm.edit_distance.lse	pssm, score.thresh, max_edits, score.min, score.max, direction, bidirect, pri
NULL (sequence)	pwm.edit_distance.lse.pos	pssm, score.thresh, max_edits, score.min, score.max, direction, bidirect, pri

K-mer summarizers

Source	func	Key params	Description
NULL (sequence)	kmer.count	kmer, extend, strand	Number of k-mer occurrences whose anchor lies inside the iterator i
NULL (sequence)	kmer.frac	kmer, extend, strand	Fraction of possible anchors within the interval that match the k-me

Masked sequence summarizers

Source	func	Key params	Description
NULL (sequence)	masked.count	NULL	Number of masked (lowercase) base pairs in the iterator interval.
NULL (sequence)	masked.frac	NULL	Fraction of base pairs in the iterator interval that are masked (lowercase).

The sections below provide additional notes for motif, interval, k-mer, and masked sequence functions.

Motif (PWM) notes

- `pssm`: Position-specific scoring matrix (matrix or data frame) with columns `A`, `C`, `G`, `T`; extra columns are ignored.
- `bidirect`: When TRUE (default), both strands are scanned and combined per genomic start (per-position union). The `strand` argument is ignored. When FALSE, only the strand specified by `strand` is scanned.
- `prior`: Pseudocount added to frequencies (default 0.01). Set to 0 to disable.

- extend: Extends the fetched sequence so boundary-anchored motifs retain full context (default TRUE). The END coordinate is padded by motif_length - 1 for all strand modes; anchors must still start inside the iterator.
- Neutral characters (N, n, *) contribute the mean log-probability of the corresponding PSSM column on both strands.
- strand: Used only when bidirect = FALSE; 1 scans the forward strand, -1 scans the reverse strand. For pwm.max.pos, strand = -1 reports the hit position at the end of the match (still relative to the forward orientation).
- score.thresh: Threshold for pwm.count, and mandatory for it - there is no default, and it must be a single value. Anchors with log-likelihood >= score.thresh are counted; only one count per genomic start. PWM scores are log-likelihoods, so the usable range depends on the PSSM, the prior and any spatial weights; calibrate with a pwm or pwm.max virtual track over the same intervals before choosing one. pwm, pwm.max and pwm.max.pos accept score.thresh but ignore it.
- Spatial weighting (spat_factor, spat_bin, spat_min, spat_max): optional position-dependent weights applied in log-space. Provide a positive numeric vector spat_factor; spat_bin (integer > 0) defines bin width; spat_min/spat_max restrict the scanning window.
- pwm.max.pos: Positions are reported 1-based relative to the final scan window (after iterator shifts and spatial trimming). Ties resolve to the most 5' anchor; the forward strand wins ties at the same coordinate. Values are signed when bidirect = TRUE (positive for forward, negative for reverse).

Edit distance notes

The edit distance functions answer "how many single-base changes are needed to reach a target PWM score?" Each genomic window (same length as the PSSM) gets an edit distance: the minimum number of substitutions (and optionally indels) required to bring its PWM log-likelihood score to the target threshold. The virtual track returns the minimum edit distance across all windows in the iterator interval.

Direction. The direction parameter controls which side of the threshold to target:

- "above" (default): minimum edits to make the score **reach or exceed** score.thresh. Use this to ask "how close is this sequence to becoming a motif match?" A window already scoring above the threshold needs 0 edits.
- "below": minimum edits to make the score **fall below** score.thresh. Use this to ask "how fragile is this motif match — how many mutations would disrupt it?" A window already scoring below the threshold needs 0 edits.

Strand handling differs by direction. When bidirect = TRUE, both strands are scanned at each genomic position. The way their edit distances are combined depends on the direction:

- direction = "above": takes the **minimum** across both strands. A motif match on *either* strand is sufficient (e.g., for TF binding), so the easier strand wins.
- direction = "below": takes the **maximum** across both strands. A single genomic substitution changes both strands simultaneously (the forward base and its reverse complement). To truly disrupt a binding site you must bring *both* strands below the threshold, so the harder strand determines the answer. For example, if the forward strand needs 3 edits to go below the threshold and the reverse needs 1, the result is 3 — after just 1 edit the forward strand would still have a match.

Then, across all windows in the iterator interval, the minimum is reported (the position where it is easiest to create or disrupt a match).

Parameters:

- `score.thresh`: The target PWM log-likelihood score that the edit distance algorithm tries to reach; mandatory, and it must be a single value. For `direction = "above"`, this is the score to reach or exceed; for `direction = "below"`, this is the score to fall below.
- `score.min`, `score.max`: Optional numeric pre-filters on the **current** (unedited) PWM score. Windows scoring outside the [`score.min`, `score.max`] range are skipped (edit distance returns NA). Default NULL (no filter). These are independent of `score.thresh` — they control *which windows to evaluate*, not the target score.

  Typical usage with `direction = "below"`: use `score.min` to restrict to windows that currently have a motif match. For example, with `score.thresh = -18` and `score.min = -14`, only windows scoring >= -14 are considered, and the edit distance tells you how many edits to push each one below -18. Without `score.min`, windows already scoring below -18 will return 0 edits.

  For LSE variants, the filters apply to the aggregate LSE score, not individual windows.
- `max_edits`: Optional positive integer. Cap the search depth: if reaching the threshold requires more than `max_edits` edits, NA is returned for that window. Default NULL (exact computation, no cap).
- `max_indels`: Optional non-negative integer (default 0). When > 0, allows insertions and deletions in addition to substitutions, using a banded Needleman-Wunsch DP. Typical values are 1-2. Only supported by the non-LSE variants (`pwm.edit_distance`, `pwm.edit_distance.pos`, `pwm.max.edit_distance`).
- `direction`: `"above"` (default) or `"below"`. See Direction above.

Spatial weighting enables position-dependent weighting for modeling positional biases. Bins are 0-indexed from the scan start. When using `gvtrack.iterator()` shifts (e.g., `sshift = -50`, `eshift = 50`), bins index from the expanded scan window start, not the original interval. Both strands use the same bin at each genomic position. Positions beyond the last bin reuse the final bin's weight. If the window size is not divisible by `spat_bin`, the last bin is shorter (e.g., scanning 500 bp with 40 bp bins yields bins 0-11 of 40 bp plus bin 12 of 20 bp). Use `spat_min` and `spat_max` to restrict scanning to a range divisible by `spat_bin` if needed.

PWM parameters can be supplied either as a single list (`params`) or via named arguments (see examples), but not both in the same call.

Interval distance notes

`distance`: Given the center 'C' of the current iterator interval, returns 'DC * X/2' where 'DC' is the normalized distance to the center of the interval that contains 'C', and 'X' is the value of the parameter (default: 0). If no interval contains 'C', the result is 'D + X/2' where 'D' is the distance between 'C' and the edge of the closest interval.

`distance.center`: Given the center 'C' of the current iterator interval, returns `NaN` if 'C' is outside of all intervals, otherwise returns the distance between 'C' and the center of the closest interval.

`distance.edge`: Computes edge-to-edge distance from the iterator interval to the closest source interval, using the same calculation as `gintervals.neighbors`. Returns 0 for overlapping intervals. Distance sign depends on the strand column of source intervals; returns unsigned (absolute) distance if no strand column exists. Returns `NA` if no source intervals exist on the current chromosome.

For distance and distance.center, distance can be positive or negative depending on the position of the coordinate relative to the interval and the strand (-1 or 1) of the interval. Distance is always positive if strand = 0 or if the strand column is missing. The result is NA if no intervals exist for the current chromosome.

Difference between distance functions: The distance function measures from the *center* of the iterator interval (a single coordinate point) to the closest *edge* of source intervals when outside, or returns a normalized distance within the interval when inside. The distance.center function measures from the center of the iterator interval to the *center* of source intervals. The distance.edge function measures *edge-to-edge* distance between intervals, exactly like gintervals.neighbors. Use distance.edge when you need the same distance computation as gintervals.neighbors within a virtual track context.

K-mer notes

- kmer: DNA sequence (case-insensitive) to count.
- extend: If TRUE (default), counts kmers whose anchor lies in the interval even if the kmer extends beyond it; when FALSE, only kmers fully contained in the interval are considered.
- strand: 1 counts matches on the forward strand, -1 counts matches on the reverse-complement strand, 0 counts both strands and sums them (default). Unlike the PWM functions - where the both-strand mode is set by bidirect and strand must be 1 or -1 - the k-mer functions accept strand = 0 directly. For palindromic kmers (equal to their own reverse complement) use 1 or -1 to avoid double counting; strand = 0 issues a warning in that case.
- For kmer.frac the denominator is the number of possible anchor positions in the interval; with strand = 0 positions on both strands are counted, so the fraction stays in [0, 1] and remains comparable to the single-strand fraction.

K-mer parameters can be supplied as a list or via named arguments (see examples), but not both in the same call.

Modify iterator behavior with 'gvtrack.iterator' or 'gvtrack.iterator.2d'.

Value

None.

See Also

gvtrack.info, gvtrack.iterator, gvtrack.iterator.2d, gvtrack.array.slice, gvtrack.ls, gvtrack.rm

gvtrack.iterator, gvtrack.iterator.2d, gvtrack.filter

Examples

```
gdb.init_examples()

gvtrack.create("vtrack1", "dense_track", "max")
gvtrack.create("vtrack2", "dense_track", "quantile", 0.5)
gextract("dense_track", "vtrack1", "vtrack2",
    gintervals(1, 0, 10000),
```

```
    iterator = 1000
)

gvtrack.create("vtrack3", "dense_track", "global.percentile")
gvtrack.create("vtrack4", "annotations", "distance")
gdist(
    "vtrack3", seq(0, 1, l = 10), "vtrack4",
    seq(-500, 500, 200)
)

gvtrack.create("cov", "annotations", "coverage")
gextract("cov", gintervals(1, 0, 1000), iterator = 100)

pssm <- matrix(
    c(
        0.7, 0.1, 0.1, 0.1, # Example PSSM
        0.1, 0.7, 0.1, 0.1,
        0.1, 0.1, 0.7, 0.1,
        0.1, 0.1, 0.7, 0.1,
        0.1, 0.1, 0.7, 0.1,
        0.1, 0.1, 0.7, 0.1
    ),
    ncol = 4, byrow = TRUE
)
colnames(pssm) <- c("A", "C", "G", "T")
gvtrack.create(
    "motif_score", NULL, "pwm",
    list(pssm = pssm, bidirect = TRUE, prior = 0.01)
)
gvtrack.create("max_motif_score", NULL, "pwm.max",
    pssm = pssm, bidirect = TRUE, prior = 0.01
)
gvtrack.create("max_motif_pos", NULL, "pwm.max.pos",
    pssm = pssm
)
gextract(
    c(
        "dense_track", "motif_score", "max_motif_score",
        "max_motif_pos"
    ),
    gintervals(1, 0, 10000),
    iterator = 500
)

# Kmer counting examples
gvtrack.create("cg_count", NULL, "kmer.count", kmer = "CG", strand = 1)
gvtrack.create("cg_frac", NULL, "kmer.frac", kmer = "CG", strand = 1)
gextract(c("cg_count", "cg_frac"), gintervals(1, 0, 10000), iterator = 1000)

gvtrack.create("at_pos", NULL, "kmer.count", kmer = "AT", strand = 1)
gvtrack.create("at_neg", NULL, "kmer.count", kmer = "AT", strand = -1)
gvtrack.create("at_both", NULL, "kmer.count", kmer = "AT", strand = 0)
gextract(c("at_pos", "at_neg", "at_both"), gintervals(1, 0, 10000), iterator = 1000)
```

```
# GC content
gvtrack.create("g_frac", NULL, "kmer.frac", kmer = "G")
gvtrack.create("c_frac", NULL, "kmer.frac", kmer = "C")
gextract("g_frac + c_frac", gintervals(1, 0, 10000),
    iterator = 1000,
    colnames = "gc_content"
)

# Masked base pair counting
gvtrack.create("masked_count", NULL, "masked.count")
gvtrack.create("masked_frac", NULL, "masked.frac")
gextract(c("masked_count", "masked_frac"), gintervals(1, 0, 10000), iterator = 1000)

# Combined with GC content (unmasked regions only)
gvtrack.create("gc", NULL, "kmer.frac", kmer = "G")
gextract("gc * (1 - masked_frac)",
    gintervals(1, 0, 10000),
    iterator = 1000,
    colnames = "gc_unmasked"
)

# Value-based track examples
# Create a data frame with intervals and numeric values
intervals_with_values <- data.frame(
    chrom = "chr1",
    start = c(100, 300, 500),
    end = c(200, 400, 600),
    score = c(10, 20, 30)
)
# Use as value-based sparse track (functions like sparse track)
gvtrack.create("value_track", intervals_with_values, "avg")
gvtrack.create("value_track_max", intervals_with_values, "max")
gextract(c("value_track", "value_track_max"),
    gintervals(1, 0, 10000),
    iterator = 1000
)

# Spatial PWM examples
# Create a PWM with higher weight in the center of intervals
pssm <- matrix(
    c(
        0.7, 0.1, 0.1, 0.1,
        0.1, 0.7, 0.1, 0.1,
        0.1, 0.1, 0.7, 0.1,
        0.1, 0.1, 0.1, 0.7
    ),
    ncol = 4, byrow = TRUE
)
colnames(pssm) <- c("A", "C", "G", "T")

# Spatial factors: low weight at edges, high in center
# For 200bp intervals with 40bp bins: bins 0, 40, 80, 120, 160
```

```
spatial_weights <- c(0.5, 1.0, 2.0, 1.0, 0.5)

gvtrack.create(
    "spatial_pwm", NULL, "pwm",
    list(
        pssm = pssm,
        bidirect = TRUE,
        spat_factor = spatial_weights,
        spat_bin = 40L
    )
)

# Compare with non-spatial PWM
gvtrack.create(
    "regular_pwm", NULL, "pwm",
    list(pssm = pssm, bidirect = TRUE)
)

gextract(c("spatial_pwm", "regular_pwm"),
    gintervals(1, 0, 10000),
    iterator = 200
)

# Using spatial parameters with iterator shifts
gvtrack.create(
    "spatial_extended", NULL, "pwm.max",
    pssm = pssm,
    spat_factor = c(0.5, 1.0, 2.0, 2.5, 2.0, 1.0, 0.5),
    spat_bin = 40L
)
# Scan window will be 280bp (100bp + 2*90bp)
gvtrack.iterator("spatial_extended", sshift = -90, eshift = 90)
gextract("spatial_extended", gintervals(1, 0, 10000), iterator = 100)

# Using spat_min/spat_max to restrict scanning to a window
# For 500bp intervals, scan only positions 30-470 (440bp window)
gvtrack.create(
    "window_pwm", NULL, "pwm",
    pssm = pssm,
    bidirect = TRUE,
    spat_min = 30, # 1-based position
    spat_max = 470 # 1-based position
)
gextract("window_pwm", gintervals(1, 0, 10000), iterator = 500)

# Combining spatial weighting with window restriction
# Scan positions 50-450 with spatial weights favoring the center
gvtrack.create(
    "window_spatial_pwm", NULL, "pwm",
    pssm = pssm,
    bidirect = TRUE,
    spat_factor = c(0.5, 1.0, 2.0, 2.5, 2.0, 1.0, 0.5, 1.0, 0.5, 0.5),
    spat_bin = 40L,
```

```
    spat_min = 50,
    spat_max = 450
)
gextract("window_spatial_pwm", gintervals(1, 0, 10000), iterator = 500)

# --- Edit distance examples ---
# (Using the same 4-position PSSM defined above)

# First, check the PWM score distribution to pick sensible thresholds
gvtrack.create("pwm_score", NULL, "pwm.max", pssm = pssm)
head(gextract("pwm_score", gintervals(1, 500, 520), iterator = 1))
# Scores range from ~-8.3 (poor) to ~-0.8 (strong match)

# --- direction = "above": how many edits to CREATE a motif match? ---
# "How many substitutions until this window scores >= -5?"
gvtrack.create("edist_above", NULL, "pwm.edit_distance",
    pssm = pssm, score.thresh = -5
)
# At 1bp: each position gets its own edit distance (0 if already matching)
head(gextract(c("pwm_score", "edist_above"),
    gintervals(1, 500, 520),
    iterator = 1
))

# --- direction = "below": how many edits to DISRUPT a motif match? ---
#
# score.thresh is the target: "push the score below this value"
# score.min is a pre-filter: "only consider windows currently scoring above this"
# These are independent — score.min controls WHICH windows to evaluate,
# score.thresh controls WHAT score to target.
#
# Example: among windows scoring >= -5 (strong matches),
# how many edits to push below -7 (disrupt the match)?
gvtrack.create("edist_below", NULL, "pwm.edit_distance",
    pssm = pssm, score.thresh = -7, score.min = -5,
    direction = "below"
)
head(gextract(c("pwm_score", "edist_above", "edist_below"),
    gintervals(1, 500, 520),
    iterator = 1
))
# score >= -5 (e.g. -2.7): gets edit distance (2 edits to go below -7)
# score < -5 (e.g. -6.4): NA (filtered out by score.min)

# Without score.min, ALL windows are evaluated — windows already
# below -7 return 0 edits:
gvtrack.create("edist_below_all", NULL, "pwm.edit_distance",
    pssm = pssm, score.thresh = -7, direction = "below"
)
head(gextract(c("pwm_score", "edist_below", "edist_below_all"),
    gintervals(1, 500, 520),
    iterator = 1
))
```

```
# score >= -5: same result as edist_below (2 edits)
# score between -7 and -5 (e.g. -6.4): edist_below=NA, edist_below_all=0
# score < -7: both return 0 (already below threshold)

# --- max_edits: cap the search depth ---
# Return NA if more than 2 substitutions are needed
gvtrack.create("edist_max2", NULL, "pwm.edit_distance",
    pssm = pssm, score.thresh = -5, max_edits = 2
)
# Positions needing 3+ edits now return NA instead of 3
head(gextract(c("pwm_score", "edist_above", "edist_max2"),
    gintervals(1, 500, 520),
    iterator = 1
))

# --- Aggregation over intervals ---
# With a larger iterator, the minimum edit distance across the
# interval is returned.
gextract(c("pwm_score", "edist_above", "edist_below_all"),
    gintervals(1, 0, 2000),
    iterator = 200
)
```

`gvtrack.filter` *Attach or clear a genomic mask filter on a virtual track*

Description

Attaches or clears a genomic mask filter on a virtual track. When a filter is attached, the virtual track function is evaluated only over the unmasked regions (i.e., regions not covered by the filter intervals).

Usage

```
gvtrack.filter(vtrack = NULL, filter = NULL)
```

Arguments

`vtrack`	virtual track name
`filter`	genomic mask to apply. Can be: • A data.frame with columns 'chrom', 'start', 'end' (intervals to mask) • A character string naming an intervals set • A character string naming a track (must be intervals-type track) • A list of any combination of the above (all will be unified) • NULL to clear the filter

Details

The filter defines regions to *exclude* from virtual track evaluation. The virtual track function will be evaluated only on the complement of the filter. Once a filter is attached to a virtual track, it applies to **all subsequent extractions** of that virtual track until explicitly cleared with `filter = NULL`.

Order of Operations:

Filters are applied *after* iterator modifiers (sshift/eshift/dim). The order is:

1. Apply iterator modifiers (gvtrack.iterator with sshift/eshift)
2. Subtract mask from the modified intervals
3. Evaluate virtual track function over unmasked regions

Semantics by function type:

- **Aggregations (avg/sum/min/max/stddev/quantile):** Length-weighted over unmasked regions
- **coverage:** Returns (covered length in unmasked region) / (total unmasked length)
- **distance/distance.center:** Unaffected by mask (pure geometry)
- **PWM/kmer:** Masked bases act as hard boundaries; matches cannot span masked regions. **Important:** When `extend=TRUE` (the default), motifs at the boundaries of unmasked segments can use bases from the adjacent masked regions to complete the motif scoring. For example, if a 4bp motif starts at position 1998 in an unmasked region that ends at 2000, and positions 2000-2002 are masked, the motif will still be scored using the masked bases. In other words, motif matches *starting positions* must be in unmasked regions, but the motif sequence itself can extend into masked regions when `extend=TRUE`. Set `extend=FALSE` to prevent any use of masked bases in scoring.

Completely Masked Intervals: If an entire iterator interval is masked, the function returns `NA` (not 0).

Value

None (invisibly).

See Also

`gvtrack.create`, `gvtrack.iterator`, `gvtrack.info`

Examples

```
gdb.init_examples()

## Basic usage: Excluding specific regions
gvtrack.create("vtrack1", "dense_track", func = "avg")

# Create intervals to mask (e.g., repetitive regions)
repeats <- gintervals(c(1, 1), c(100, 500), c(200, 600))

# Attach filter - track will be evaluated excluding these regions
gvtrack.filter("vtrack1", filter = repeats)
```

```
# Extract values - masked regions are excluded from calculation
result_filtered <- gextract("vtrack1", gintervals(1, 0, 1000))

# Check filter info
gvtrack.info("vtrack1")

# Clear the filter and compare
gvtrack.filter("vtrack1", filter = NULL)
result_unfiltered <- gextract("vtrack1", gintervals(1, 0, 1000))

## Using multiple filter sources (combined automatically)
centromeres <- gintervals(1, 10000, 15000)
telomeres <- gintervals(1, 0, 1000)
combined_mask <- list(repeats, centromeres, telomeres)

gvtrack.filter("vtrack1", filter = combined_mask)
result_multi_filter <- gextract("vtrack1", gintervals(1, 0, 20000))

## Filters work with iterator modifiers
gvtrack.create("vtrack2", "dense_track", func = "sum")
gvtrack.filter("vtrack2", filter = repeats)
gvtrack.iterator("vtrack2", sshift = -50, eshift = 50)

# Iterator shifts applied first, then mask subtracted
result_shifted <- gextract("vtrack2", gintervals(1, 1000, 2000), iterator = 100)
```

`gvtrack.info` *Returns the definition of a virtual track*

Description

Returns the definition of a virtual track.

Usage

```
gvtrack.info(vtrack = NULL)
```

Arguments

`vtrack`	virtual track name

Details

This function returns the internal representation of a virtual track.

Value

Internal representation of a virtual track.

See Also

gvtrack.create

Examples

```
gdb.init_examples()
gvtrack.create("vtrack1", "dense_track", "max")
gvtrack.info("vtrack1")
```

---

gvtrack.iterator *Defines modification rules for a one-dimensional iterator in a virtual track*

Description

Defines modification rules for a one-dimensional iterator in a virtual track.

Usage

```
gvtrack.iterator(vtrack = NULL, dim = NULL, sshift = 0, eshift = 0)
```

Arguments

vtrack	virtual track name
dim	use 'NULL' or '0' for 1D iterators. '1' converts 2D iterator to (chrom1, start1, end1) , '2' converts 2D iterator to (chrom2, start2, end2)
sshift	shift of 'start' coordinate
eshift	shift of 'end' coordinate

Details

This function defines modification rules for one-dimensional iterator intervals in a virtual track.

'dim' converts a 2D iterator interval (chrom1, start1, end1, chrom2, start2, end2) to a 1D interval. If 'dim' is '1' the interval is converted to (chrom1, start1, end1). If 'dim' is '2' the interval is converted to (chrom2, start2, end2). If 1D iterator is used 'dim' must be set to 'NULL' or '0' (meaning: no conversion is made).

Iterator interval's 'start' coordinate is modified by adding 'sshift'. Similarly 'end' coordinate is altered by adding 'eshift'.

A shifted interval that runs past a chromosome boundary is clamped to it. If nothing is left of it - it falls entirely outside the chromosome, or the shifts collapse it to zero length - the virtual track returns 'NaN'.

Value

None.

See Also

gvtrack.create, gvtrack.iterator.2d

Examples

```
gdb.init_examples()

gvtrack.create("vtrack1", "dense_track")
gvtrack.iterator("vtrack1", sshift = 200, eshift = 200)
gextract("dense_track", "vtrack1", gintervals(1, 0, 500))

gvtrack.create("vtrack2", "dense_track")
gvtrack.iterator("vtrack2", dim = 1)
gextract("vtrack2", gintervals.2d(1, 0, 1000, 1, 0, -1),
    iterator = "rects_track"
)
```

---

gvtrack.iterator.2d *Defines modification rules for a two-dimensional iterator in a virtual track*

Description

Defines modification rules for a two-dimensional iterator in a virtual track.

Usage

```
gvtrack.iterator.2d(
  vtrack = NULL,
  sshift1 = 0,
  eshift1 = 0,
  sshift2 = 0,
  eshift2 = 0
)
```

Arguments

vtrack	virtual track name
sshift1	shift of 'start1' coordinate
eshift1	shift of 'end1' coordinate
sshift2	shift of 'start2' coordinate
eshift2	shift of 'end2' coordinate

Details

This function defines modification rules for one-dimensional iterator intervals in a virtual track.

Iterator interval's 'start1' coordinate is modified by adding 'sshift1'. Similarly 'end1', 'start2', 'end2' coordinates are altered by adding 'eshift1', 'sshift2' and 'eshift2' accordingly.

A shifted interval that runs past a chromosome boundary is clamped to it. If nothing is left of it - it falls entirely outside the chromosome, or the shifts collapse it to zero length - the virtual track returns 'NaN'.

Value

None.

See Also

gvtrack.create, gvtrack.iterator

Examples

```
gdb.init_examples()
gvtrack.create("vtrack1", "rects_track")
gvtrack.iterator.2d("vtrack1", sshift1 = 1000, eshift1 = 2000)
gextract(
    "rects_track", "vtrack1",
    gintervals.2d(1, 0, 5000, 2, 0, 5000)
)
```

---

gvtrack.ls *Returns a list of virtual track names*

Description

Returns a list of virtual track names.

Usage

```
gvtrack.ls(
  pattern = "",
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  useBytes = FALSE
)
```

Arguments

pattern, ignore.case, perl, fixed, useBytes
see 'grep'

Details

This function returns a list of virtual tracks that exist in current R environment that match the pattern (see 'grep'). If called without any arguments all virtual tracks are returned.

Value

An array that contains the names of virtual tracks.

See Also

grep, gvtrack.create, gvtrack.rm

Examples

```
gdb.init_examples()
gvtrack.create("vtrack1", "dense_track", "max")
gvtrack.create("vtrack2", "dense_track", "quantile", 0.5)
gvtrack.ls()
gvtrack.ls(pattern = "*2")
```

---

gvtrack.rm *Deletes a virtual track*

Description

Deletes a virtual track.

Usage

```
gvtrack.rm(vtrack = NULL)
```

Arguments

vtrack virtual track name

Details

This function deletes a virtual track from current R environment.

Value

None.

See Also

gvtrack.create, gvtrack.ls

Examples

```
gdb.init_examples()
gvtrack.create("vtrack1", "dense_track", "max")
gvtrack.create("vtrack2", "dense_track", "quantile", 0.5)
gvtrack.ls()
gvtrack.rm("vtrack1")
gvtrack.ls()
```

---

gwget *Downloads files from FTP server*

Description

Downloads multiple files from FTP server

Usage

```
gwget(url = NULL, path = NULL)
```

Arguments

url	URL of FTP server
path	directory path where the downloaded files are stored

Details

This function downloads files from FTP server given by 'url'. The address in 'url' can contain wildcards to download more than one file at once. Files are downloaded to a directory given by 'path' argument. If 'path' is 'NULL', file are downloaded into 'GROOT/downloads'.

Value

An array of file names that have been downloaded.

See Also

gtrack.import_set

Examples

```
gdb.init_examples()

outdir <- tempdir()
gwget("ftp://hgdownload.soe.ucsc.edu/goldenPath/hg19/chromosomes/md5sum.txt", path = outdir)
```

---

gwilcox *Calculates Wilcoxon test on sliding windows over track expression*

Description

Calculates Wilcoxon test on sliding windows over the values of track expression.

Usage

```
gwilcox(
  expr = NULL,
  winsize1 = NULL,
  winsize2 = NULL,
  maxpval = 0.05,
  onetailed = TRUE,
  what2find = 1,
  intervals = NULL,
  iterator = NULL,
  intervals.set.out = NULL
)
```

Arguments

expr	track expression
winsize1	number of values in the first sliding window
winsize2	number of values in the second sliding window
maxpval	maximal P-value
onetailed	if 'TRUE', Wilcoxon test is performed one tailed, otherwise two tailed
what2find	if '-1', lows are searched. If '1', peaks are searched. If '0', both peaks and lows are searched
intervals	genomic scope for which the function is applied
iterator	track expression iterator of "fixed bin" type. If 'NULL' iterator is determined implicitly based on track expression.
intervals.set.out	intervals set name where the function result is optionally outputted

Details

This function runs a Wilcoxon test (also known as a Mann-Whitney test) over the values of track expression in the two sliding windows having an identical center. The sizes of the windows are specified by 'winsize1' and 'winsize2'. 'gwilcox' returns intervals where the smaller window tested against a larger window gives a P-value below 'maxpval'. The test can be one or two tailed.

'what2find' argument controls what should be searched: peaks, lows or both.

If 'intervals.set.out' is not 'NULL' the result is saved as an intervals set. Use this parameter if the result size exceeds the limits of the physical memory.

Value

If 'intervals.set.out' is 'NULL' a data frame representing the intervals with an additional 'pval' column where P-value is below 'maxpval'.

See Also

gscreen, gsegment

Examples

```
gdb.init_examples()
gwilcox("dense_track", 100000, 1000,
    maxpval = 0.01,
    what2find = 1
)
```

---

print.gsynth.model *Print summary of a gsynth.model*

Description

Print summary of a gsynth.model

Usage

```
## S3 method for class 'gsynth.model'
print(x, ...)
```

Arguments

x	A gsynth.model object
...	Additional arguments (ignored)

Index

* **~2d**
  gintervals.2d.intersect, 59
  gintervals.2d.union, 60
* **~ALLGENOME**
  gintervals.2d.all, 56
  gintervals.all, 61
* **~BED**
  gintervals.import_bed, 81
* **~DNA**
  gseq.extract, 127
* **~GFF**
  gintervals.import_gff, 83
* **~GTF**
  gintervals.import_gff, 83
* **~Mann-Whitney**
  gsegment, 124
  gwilcox, 226
* **~TSS**
  gintervals.neighbors, 96
* **~VCF**
  gintervals.import_vcf, 84
* **~annotate**
  gintervals.neighbors, 96
* **~apply**
  gintervals.mapply, 93
* **~array**
  gtrack.array.extract, 163
  gtrack.array.get_colnames, 164
  gtrack.array.import, 165
  gtrack.array.set_colnames, 166
  gvtrack.array.slice, 205
* **~attribute**
  gdb.get_readonly_attrs, 32
  gdb.set_readonly_attrs, 41
  gintervals.attr.export, 65
  gintervals.attr.get, 66
  gintervals.attr.import, 67
  gintervals.attr.set, 68
  gtrack.attr.export, 167
  gtrack.attr.get, 168
  gtrack.attr.import, 169
  gtrack.attr.set, 170
* **~attr**
  gdb.get_readonly_attrs, 32
  gdb.set_readonly_attrs, 41
  gintervals.attr.export, 65
  gintervals.attr.get, 66
  gintervals.attr.import, 67
  gintervals.attr.set, 68
  gtrack.attr.export, 167
  gtrack.attr.get, 168
  gtrack.attr.import, 169
  gtrack.attr.set, 170
* **~auto-correlation**
  gcompute_strands_autocorr, 13
* **~autocorrelation**
  gcompute_strands_autocorr, 13
* **~band**
  gintervals.2d.band_intersect, 57
* **~bedgraph**
  gtrack.export_bedgraph, 182
  gtrack.import, 185
  gtrack.import_set, 189
* **~bigwig**
  gtrack.export_bigwig, 184
  gtrack.import, 185
  gtrack.import_set, 189
* **~canonic**
  gintervals.canonic, 69
* **~cartesian**
  giterator.cartesian_grid, 113
* **~cd**
  gdir.cd, 43
* **~chain**
  gintervals.as_chain, 64
  gintervals.liftover, 86
  gintervals.load_chain, 90
  gtrack.liftover, 191

* **~chromosomes**
  gintervals.2d.all, 56
  gintervals.all, 61
* **~chromosome**
  gintervals.2d.all, 56
  gintervals.all, 61
* **~cluster**
  gcluster.run, 12
* **~columns**
  gtrack.array.get_colnames, 164
  gtrack.array.set_colnames, 166
* **~contacts**
  gcis_decay, 10
  gtrack.2d.import_contacts, 161
* **~convert**
  gtrack.convert, 171
* **~correlation**
  gcompute_strands_autocorr, 13
* **~coverage**
  gintervals.coverage_fraction, 73
  gintervals.covered_bp, 74
* **~create**
  gdb.create, 26
  gdb.create_linked, 29
  gdir.create, 44
  gtrack.2d.create, 159
  gtrack.array.import, 165
  gtrack.create, 174
  gtrack.create_dense, 175
  gtrack.create_sparse, 178
* **~cwd**
  gdir.cwd, 44
* **~database**
  gdb.create, 26
  gdb.create_linked, 29
  gdb.init, 33
  gdb.unload, 42
  gdir.cd, 43
  gdir.create, 44
  gdir.cwd, 44
  gdir.rm, 45
  gintervals.dataset, 75
  gintervals.dbs, 76
  gtrack.dataset, 180
  gtrack.dbs, 181
* **~data**
  gdb.init, 33
  gdir.cd, 43
  gdir.create, 44
  gdir.cwd, 44
  gdir.rm, 45
* **~db**
  gdb.create_linked, 29
  gdb.init, 33
  gdb.reload, 41
  gdir.cd, 43
  gdir.create, 44
  gdir.cwd, 44
  gdir.rm, 45
* **~dense**
  gtrack.create_dense, 175
* **~diff**
  gintervals.diff, 77
* **~directory**
  gdir.cd, 43
  gdir.create, 44
  gdir.cwd, 44
  gdir.rm, 45
* **~dir**
  gdir.cd, 43
  gdir.create, 44
  gdir.cwd, 44
  gdir.rm, 45
* **~distribution**
  gdist, 46
* **~energy**
  gtrack.create_pwm_energy, 177
* **~export**
  gtrack.export_bedgraph, 182
  gtrack.export_bigwig, 184
* **~extract**
  gextract, 48
  glookup, 117
  gseq.extract, 127
  gtrack.array.extract, 163
* **~filter**
  gvtrack.filter, 218
* **~folder**
  gdir.cd, 43
  gdir.create, 44
  gdir.cwd, 44
  gdir.rm, 45
* **~fragment**
  gtrack.2d.import_contacts, 161
* **~ftp**
  gwget, 225

* **~gcompute_strands_autocorr**
  gcompute_strands_autocorr, 13
* **~genes**
  gdb.create, 26
  gintervals.import_genes, 82
* **~genome**
  gintervals.2d.all, 56
  gintervals.all, 61
* **~genomics**
  gintervals.coverage_fraction, 73
  gintervals.covered_bp, 74
* **~import**
  gintervals.import_bed, 81
  gintervals.import_genes, 82
  gintervals.import_gff, 83
  gintervals.import_vcf, 84
  gtrack.array.import, 165
* **~info**
  gtrack.info, 190
* **~intersect**
  gintervals.2d.band_intersect, 57
  gintervals.2d.intersect, 59
  gintervals.intersect, 85
* **~intervals**
  gintervals, 54
  gintervals.2d, 55
  gintervals.as_chain, 64
  gintervals.attr.export, 65
  gintervals.attr.get, 66
  gintervals.attr.import, 67
  gintervals.attr.set, 68
  gintervals.canonic, 69
  gintervals.chrom_sizes, 71
  gintervals.dataset, 75
  gintervals.dbs, 76
  gintervals.exists, 78
  gintervals.force_range, 78
  gintervals.from_strings, 80
  gintervals.import_bed, 81
  gintervals.import_genes, 82
  gintervals.import_gff, 83
  gintervals.import_vcf, 84
  gintervals.is.bigset, 86
  gintervals.liftover, 86
  gintervals.load, 89
  gintervals.load_chain, 90
  gintervals.ls, 92
  gintervals.neighbors, 96
  gintervals.path, 102
  gintervals.rm, 107
  gintervals.save, 108
  gintervals.update, 112
  giterator.intervals, 115
  gscreen, 123
  gtrack.ls, 195
* **~interval**
  gscreen, 123
* **~iterator**
  giterator.cartesian_grid, 113
  giterator.intervals, 115
* **~liftover**
  gintervals.as_chain, 64
  gintervals.liftover, 86
  gintervals.load_chain, 90
  gtrack.liftover, 191
* **~lookup**
  glookup, 117
  gtrack.lookup, 193
* **~ls**
  gintervals.ls, 92
  gtrack.ls, 195
  gtrack.var.ls, 202
  gvtrack.ls, 223
* **~mapped**
  gtrack.import_mappedseq, 187
* **~mapply**
  gintervals.mapply, 93
* **~modify**
  gtrack.modify, 196
* **~nearest**
  gintervals.neighbors, 96
* **~neighbors**
  gintervals.neighbors, 96
* **~neighbor**
  gintervals.neighbors, 96
* **~partition**
  gpartition, 119
* **~path**
  gintervals.dataset, 75
  gintervals.dbs, 76
  gintervals.path, 102
  gtrack.dataset, 180
  gtrack.dbs, 181
  gtrack.path, 198
* **~percentiles**
  gbins.quantiles, 7

gintervals.quantiles, 103
gquantiles, 120
* **~property**
gtrack.info, 190
* **~pssm**
gtrack.create_pwm_energy, 177
* **~pwd**
gdir.cwd, 44
* **~pwm**
gtrack.create_pwm_energy, 177
* **~quantiles**
gbins.quantiles, 7
gintervals.quantiles, 103
gquantiles, 120
* **~rbind**
gintervals.rbind, 106
* **~rm**
gdir.rm, 45
* **~sample**
gsample, 122
* **~screen**
gscreen, 123
* **~segment**
gsegment, 124
* **~sequence**
gseq.extract, 127
gtrack.import_mappedseq, 187
* **~smooth**
gtrack.smooth, 200
* **~sparse**
gtrack.create_sparse, 178
* **~statistics**
gintervals.summary, 109
gsummary, 140
* **~strand**
gintervals.neighbors, 96
* **~summary**
gbins.summary, 9
gintervals.summary, 109
gsummary, 140
* **~track**
gtrack.2d.create, 159
gtrack.2d.import, 160
gtrack.2d.import_contacts, 161
gtrack.array.import, 165
gtrack.convert, 171
gtrack.copy, 173
gtrack.create, 174
gtrack.create_dense, 175
gtrack.create_pwm_energy, 177
gtrack.create_sparse, 178
gtrack.dataset, 180
gtrack.dbs, 181
gtrack.exists, 182
gtrack.export_bedgraph, 182
gtrack.export_bigwig, 184
gtrack.import, 185
gtrack.import_mappedseq, 187
gtrack.import_set, 189
gtrack.info, 190
gtrack.liftover, 191
gtrack.lookup, 193
gtrack.modify, 196
gtrack.mv, 197
gtrack.path, 198
gtrack.rm, 199
gtrack.smooth, 200
* **~union**
gintervals.2d.union, 60
gintervals.union, 111
* **~variable**
gtrack.var.get, 201
gtrack.var.ls, 202
gtrack.var.rm, 203
gtrack.var.set, 204
* **~virtual**
gvtrack.array.slice, 205
gvtrack.clear, 206
gvtrack.create, 207
gvtrack.filter, 218
gvtrack.info, 220
gvtrack.iterator, 221
gvtrack.iterator.2d, 222
gvtrack.ls, 223
gvtrack.rm, 224
* **~wig**
gtrack.import, 185
gtrack.import_set, 189
* **~wilcoxon**
gsegment, 124
gwilcox, 226
* **motif functions**
gseq.read_homer, 136
gseq.read_jaspar, 137
gseq.read_meme, 138
* **package**

misha-package, 6

dir.create, *44*

gbins.quantiles, 7, *104*, *121*
gbins.summary, 9, *109*, *141*
gcis_decay, 10
gcluster.run, 12
gcompute_strands_autocorr, 13
gcor, 15
gdataset.example_path, 16
gdataset.info, 17, *19, 20*
gdataset.load, *17*, 18, *19–21*, *29*
gdataset.ls, *17, 18*, 19, *21*, *29*, *75, 76*, *180, 181*
gdataset.save, *17, 18*, 19
gdataset.unload, *18*, 20
gdb.build_genome, 21, *31*, *35*, *38–40*
gdb.convert_to_indexed, 23, *158*
gdb.create, *21*, *23*, *25*, 26, *29*, *34*, *41*, *51*, *83*
gdb.create_genome, 28
gdb.create_linked, 29
gdb.export_fasta, 30, *51*, *53*
gdb.genome_info, *23*, 31, *40*
gdb.get_readonly_attrs, 32, *42*, *170, 171*
gdb.info, 32
gdb.init, *25*, *27*, *30*, 33, *35*, *41–45*
gdb.init.examples *(*gdb.init_examples*)*, 34
gdb.init_examples, *16, 17*, 34
gdb.install_gff3_converter, 35, *36*
gdb.install_gtf_converter, 36, *39*
gdb.install_intervals, *21–23*, 37
gdb.list_genomes, *23*, *31*, 39
gdb.mark_cache_dirty, 40
gdb.reload, *27*, *34*, *40*, 41, *42*
gdb.set_readonly_attrs, *32*, 41
gdb.unload, 42
gdir.cd, *34*, *41*, 43, *45*, *159*, *174*, *179*, *186*
gdir.create, *43*, 44, *45*, *160, 161*, *163*, *166*, *175*, *178, 179*, *186*, *189, 190*, *194*, *201*
gdir.cwd, *43*, 44, *44, 45*
gdir.rm, *43, 44*, 45, *45*
gdist, *8–11*, 46, *47*, *49, 50*, *118*, *120, 121*, *124*, *126*, *141*
gextract, *16*, *47*, 48, *49*, *118*, *120, 121*, *123–125*, *128*, *141*, *164*, *166*, *183, 184*, *186*, *190*
ggenome.implant, 50, *53*
ggenome.transplant, *51*, 52
gintervals, 54, *56*, *61*, *69*, *71*, *73, 74*, *77–79*, *81*, *83*, *85*, *88*, *90*, *92*, *97*, *106–108*, *112*, *179*
gintervals.2d, *54*, 55, *57*, *59, 60*, *69*, *71*, *78, 79*, *81*, *85*, *90*, *92*, *106–108*, *112*
gintervals.2d.all, 56
gintervals.2d.band_intersect, 57, *85*
gintervals.2d.convert_to_indexed, *25*, 58
gintervals.2d.intersect, 59, *60*
gintervals.2d.union, 60, *60*
gintervals.all, 61, *73*
gintervals.annotate, 61
gintervals.as_chain, 64
gintervals.attr.export, 65, *66–68*
gintervals.attr.get, 66, *66–68*
gintervals.attr.import, *66*, 67, *68*
gintervals.attr.set, *66, 67*, 68
gintervals.canonic, 69, *74*, *79*, *106*
gintervals.chrom_sizes, 71
gintervals.convert_to_indexed, *25*, 72
gintervals.coverage_fraction, 73, *74*
gintervals.covered_bp, *73*, 74
gintervals.dataset, 75, *76*, *92*
gintervals.dbs, *75*, 76
gintervals.diff, 77, *85*, *112*
gintervals.exists, *71*, *75*, 78, *86*, *90*, *92*, *103*, *107, 108*, *113*
gintervals.force_range, *54*, *56*, 78, *102*
gintervals.from_mat, 79, *110*
gintervals.from_strings, 80
gintervals.import_bed, 81, *84*
gintervals.import_genes, *27*, 82
gintervals.import_gff, *82*, 83, *84*
gintervals.import_vcf, *82*, 84, *84*
gintervals.intersect, *57*, *59, 60*, *73*, *77*, 85, *112*
gintervals.is.bigset, 86, *90*
gintervals.liftover, *64*, 86, *91*, *193*
gintervals.load, *71, 72*, *78*, *86*, 89, *92*, *108*, *113*
gintervals.load_chain, *64*, *88*, 90, *193*
gintervals.ls, *34*, *71*, *75, 76*, *78*, *86*, *90*, 92, *103*, *107, 108*, *113*
gintervals.mapply, 93
gintervals.mark_overlaps, 95

gintervals.neighbors, 96, *100*
gintervals.neighbors.directional *(*gintervals.neighbors.upstream*)*, 99
gintervals.neighbors.downstream, *97*
gintervals.neighbors.downstream *(*gintervals.neighbors.upstream*)*, 99
gintervals.neighbors.upstream, *97*, 99
gintervals.normalize, 101
gintervals.path, 102, *198*
gintervals.quantiles, *8*, 103, *121*
gintervals.random, 104
gintervals.rbind, 106
gintervals.rm, *78*, *92*, 107, *108*
gintervals.save, *71*, *72*, *78*, *86*, *90*, *92*, *107*, 108, *113*
gintervals.summary, *10*, 109, *141*
gintervals.to_mat, *79*, *80*, 110
gintervals.union, *60*, *77*, *85*, 111
gintervals.update, 112
giterator.cartesian_grid, 113, *116*
giterator.intervals, *114*, 115
glookup, *50*, 117, *120*, *194*
gpartition, *50*, *118*, 119
gquantiles, *8*, *47*, *49*, *104*, 120, *121*, *124*, *126*, *141*
grep, *92*, *196*, *203*, *224*
grevcomp, 121, *140*
gsample, *50*, 122
gscreen, *16*, *47*, *49*, *120*, *121*, 123, *124*, *126*, *141*, *227*
gsegment, *47*, *49*, *121*, 124, *124*, *126*, *141*, *227*
gseq.comp, 126, *139*, *140*
gseq.extract, *30*, *51*, *53*, 127, *130*
gseq.kmer, 128, *130*
gseq.kmer.dist, 130
gseq.pwm, 131, *136*, *138*
gseq.pwm_edits, 134
gseq.read_homer, 136
gseq.read_homer(), *137*, *138*
gseq.read_jaspar, 137
gseq.read_jaspar(), *137*, *138*
gseq.read_meme, 138
gseq.read_meme(), *137*
gseq.rev, *126*, 139, *140*
gseq.revcomp, *126*, *139*, 140
gsetroot, *29*, *40*, *42*, *159*, *174*, *179*, *186*
gsetroot *(*gdb.init*)*, 33
gsummary, *10*, *16*, *47*, *49*, *109*, *121*, *124*, *125*, 140, *141*
gsynth.bin_map, 142, *143*, *144*, *157*
gsynth.cell_merge, *142*, 143, *151*
gsynth.convert, 144, *146*, *153*
gsynth.forbid_kmer, 145
gsynth.load, *144*, 146, *153*, *157*
gsynth.random, 147
gsynth.replace_kmer, 149
gsynth.sample, *142–145*, *147*, *148*, 150, *155*, *157*
gsynth.save, *144*, *146*, *152*, 153, *157*
gsynth.score, 154
gsynth.train, *142*, *143*, *145*, *146*, *148*, *150–154*, 155, *155*
gtrack.2d.convert_to_indexed, 158
gtrack.2d.create, *158*, 159, *172*, *175*, *178*, *179*, *194*, *199*, *201*
gtrack.2d.import, *158*, 160, *163*
gtrack.2d.import_contacts, *11*, *158*, 161
gtrack.array.extract, *50*, 163, *165–167*, *206*
gtrack.array.get_colnames, 164, *164*, *167*, *206*
gtrack.array.import, *50*, *164*, 165
gtrack.array.set_colnames, *165*, 166, *166*
gtrack.attr.export, 167
gtrack.attr.get, *32*, *42*, *160*, 168, *168*, *170*, *171*, *195*
gtrack.attr.import, *168*, 169, *169–171*
gtrack.attr.set, *32*, *42*, *168*, *169*, 170, *170*
gtrack.convert, 171
gtrack.convert_to_indexed, *25*, *158*, 172
gtrack.copy, 173, *197*, *198*
gtrack.create, *160*, *172*, *173*, 174, *178*, *179*, *182*, *196*, *197*, *199*, *201*
gtrack.create_dense, *173*, 175
gtrack.create_dirs, 177
gtrack.create_pwm_energy, 177
gtrack.create_sparse, *160*, *172*, *173*, *175*, *176*, 178, *178*, *194*, *199*, *201*
gtrack.dataset, 180, *181*, *196*
gtrack.dbs, *180*, 181
gtrack.exists, *180*, *181*, 182, *191*, *196*, *198*, *199*
gtrack.export_bedgraph, 182, *184*
gtrack.export_bigwig, *183*, 184

gtrack.import, *50*, *176*, 185, *190*
gtrack.import_mappedseq, 187
gtrack.import_set, *186*, 189, *225*
gtrack.info, *160, 161*, *163*, *165–167*, *175*, *176*, *178, 179*, *182–184*, *186*, *189*, 190, *190*, *194*, *201*
gtrack.liftover, *88*, *91*, 191
gtrack.lookup, *118*, 193
gtrack.ls, *34*, *180–182*, *191*, 195, *198, 199*
gtrack.modify, *160*, *175, 176*, *178, 179*, *194*, 196, *201*
gtrack.mv, *174*, 197
gtrack.path, *103*, 198
gtrack.rm, *107*, *160, 161*, *163*, *166*, *174–176*, *178, 179*, *182*, *186*, *189, 190*, *194*, *196–198*, 199, *201*
gtrack.smooth, *160*, *175*, *178, 179*, *194*, *199*, 200
gtrack.var.get, 201, *203, 204*
gtrack.var.ls, 202, *202–204*
gtrack.var.rm, *202*, 203, *203, 204*
gtrack.var.set, *171*, *202, 203*, 204
gvtrack.array.slice, *165*, *167*, 205, *213*
gvtrack.clear, 206
gvtrack.create, *129*, *132, 133*, *136*, *206*, 207, *219*, *221–225*
gvtrack.filter, *213*, 218
gvtrack.info, *213*, *219*, 220
gvtrack.iterator, *213*, *219*, 221, *223*
gvtrack.iterator.2d, *213*, 222, *222*
gvtrack.ls, *34*, *206*, *213*, 223, *225*
gvtrack.rm, *206*, *213*, 224, *224*
gwget, *190*, 225
gwilcox, *126*, 226

mapply, *94*
misha *(*misha-package*)*, 6
misha-package, 6

print.gsynth.model, 227

tempdir, *35*