xlx Package

May 19, 2017

Type Package

Title XLX

Version 1.5

Date 2013-12-12

Author Alain Viari

Maintainer `<alain.viari@inria.fr>`

Description eXtensions to LX library.

License GPL

Depends methods,
lx,
rbgzf

Imports bit,
digest,
intervals

Suggests KFAS

RoxygenNote 5.0.1

R topics documented:

`apply.cloc` *apply function to clocations by chromosomes*

Description

this is a variant of apply.clocs where `fun` is called on each clocation (instead of matrix of clocations). it can be viewed as a simple `apply(clocations, 1, fun, ...)` except that the job is actually split by chromosome for multithreading.

Usage

```
apply.cloc(clocations, fun, ..., handle = NULL, keep.order = TRUE,
  use.threads = lx.use.threads(), mc.cores = lx.options(mc.cores))
```

Arguments

`clocations`	nx3 matrix of clocations
`fun`	: function or function name called as `fun(cloc, handle, ...)`
`...`	anything passed to `fun`
`handle`	optional (basta or baf) file handle. if use.threads==TRUE then handle will be properly duplicated thru calls (as with lx.happly)
`keep.order`	keep `fun` results in the same order as clocations
`use.threads`	(see lx.use.threads)
`mc.cores`	number of processes (see HELP.LX.OPTIONS)

Details

`fun` first argument is a single clocation **not** a matrix of clocations as in apply.clocs

Value

a (unnamed) vector of results of `fun`

Note

if result order is not important, then use keep.order=FALSE (this will slightly speedup the operation and save memory)

See Also

apply.clocs

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
slocs <- c("seq1:1-10", "seq2:2-3", "seq1:15-20")
clocs <- clocations(lapply(slocs, sloc2cloc, handle=fh))
seqs <- apply.cloc(clocs, function(cloc, handle) {
                       basta.fetch.cloc(handle, cloc)
                     }, handle=fh, use.threads=TRUE, mc.cores=2)
seqs <- apply.cloc(clocs, function(cloc, handle, foo) {
                paste0(foo, basta.fetch.cloc(handle, cloc))
}, handle=fh, foo="seq:", use.threads=TRUE, mc.cores=2)

basta.close(fh)
```

`apply.clocs` *apply function to clocations by chromosomes*

Description

split a nx3 matrix of clocations by chromosomes (first column), and apply user's function to each submatrix in turn.
this is a pivotal function of the XLX library to split job across chromosomes for multithreading or disk pooling

Usage

```
apply.clocs(clocations, fun, ..., handle = NULL, flatten = FALSE,
  use.threads = lx.use.threads(), mc.cores = lx.options(mc.cores))
```

Arguments

`clocations`	nx3 matrix of clocations
`fun`	: function or function name called as `fun(clocs, handle, ...)` (see details)
`...`	anything passed to `fun`
`handle`	optional (basta or baf) file handle. if use.threads==TRUE then handle will be properly duplicated thru calls (as with lx.happly)
`flatten`	flatten results and reorder them in the same order as clocations (see details)
`use.threads`	(see lx.use.threads)
`mc.cores`	number of processes (see HELP.LX.OPTIONS)

Details

`fun` first argument is a matrix of clocations (on a single chromosome) **not** a single clocation. see apply.cloc for this variant.
the `flatten` parameter is only meaningful if the results of `fun(clocs, handle,...)` is a list of length exactly equals to nrow(clocs). then all the results will be catenated and reordered in the same order as in clocations.
be careful with NULL (or empty) elements in results that may be swallowed.
if at least one call to `fun` does not meet this criterion, then a warning is raised and results will not be flattened at all. (the apply.cloc version will take care of this).

Value

a named list of results of `fun` (names are `as.character(chrindex)`) or a flattened (unnamed) list if flatten==TRUE (see Details).

See Also

apply.cloc

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
slocs <- c("seq1:1-10", "seq2:2-3", "seq1:15-20")
clocs <- clocations(lapply(slocs, sloc2cloc, handle=fh))
seqs <- apply.clocs(clocs, function(sublocs, handle) {
                       apply(sublocs, 1, basta.fetch.cloc, handle=handle)
                     }, handle=fh, use.threads=TRUE, mc.cores=2)
seqs <- apply.clocs(clocs, function(sublocs, handle) {
                       apply(sublocs, 1, basta.fetch.cloc, handle=handle)
                     }, handle=fh, flatten=TRUE, use.threads=TRUE, mc.cores=2)
basta.close(fh)
```

`as.character` *Coerce Dna to character string*

Description

Coerce Dna sequence to character string

Usage

```
## S3 method for class 'Dna'
as.character(x, ...)
```

Arguments

`x`	Dna object to be coerced to character
`...`	further arguments passed to or from other methods.

Examples

```
x <- Dna("acgtnacgtn")
as.character(x)
```

`as.Dna` *Generic method to coerce to Dna*

Description

coerce character string or Dna object to Dna object
this is basically the same as the Dna constructor.
for Dna object this allows to force a re-encoding

Usage

```
as.Dna(obj, code, pattern)
```

Arguments

`obj`	object to coerce to Dna
`code`	see Dna
`pattern`	see Dna

Value

Dna object

See Also

Dna, as.Dna.Dna, as.Dna.character

`as.Dna.character` *Coerce character string to Dna*

Description

see as.Dna
this is basically the same as the Dna constructor.

Usage

```
## S3 method for class 'character'
as.Dna(obj, ...)
```

Arguments

`obj`	character string to coerce to Dna
`...`	any argument to Dna

Examples

```
as.Dna("acgtacgt", code='strict')
```

as.Dna.Dna *Coerce Dna to Dna*

Description

see as.Dna
this is basically used to force Dna reencoding.

Usage

```
## S3 method for class 'Dna'
as.Dna(obj, code = obj$code, pattern = obj$pattern)
```

Arguments

`obj`	object to coerce to Dna
`code`	see Dna
`pattern`	see Dna

Examples

```
x <- Dna("acgtacgt")
as.Dna(x, code='strict')
```

as.ri *Generic method to coerce to range*

Description

coerce to range-index (ri) object from library bit

Usage

```
as.ri(obj)
```

Arguments

`obj`	object to coerce to ri

See Also

ri in library bit, as.ri.Dna

`as.ri.Dna` — *Coerce Dna to ri range [1, length(dna)]*

Description

Coerce Dna to ri range [1, length(dna)]

Usage

```
## S3 method for class 'Dna'
as.ri(obj)
```

Arguments

`obj`	object to coerce to ri

See Also

as.ri, ri

`as.ri.ri` — *Coerce bit::ri to bit::ri*

Description

just a trivial helper (absent from bit)

Usage

```
## S3 method for class 'ri'
as.ri(obj)
```

Arguments

`obj`	object to coerce to ri

baf.bin.cloc *binning coverage or GC content using relative coordinates*

Description

coord defines a region on a chromosome. this function collects coverage or GC content by bins of width binsize within the region.

Usage

```
baf.bin.cloc(handle, clocation, binsize = 10000L, what = c("coverage",
  "gc"), fun = sum, drop = TRUE, na.gc = FALSE, ...,
  .quick = any(sapply(c(sum, mean), identical, fun)) && (!na.gc))
```

Arguments

handle	file handle (as returned by baf.open)
clocation	relative clocation = c(seqname, from, to) (1-based)
binsize	size of bins
what	what to collect "coverage" or "gc" (may be abrreviated)
fun	collect function (e.g. sum, mean, median, user-closure, ...) see details.
drop	drop the last element of result if region width is not a muliple of binsize
na.gc	boolean to specify how to handle GC content for positions with 0 coverage. na.gc = TRUE will produce 0/0 = NA and na.gc = FALSE will produce 0/0 = 0.
...	optional arguments to be passed to fun
.quick	use a quicker algorithm (valid for fun=sum or fun=mean only and na.gc=FALSE) at the expense of memory overhead.

Details

let us note allele.counts the binsize x 4 matrix of alleles counts in each bin.
if (what=="cover") then fun(coverage) is collected in each bin,
with coverage = rowSums(allele.counts)

if (what=="gc") then fun(gc.line) is collected in each bin,
with gc.line = rowSums(GC.allele.counts) / rowSums(allele.counts). (with a special treatment of NA's. see below)
Therefore fun=sum will produce the number of GC alleles in bin and fun=mean will produce the %GC. Note that functions other than sum or mean are usually meaningless with what=="gc"

fun can be any function or user-supplied closure taking a numerical vector as input and returning a scalar.

the na.gc parameter is intended to handle the special case where coverage=0 at a position.
Then the computed gc.line at this position is NA if na.gc=TRUE, and 0 if na.gc=FALSE.
Please note that na.gc=TRUE will disable quick mode.

the drop parameter handles the last bin when region width is not a muliple of binsize./cr if drop=TRUE then the last (incomplete) bin is omited. if drop=FALSE then the last (incomplete) bin is included.

Value

numeric vector of size n containing binned cover or gc content

Note

see HELP.COORD for help on coordinates systems

See Also

baf.fetch.cloc baf.bin.coord

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
i <- baf.name2index(baf, "machaon")
baf.fetch.cloc(baf, c(i, 560, 565))
# 'fun' usage
baf.bin.coord(baf, c(33725, 33730), 3) # sum
baf.bin.cloc(baf, c(i, 560, 565), 3, fun=mean)
baf.bin.cloc(baf, c(i, 560, 565), 3, fun=median)
# 'na.gc' usage
baf.bin.cloc(baf, c(i, 560, 565), 3, what="gc")
baf.bin.cloc(baf, c(i, 560, 565), 3, what="gc", na.gc=TRUE)
baf.bin.cloc(baf, c(i, 560, 565), 3, what="gc", na.gc=TRUE, na.rm=TRUE)
# 'drop' usage
baf.bin.cloc(baf, c(i, 560, 565), 4)
baf.bin.cloc(baf, c(i, 560, 565), 4, drop=FALSE)
baf.bin.cloc(baf, c(i, 560, 565), 10)
baf.bin.cloc(baf, c(i, 560, 565), 10, drop=FALSE)
baf.close(baf)
```

`baf.bin.coord` *binning coverage or GC content using absolute coordinates*

Description

`coord` defines a region on a chromosome. this function collects coverage or GC content by bins of width `binsize` within the region.

Usage

```
baf.bin.coord(handle, coord, binsize = 10000L, what = c("coverage", "gc"),
  fun = sum, drop = TRUE, na.gc = FALSE, ..., .quick = any(sapply(c(sum,
  mean), identical, fun)) && (!na.gc))
```

Arguments

`handle`	file handle (as returned by baf.open)
`coord`	absolute sequence coordinates (c(absfrom, absto)) (1-based) or a single position absfrom (this implies absto=absfrom)
`binsize`	size of bins
`what`	what to collect "coverage" or "gc" (may be abrreviated)

`fun`	collect function (e.g. sum, mean, median, user-closure, ...) see details.
`drop`	drop the last element of result if region width is not a muliple of binsize
`na.gc`	boolean to specify how to handle GC content for positions with 0 coverage. na.gc = TRUE will produce 0/0 = NA and na.gc = FALSE will produce 0/0 = 0.
`...`	optional arguments to be passed to `fun`
`.quick`	use a quicker algorithm (valid for fun=sum or fun=mean only and na.gc=FALSE) at the expense of memory overhead.

Details

let us note allele.counts the binsize x 4 matrix of alleles counts in each bin.
if (what=="cover") then fun(coverage) is collected in each bin,
with coverage = rowSums(allele.counts)

if (what=="gc") then fun(gc.line) is collected in each bin,
with gc.line = rowSums(GC.allele.counts) / rowSums(allele.counts). (with a special treatment of NA's. see below)
Therefore fun=sum will produce the number of GC alleles in bin and fun=mean will produce the %GC. Note that functions other than sum or mean are usually meaningless with what=="gc"

`fun` can be any function or user-supplied closure taking a numerical vector as input and returning a scalar.

the `na.gc` parameter is intended to handle the special case where coverage=0 at a position.
Then the computed gc.line at this position is NA if na.gc=TRUE, and 0 if na.gc=FALSE.
Please note that na.gc=TRUE will disable quick mode.

the `drop` parameter handles the last bin when region width is not a muliple of binsize./cr if drop=TRUE then the last (incomplete) bin is omited. if drop=FALSE then the last (incomplete) bin is included.

Value

numeric vector of size n containing binned cover or gc content

Note

see HELP.COORD for help on coordinates systems

See Also

baf.fetch.coord baf.bin.cloc

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
baf.fetch.coord(baf, c(33725, 33730))
# 'fun' usage
baf.bin.coord(baf, c(33725, 33730), 3) # sum
baf.bin.coord(baf, c(33725, 33730), 3, fun=mean)
baf.bin.coord(baf, c(33725, 33730), 3, fun=median)
# 'na.gc' usage
baf.bin.coord(baf, c(33725, 33730), 3, what="gc")
baf.bin.coord(baf, c(33725, 33730), 3, what="gc", na.gc=TRUE)
baf.bin.coord(baf, c(33725, 33730), 3, what="gc", na.gc=TRUE, na.rm=TRUE)
# 'drop' usage
baf.bin.coord(baf, c(33725, 33730), 4)
baf.bin.coord(baf, c(33725, 33730), 4, drop=FALSE)
baf.bin.coord(baf, c(33725, 33730), 10)
baf.bin.coord(baf, c(33725, 33730), 10, drop=FALSE)
baf.close(baf)
```

`baf.close` *close baf file*

Description

same as lx.close

Usage

```
baf.close(handle)
```

Arguments

`handle`	file handle (opened by baf.open)

`baf.count.filter` *filter allele counts*

Description

filter allele counts according to coverage, min and max number of alleles and max allele frequency (see details).

Usage

```
baf.count.filter(count, lowread = 2L, mincov = 10L, minall = 1L,
  maxall = 2L, deltafreq = 0.1, what = c("count", "index", "logical"))
```

Arguments

`count`	nx4 integer matrix of alleles counts. as returned by `baf.fetch.xxx`, `baf.sample.xxx` or `baf.heterozygous.xxx`.
`lowread`	low read threshold (see details)
`mincov`	minimal coverage (see details)
`minall`	minimal number of alleles (see details)
`maxall`	maximal number of alleles (see details)
`deltafreq`	max allele frequency (see details)
`what`	kind of result to return (see value)

Details

definitions
an allele X is present iff:

- count_X > `lowread`

a site (i.e. a row of `count`) is valid iff:

- coverage >= `covmin`
- `minall` <= nb_alleles <= `maxall`
- if (nb_alleles > 1) abs(0.5 - count_max_allele / coverage) <= `deltafreq`

if `mincov` is < 0 then `mincov` is computed as:
`median(coverage) + mincov * mad(coverage)`

if `delta.freq` == NA or maxall < 2 then freq condition is ignored

Value

- if (what == "count") filtered count matrix
- if (what == "index") indexes of valid rows
- if (what == "logical") logical vector indicating valid rows

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
cnt <- baf.sample(baf, sample.size=Inf)
cnt.ok <- baf.count.filter(cnt, lowread=2, mincov=10, minall=1, maxall=4, deltafreq=NA)
baf.close(baf)
```

`baf.count.genotype` *get genotype from allele counts*

Description

retrieve genotypes from allele counts matrix

Usage

```
baf.count.genotype(count, lowread = 2L, what = c("symbol", "index",
  "string"), sorted = FALSE, use.threads = lx.use.threads())
```

Arguments

`count`	nx4 integer matrix of alleles counts. as returned by `baf.fetch.xxx`, `baf.sample.xxx` or `baf.heterozygous.xxx`.
`lowread`	low read threshold (see details)
`what`	kind of result to return (see value)
`sorted`	sort result by frequencies (see details)
`use.threads`	(see lx.use.threads)

Details

this function returns the list of allele(s) present at each row of the `count` matrix.
an allele X is present iff: `count_X > lowread`
the typeof result depends upon the `what` parameter:

- if (what == "symbol") list of character array of alleles symbols
- if (what == "index") list of integer array of alleles indices
- if (what == "string") array of genotype strings

if `sorted`==FALSE (default) each array element of the result list (or each character in string) appears in the same order as in `colnames(count)` whatever the frequency value. if `sorted`==TRUE then array elements are sorted by decreasing frequency.

Value

list of alleles or array of strings (see details)

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
cnt <- baf.sample(baf, sample.size=Inf)
# sample: count all genotypes
cnt.ok <- baf.count.filter(cnt, lowread=0, mincov=10, minall=1, maxall=4, deltafreq=NA)
geno <- baf.count.genotype(cnt.ok, lowread=0)
table(nb.all <- sapply(geno, length))
# sample: ordered bi-allelic genotypes
cnt.ok <- baf.count.filter(cnt, lowread=0, mincov=10, minall=2, maxall=2, deltafreq=NA)
geno <- baf.count.genotype(cnt.ok, lowread=0, sorted=TRUE)
baf.close(baf)
```

`baf.count.regularize` *allelic frequency regularization*

Description

regularize allelic frequency using various methods

Usage

```
baf.count.regularize(count, tot, method = c("poisson", "gaussian"), ...)
```

Arguments

`count`	nx4 integer matrix of alleles counts. as returned by `baf.fetch.xxx`, `baf.sample.xxx` or
`tot`	total count (i.e. depth)
`method`	method to use ("poisson" or "gaussian")
`...`	specific method parameters

Value

names list of 3 vectors (cnt=regularized counts, tot=regularized total, raf=regularized all. freq.)

Examples

```
N <- 10000
cnt <- round(rpois(N, 30))
tot <- cnt + round(rpois(N, 30))
unreg <- cnt / tot
reg <- baf.count.regularize(cnt, tot, "gaussian", .seed=0)
```

---

baf.count.regularize.gaussian
allelic frequency regularization

Description

regularize allelic frequency using gaussian pseudo-counts

Usage

```
baf.count.regularize.gaussian(count, tot, sd = 0.5, n = 10L, .seed = -1L)
```

Arguments

`count`	nx4 integer matrix of alleles counts. as returned by `baf.fetch.xxx`, `baf.sample.xxx` or
`tot`	total count (i.e. depth)
`sd`	gaussian standard deviation
`n`	number of draws per point
`.seed`	seed for random (do not seed if < 0)

Value

names list of 3 vectors (cnt=regularized counts, tot=regularized total, raf=regulized all. freq.)

Examples

```
N <- 10000
cnt <- round(rpois(N, 5))
tot <- cnt + round(rpois(N, 5))
unreg <- cnt / tot
reg <- baf.count.regularize(cnt, tot, "gaussian", .seed=0)
```

`baf.count.regularize.poisson`
allelic frequency regularization

Description

regularize allelic frequency using poisson pseudo-counts

Usage

```
baf.count.regularize.poisson(count, tot, n = 10L, .eps = 0, .seed = -1L)
```

Arguments

`count`	nx4 integer matrix of alleles counts. as returned by `baf.fetch.xxx`, `baf.sample.xxx` or `baf.heterozygous.xxx`.
`tot`	total count (i.e. depth)
`n`	number of draws per point
`.eps`	lambda correction
`.seed`	seed for random (do not seed if < 0)

Value

names list of 3 vectors (cnt=regularized counts, tot=regularized total, raf=regulized all. freq.)

Examples

```
N <- 10000
cnt <- round(rpois(N, 5))
tot <- cnt + round(rpois(N, 5))
unreg <- cnt / tot
reg <- baf.count.regularize(cnt, tot, "poisson", .seed=0)
```

`baf.fetch.cloc` *fetch allele counts*

Description

fetch allele counts using using relative coordinates.
`cloc` defines a region on a chromosome. this function returns a count matrix (with 4 columns) of the number of symbols at each position within the region.

Usage

```
baf.fetch.cloc(handle, clocation)
```

Arguments

`handle`	file handle (as returned by baf.open)
`clocation`	relative clocation = c(seqname, from, to) (1-based)

Value

integer matrix of size n x 4 containing allele counts.

Note

see HELP.COORD for help on coordinates systems

See Also

baf.fetch.coord, baf.bin.cloc, baf.fetch.points.chr, baf.heterozygous.cloc

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
baf.fetch.coord(baf, c(33725, 33732))
i <- baf.name2index(baf, "machaon")
baf.fetch.cloc(baf, c(i, 560, 567))
baf.close(baf)
```

`baf.fetch.coord` *fetch allele counts*

Description

fetch allele counts using absolute coordinates.
`coord` defines a region on a chromosome. this function returns a count matrix (with 4 columns) of the number of symbols at each position within the region.

Usage

```
baf.fetch.coord(handle, coord)
```

Arguments

`handle`	file handle (as returned by baf.open)
`coord`	absolute sequence coordinates (c(absfrom, absto)) (1-based) or a single position absfrom (this implies absto=absfrom)

Value

integer matrix of size n x 4 containing allele counts.

Note

see HELP.COORD for help on coordinates systems

See Also

baf.fetch.cloc, baf.bin.coord, baf.fetch.points.chr, baf.heterozygous.coord

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
baf.fetch.coord(baf, c(33725, 33732))
i <- baf.name2index(baf, "machaon")
baf.fetch.cloc(baf, c(i, 560, 567))
baf.close(baf)
```

`baf.fetch.points.chr` *fetch allele counts*

Description

fetch allele counts at several relative point locations on the same chromosome.
`relpts` is a set of relative **point** positions on the same chromosome. this function returns a count matrix (with 4 columns) of the number of symbols at each position. this formaly equivalent to:
`clocs <- lapply(relpts, function(x) c(chrindex, x, x))`
`do.call(rbind, lapply(clocs, baf.fetch.cloc, handle=handle))`
but is much quicker when relpts vector is large and values span most of the chromosome.
The idea is to load the allele counts by chunks of size `.chunk.size` instead of accessing each location individually (thus reducing disk access overhead).

Usage

```
baf.fetch.points.chr(handle, chr, relpts, .chunk.size = 1000000L)
```

Arguments

`handle`	file handle (as returned by baf.open)
`chr`	chromosome index (if integer) or chromosome name (if character)
`relpts`	vector of relative positions (1-based) on this chromosome
`.chunk.size`	<internal parameter> size of chunk. changing this parameter will only affect time or memory used, not result.

Value

integer matrix of size n x 4 containing allele counts.

See Also

baf.fetch.cloc, baf.fetch.coord

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
x <- baf.fetch.points.chr(baf, 3, 550:570)
y <- baf.fetch.points.chr(baf, 3, 550:570, .chunk.size=1)
identical(x, y)
baf.close(baf)
```

`baf.heterozygous` *get heterozygous positions on all chromosomes*

Description

get heterozygous positions on all chromosomes this function gather all heterozygous positions defined as valid by baf.count.filter:

- coverage >= `covmin`
- `minall` <= nb_alleles <= `maxall`
- if (nb_alleles > 1) abs(0.5 - count_max_allele / coverage) <= `deltafreq`

Usage

```
baf.heterozygous(handle, chrs = NULL, lowread = 2L, mincov = 10L,
  deltafreq = 0.1, flatten = TRUE, .chunk.size = NA,
  .sample.size = 1000000L, use.threads = lx.use.threads())
```

Arguments

`handle`	file handle (as returned by baf.open)
`chrs`	integer (chromosome indexes) or character (chromosome names) vector specifying which chromosomes to use. Use NULL to specify all chromosome declared in baf header.
`lowread`	low read threshold (see baf.count.filter)
`mincov`	minimal coverage (see baf.count.filter)
`deltafreq`	max allele frequency (see baf.count.filter)
`flatten`	if TRUE flatten all chromosomes within a single matrix else return a list of such matrices, one per chromosome
`.chunk.size`	chunk size for loading chromosomes. see baf.heterozygous.chr
`.sample.size`	sample size to determine mincov for the case where mincov < 0 (see note below).
`use.threads`	(see lx.use.threads)

Value

(list of) integer matrix of size n x 4 containing allele counts at heterozygous sites. if flatten is TRUE matrix rownames are of the form: chr.position (1-based) else position (1 based) only.

Note

if (mincov < 0) then mincov will be estimated (as `median(coverage) + mincov * mad(coverage)`) as in baf.count.filter. However the region for computing coverage will depends upon the `.chunk.size` parameter: if `.chunk.size == NA` then this will be performed on each chromosome separately (therefore leading to potential different values of mincov per chromosome). if `.chunk.size != NA` then mincov will be first evaluated on a sample of `.sample.size` data points. Therefore for exact results, you better use a positive value for mincov.

see HELP.COORD for help on coordinates systems

See Also

baf.heterozygous.chr

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
baf.heterozygous(baf, NULL, deltafreq=0.5)
baf.heterozygous(baf, NULL, deltafreq=0.5, flatten=FALSE)
baf.close(baf)
```

`baf.heterozygous.chr` *get heterozygous positions on chromosome*

Description

get heterozygous positions on chromosome `chrindex` defines a chromosome index. this function gather all heterozygous positions defined as valid by baf.count.filter:

- coverage >= `covmin`
- `minall` <= nb_alleles <= `maxall`
- if (nb_alleles > 1) abs(0.5 - count_max_allele / coverage) <= `deltafreq`

Usage

```
baf.heterozygous.chr(handle, chr, lowread = 2L, mincov = 10L,
  deltafreq = 0.1, .chunk.size = NA, .sample.size = 1000000L)
```

Arguments

`handle`	file handle (as returned by baf.open)
`chr`	chromosome index (if integer) or chromosome name (if character)
`lowread`	low read threshold (see baf.count.filter)
`mincov`	minimal coverage (see baf.count.filter)
`deltafreq`	max allele frequency (see baf.count.filter)
`.chunk.size`	chunk size for loading chromosome (to save memory). Use NA to load in one single chunk (quicker but memory expensive). see note below for the compatibility with a negative mincov.
`.sample.size`	sample size to determine mincov for the case where .chunk.size != NA and mincov < 0 (see note).

Value

integer matrix of size n x 4 containing allele counts at heterozygous sites. with (1-based) positions as rownames.

Note

if (mincov < 0) then mincov will be estimated (`median(coverage) + mincov * mad(coverage)`) as in baf.count.filter. However the region for computing coverage will depends upon the `.chunk.size` parameter: if `.chunk.size == NA` then this will be performed on the whole chromosome. if `.chunk.size != NA` then mincov will be first evaluated on a sample of `.sample.size` data points. Therefore for exact results, you should better use a positive value for mincov.

see HELP.COORD for help on coordinates systems

See Also

baf.heterozygous.coord, baf.heterozygous.cloc

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
baf.heterozygous.chr(baf, 3, deltafreq=0.5)
baf.close(baf)
```

`baf.heterozygous.cloc` *get heterozygous positions within region*

Description

get heterozygous positions within region using relative coordinates.
`clocation` defines a region on a chromosome. this function gather all heterozygous positions defined as valid by baf.count.filter:

- coverage >= `covmin`
- `minall` <= nb_alleles <= `maxall`
- if (nb_alleles > 1) abs(0.5 - count_max_allele / coverage) <= `deltafreq`

Usage

```
baf.heterozygous.cloc(handle, clocation, lowread = 2L, mincov = 10L,
  deltafreq = 0.1)
```

Arguments

`handle`	file handle (as returned by baf.open)
`clocation`	relative clocation = c(seqname, from, to) (1-based)
`lowread`	low read threshold (see baf.count.filter)
`mincov`	minimal coverage (see baf.count.filter)
`deltafreq`	max allele frequency (see baf.count.filter)

Value

integer matrix of size n x 4 containing allele counts at heterozygous sites. with (1-based) positions as rownames.

Note

take care that if mincov < 0 the actual mincov will be computed on this region (as `median(coverage) + mincov * mad(c` not on whole chromosome nor genome. you better use a positive value for mincov.

see HELP.COORD for help on coordinates systems

See Also

baf.heterozygous.coord, baf.count.filter

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
i <- baf.name2index(baf, "machaon")
baf.heterozygous.cloc(baf, c(i, 560, 567), deltafreq=0.5)
baf.close(baf)
```

`baf.heterozygous.coord` *get heterozygous positions within region*

Description

get heterozygous positions within region using absolute coordinates.
`coord` defines a region on a chromosome. this function gather all heterozygous positions defined as valid by baf.count.filter:

- coverage >= `covmin`
- `minall` <= nb_alleles <= `maxall`
- if (nb_alleles > 1) abs(0.5 - count_max_allele / coverage) <= `deltafreq`

Usage

```
baf.heterozygous.coord(handle, coord, lowread = 2L, mincov = 10L,
  deltafreq = 0.1)
```

Arguments

`handle`	file handle (as returned by baf.open)
`coord`	absolute sequence coordinates (c(absfrom, absto)) (1-based) or a single position absfrom (this implies absto=absfrom)
`lowread`	low read threshold (see baf.count.filter)
`mincov`	minimal coverage (see baf.count.filter)
`deltafreq`	max allele frequency (see baf.count.filter)

Value

integer matrix of size n x 4 containing allele counts at heterozygous sites. with (1-based) positions as rownames.

Note

take care that if mincov < 0 the actual mincov will be computed on this region (as `median(coverage) + mincov * mad(c` not on whole chromosome nor genome. you better use a positive value for mincov.

see HELP.COORD for help on coordinates systems

See Also

baf.heterozygous.cloc, baf.count.filter

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
baf.heterozygous.coord(baf, c(33725, 33732), deltafreq=0.5)
baf.close(baf)
```

`baf.index2name` *convert seqindex to seqname*

Description

convert seqindex to seqname.
see basta.index2name this is the same function

Usage

```
baf.index2name(handle, seqindex)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`seqindex`	integer vector of 1-based sequence index

baf.name2index — *convert seqname to seqindex*

Description

convert seqname to seqindex.
see basta.name2index this is the same function

Usage

```
baf.name2index(handle, seqname)
```

Arguments

handle	basta/baf file handle (as returned by basta.open or baf.open)
seqname	character vector of sequence name(s)

baf.open — *open baf file*

Description

open baf file for reading

Usage

```
baf.open(filename)
```

Arguments

filename	baf file name

Value

baf file handle

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
baf.close(baf)
```

baf.sample *get sample of alleles counts*

Description

sample positions on all chromosomes and return allele counts

Usage

```
baf.sample(handle, chrs = NULL, sample.size = 1000000L, .seed = -1L,
  use.threads = lx.use.threads())
```

Arguments

`handle`	file handle (as returned by baf.open)
`chrs`	integer (chromosome indexes) or character (chromosome names) vector specifying which chromosomes to use. Use NULL to specify all chromosome declared in baf header.
`sample.size`	sample size (set to +Inf to collect all positions)
`.seed`	random seed (for reproducibility). use a strictly negative integer to disable seeding.
`use.threads`	(see lx.use.threads)

Value

integer matrix of size .sample.size x 4 containing allele counts. with chrindex '.' (1-based) positions as rownames.

Note

you may set sample.size to +Inf to collect allele counts on all position. but be careful this may use a very large amount of memory.

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
smp <- baf.sample(baf, sample.size=100, .seed=0, use.threads=FALSE)
baf.close(baf)
```

`baf2clocs` *make clocations spanning all chromosomes*

Description

make clocations spanning all chromosomes declared in basta/baf file

Usage

```
baf2clocs(handle)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)

Details

this is the same function as basta2clocs

Value

nx3 matrix of clocations

Note

see HELP.COORD for help on coordinates systems

`baf2coords` *make absolute coordinates spanning all chromosomes*

Description

make absolute coordinates spanning all chromosomes declared in basta/baf file

Usage

```
baf2coords(handle)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)

Details

this is the same function as basta2coords

Value

nx2 matrix of absolute coordinates (1-based)

Note

see HELP.COORD for help on coordinates systems

`basta.close` *close basta file*

Description

same as lx.close

Usage

```
basta.close(handle)
```

Arguments

`handle`	file handle (opened by basta.open)

`basta.count.cloc` *fetch symbols counts using relative clocation*

Description

fetch sequence thru basta.fetch.cloc and count the number of occurences of symbols specified in `sym` in sliding window of size `winsize`

Usage

```
basta.count.cloc(handle, clocation, truncate = TRUE, sym = c("A", "C", "G",
  "T", "other"), winsize = clocation[3] - clocation[2] + 1,
  case.sensitive = FALSE, drop = TRUE)
```

Arguments

`handle`	basta file handle (as returned by basta.open)
`clocation`	relative clocation = c(seqname, from, to) (1-based)
`truncate`	truncate 3' to seq.size if needed
`sym`	vector of strings specifying symbols to be counted. see details.
`winsize`	sliding window size (defaults to whole sequence)
`case.sensitive`	symbols in `sym` are case sensitive
`drop`	drop the last window if sequence length is not a muliple of winsize

Details

each string in `sym` specifies a set of symbols to be counted. if this set starts with '!', it means symbols **not** in set. As a special case the string "Other" is equivalent to "!ACGT".

Value

a matrix or vector of counts. if `length(sym)==1` returns a vector of symbol(s) counts for each position of the sliding window. if `length(sym)>1` returns a matrix of symbols counts with length(sym) columns and each row corresponds to each position of the sliding window.

Note

see HELP.COORD for help on coordinates systems

See Also

basta.count.coord

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
basta.fetch.cloc(fh, c(1, 1, 24))
# count all DNA symbols
basta.count.cloc(fh, c(1, 1, 24))
# count GC only
basta.count.cloc(fh, c(1, 1, 24), sym="GC")
# count GC in sliding windows
basta.count.cloc(fh, c(1, 1, 24), sym="GC", winsize=4)
basta.close(fh)
```

`basta.count.coord` *fetch symbols counts using absolute coordinates*

Description

fetch sequence thru basta.fetch.coord and count the number of occurences of symbols specified in `sym` in sliding window of size `winsize`

Usage

```
basta.count.coord(handle, coord, sym = c("A", "C", "G", "T", "other"),
  winsize = diff(range(coord)) + 1, case.sensitive = FALSE, drop = TRUE)
```

Arguments

`handle`	basta file handle (as returned by basta.open)
`coord`	absolute coordinates (c(absfrom, absto)) (1-based) or single absolute position.
`sym`	vector of strings specifying symbols to be counted. see details.
`winsize`	sliding window size (defaults to whole sequence)
`case.sensitive`	symbols in `sym` are case sensitive
`drop`	drop the last window if sequence length is not a muliple of winsize

Details

each string in `sym` specifies a set of symbols to be counted. if this set starts with '!', it means symbols **not** in set. As a special case the string "Other" is equivalent to "!ACGT".

Value

a matrix or vector of counts. if `length(sym)==1` returns a vector of symbol(s) counts for each position of the sliding window. if `length(sym)>1` returns a matrix of symbols counts with length(sym) columns and each row corresponds to each position of the sliding window.

Note

see HELP.COORD for help on coordinates systems

See Also

basta.count.coord

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
basta.fetch.coord(fh, c(1, 24))
# count all DNA symbols
basta.count.coord(fh, c(1, 24))
# count GC only
basta.count.coord(fh, c(1, 24), sym="GC")
# count GC in sliding windows
basta.count.coord(fh, c(1, 24), sym="GC", winsize=4)
basta.close(fh)
```

`basta.fetch.cloc` *fetch sequence using relative clocation*

Description

fetch sequence using relative clocation

Usage

```
basta.fetch.cloc(handle, clocation, truncate = TRUE)
```

Arguments

`handle`	basta file handle (as returned by basta.open)
`clocation`	relative clocation = c(seqname, from, to) (1-based)
`truncate`	truncate 3’ to seq.size if needed

Value

sequence string

Note

see HELP.COORD for help on coordinates systems

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
basta.fetch.coord(fh, c(25, 34))
basta.fetch.sloc(fh, "seq2:1-10")
basta.fetch.cloc(fh, c(2, 1, 10))
basta.close(fh)
```

basta.fetch.coord *fetch sequence using absolute coordinates*

Description

fetch sequence using absolute coordinates

Usage

```
basta.fetch.coord(handle, coord)
```

Arguments

`handle`	basta file handle (as returned by basta.open)
`coord`	absolute coordinates (c(absfrom, absto)) (1-based) or single absolute position.

Value

sequence string

Note

see HELP.COORD for help on coordinates systems

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
basta.fetch.coord(fh, c(25, 34))
basta.fetch.sloc(fh, "seq2:1-10")
basta.fetch.cloc(fh, c(2, 1, 10))
basta.close(fh)
```

basta.fetch.points.chr *fetch sequence at several relative point locations*

Description

`relpts` s a set of relative **point** positions on the same chromosome. this function returns sequence of length size starting at each position. this formaly equivalent to:
`clocs <- lapply(relpts, function(x) c(chrindex, x, x+size-1))`
`res <- unlist(lapply(clocs, basta.fetch.cloc, handle=handle))`
but is much quicker when relpts vector is large and values span most of the chromosome.
The idea is to load the chromosome counts by chunks of size `.chunk.size` instead of accessing each location individually (thus reducing disk access overhead).

Usage

```
basta.fetch.points.chr(handle, chr, relpts, size = 1L,
  .chunk.size = 1000000L)
```

Arguments

`handle`	file handle (as returned by basta.open)
`chr`	chromosome index (if integer) or chromosome name (if character)
`relpts`	vector of relative positions (1-based) on this chromosome
`size`	size of sequence to fetch
`.chunk.size`	<internal parameter> size of chunk. changing this parameter will only affect time or memory used, not result.

Value

array of character string giving the sequence starting at each point location.

Examples

```
basta <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
x <- basta.fetch.points.chr(basta, 1, 1:10)
y <- lx.strsplit(basta.fetch.cloc(basta, c(1,1,10)), "")
identical(x, y)
x <- basta.fetch.points.chr(basta, 1, 15:25, size=10)
basta.close(basta)
```

`basta.fetch.sloc` *fetch sequence using relative slocation*

Description

fetch sequence using relative slocation

Usage

```
basta.fetch.sloc(handle, slocation, zero.based.loc = FALSE, truncate = TRUE)
```

Arguments

`handle`	basta file handle (as returned by basta.open)
`slocation`	relative slocation ("seqname:from-to")
`zero.based.loc`	given slocation is 0-based
`truncate`	truncate 3’ to seq.size if needed

Value

sequence string

Note

see HELP.COORD for help on coordinates systems

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
basta.fetch.coord(fh, c(25, 34))
basta.fetch.sloc(fh, "seq2:1-10")
basta.fetch.cloc(fh, c(2, 1, 10))
basta.close(fh)
```

`basta.index2name` *convert seqindex to seqname*

Description

convert seqindex to seqname

Usage

```
basta.index2name(handle, seqindex)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`seqindex`	integer vector of 1-based sequence index

Value

character vector of seq name or NULL if index out of bounds

See Also

basta.name2index

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
basta.name2index(fh, 'seq1')
basta.index2name(fh, 2)
basta.name2index(fh, c('seq1', 'nothere'))
basta.index2name(fh, 1:3)
basta.close(fh)
```

basta.name2index *convert seqname to seqindex*

Description

convert seqname to seqindex

Usage

```
basta.name2index(handle, seqname)
```

Arguments

handle	basta/baf file handle (as returned by basta.open or baf.open)
seqname	character vector of sequence name(s)

Value

integer vector of 1-based sequence index or 0 if seqname not found

See Also

basta.index2name

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
basta.name2index(fh, 'seq1')
basta.index2name(fh, 2)
basta.name2index(fh, c('seq1', 'nothere'))
basta.index2name(fh, 1:3)
basta.close(fh)
```

basta.open *open basta file*

Description

open basta file for reading

Usage

```
basta.open(filename, check.crc32 = FALSE)
```

Arguments

filename	basta file name
check.crc32	perform crc32 check

Value

basta file handle

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'), check.crc32=TRUE)
basta.close(fh)
```

---

basta2clocs *make clocations spanning all chromosomes*

Description

make clocations spanning all chromosomes declared in basta/baf file

Usage

```
basta2clocs(handle)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)

Value

nx3 matrix of clocations

Note

see HELP.COORD for help on coordinates systems

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs <- basta2clocs(fh)
basta.close(fh)
```

`basta2coords` *make absolute coordinates spanning all chromosomes*

Description

make absolute coordinates spanning all chromosomes declared in basta/baf file

Usage

```
basta2coords(handle)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)

Value

nx2 matrix of absolute coordinates (1-based)

Note

see HELP.COORD for help on coordinates systems

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
coords <- basta2coords(fh)
basta.close(fh)
```

`bed.read` *read bed regions from file*

Description

read file in bed (0-based) format and return a dataframe

Usage

```
bed.read(filename)
```

Arguments

`filename`	bed file name

Value

nx3 dataframe with colnames: "chr" (character) "from" (integer), "to" (integer)

Note

`from, to` coordinates are 0-based

See Also

bed2clocs

Examples

```
bed <- bed.read(lx.system.file('samples/test.bed', 'xlx'))
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs <- bed2clocs(fh, bed)
basta.close(fh)
```

`bed2clocs` *convert bed regions to matrix of clocations*

Description

see HELP.COORD for help on coordinates systems

Usage

```
bed2clocs(handle, bed, check = TRUE)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open
`bed`	dataframe (as returned by bed.read)
`check`	check that boundaries are correct

Value

a nx3 matrix of (1-based) clocations

Examples

```
bed <- bed.read(lx.system.file('samples/test.bed', 'xlx'))
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs <- bed2clocs(fh, bed)
basta.close(fh)
```

bits2clocs *transform bitfields into matrix of clocations*

Description

considering a **list of bitfields** (each of them representing allowed positions on a chromosome), this function will recover runs of TRUE's (larger than the given threshold) on each of them and return them as a nx3 matrix of clocations. the input list should be named by the chromosome indexes.

Usage

```
bits2clocs(bits, minreg = 1L, p0 = 1L, delta = 1L,
  use.threads = lx.use.threads())
```

Arguments

`bits`	named list of bitfields (see note)
`minreg`	minimum region size
`p0`	region origin (see details)
`delta`	region size factor (see details)
`use.threads`	(see lx.use.threads)

Details

names of the `bits` parameter are chromosome indexes (in order to put them into clocations)
`p0` and `delta` are two parameters to transform indices of TRUE's in bitfields into actual positions on chromosomes according to:
`pos = p0 + (i-1) * delta`
this is useful when indices actually correspond to binned values (delta=binsize) or to regions that do not start at 1 (p0 = from)

Value

a matrix of clocations

See Also

runs2clocs for single bitfield version

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
slocs <- c("seq1:1-10", "seq1:15-20", "seq2:2-3")
clocs <- clocs.matrix(lapply(slocs, sloc2cloc, handle=fh))
bits <- clocs2bits(fh, clocs)
rclocs <- bits2clocs(bits)
identical(clocs, rclocs)
rclocs <- bits2clocs(bits, 5)
rclocs <- bits2clocs(bits, 50)
bits[[1]] <- bit::bit(length(bits[[1]]))
rclocs <- bits2clocs(bits)
```

```
bits[[2]] <- bit::bit(length(bits[[2]]))
rclocs <- bits2clocs(bits)
basta.close(fh)
```

c *Catenate two or more Dna sequences*

Description

Catenate two or more Dna sequences

Usage

```
## S3 method for class 'Dna'
c(obj, ...)
```

Arguments

`obj`	Dna object
`...`	Dna objects or character strings (may be mixed)

Value

a Dna sequence

Examples

```
x <- Dna("acgtnacgtn")
c(x, "rryy", Dna('gg'))
```

cloc2coord *transform relative clocation to absolute coordinates*

Description

see HELP.COORD for help on coordinates systems

Usage

```
cloc2coord(handle, clocation, truncate = TRUE)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`clocation`	relative clocation c(chrindex, from, to) (1-based)
`truncate`	truncate 3' to seq.size if needed

Value

absolute coordinates c(absfrom, absto) (1-based), NULL on error

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
cloc2coord(fh, c(1, 1, 10))
cloc2coord(fh, c(2, 1, 10))
basta.close(fh)
```

---

cloc2sloc — *transform relative clocation to relative slocation*

Description

see HELP.COORD for help on coordinates systems

Usage

```
cloc2sloc(handle, clocation, zero.based.loc = FALSE)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`clocation`	relative clocation c(chrindex, from, to) (1-based)
`zero.based.loc`	returned slocation should be 0-based

Value

relative slocation "chrname:from-to" (0 or 1-based), NULL on error

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
cloc2sloc(fh, c(1, 1, 10))
basta.close(fh)
```

clocations — *create a matrix of clocations from data*

Description

check if data is a proper matrix of clocations and reformat it if necessary.

Usage

```
clocations(x = NULL)
```

Arguments

x	data to reformat (see details)

Details

data can be :

- NULL : return empty matrix
- matrix : (should be nx3) then just setup colnames and storage mode
- dataframe : (should be nx3) then convert to matrix
- anything else: transform to nx3 matrix

Value

nx3 matrix of clocations with proper colnames and storage.

See Also

clocs.matrix

Examples

```
clocations() # empty clocs
clocations(list(c(1,1,10), c(2,1,10)))
clocations(c(1,1,10, 2,1,10))
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocations(lapply(c("seq1:1-10", "seq2:1-10"), sloc2cloc, handle=fh))
basta.close(fh)
```

clocs.inter — *intersect two sets of clocations*

Description

each set of clocations represents intervals on chromosomes. this function intersects (by chromosome) all intervals from the first set with all intervals from the second set and retains intervals above a specified width.

Usage

```
clocs.inter(clocations1, clocations2, minreg = 1L,
  use.threads = lx.use.threads())
```

Arguments

`clocations1`	nx3 matrix of relative clocations (1-based)
`clocations2`	mx3 matrix of relative clocations (1-based)
`minreg`	minimum interval width (see details)
`use.threads`	(see lx.use.threads)

Details

minreg parameter: all resulting intervals strictly smaller than `minreg` are discarded

Value

kx3 matrix of relative clocations (1-based)

Note

intersecting a set with itself is formally equivalent to calling clocs.reduce

require library `intervals`

this function works chromosome by chromosome to allow more efficient multithreading.

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocsF <- coords2clocs(fh, c(1:10, 25:30))
clocsF[,3] <- clocsF[,2] + 5
clocs1 <- coords2clocs(fh, 1:3)
clocs1[,3] <- clocs1[,2] + 1
clocs2 <- coords2clocs(fh, 25)
clocs2[,3] <- clocs2[,2] + 2
identical(clocs.inter(clocsF, clocsF), clocs.reduce(clocsF))
clocs.inter(clocsF, clocs1)
clocs.inter(clocsF, clocs2)
clocs.inter(clocs1, clocs2)
basta.close(fh)
```

`clocs.is.disjoint` *test if clocations are disjoint*

Description

Test if clocations are (weakly or strongly) disjoint.
weakly disjoint <=> no interval is completely included in another one (but intervals may overlap)
strongly disjoint <=> no two intervals overlap.

Usage

```
clocs.is.disjoint(clocations, strong = TRUE)
```

Arguments

`clocations`	nx3 matrix of relative clocations (1-based)
`strong`	if strongly disjoint

Value

boolean TRUE if (weakly/strongly) disjoint

Note

this function runs slightly quicker if clocations has already been sorted by clocs.sort with decreasing.to=TRUE.

Examples

```
clocs <- clocations(c(1,1,10, 1,11,20, 1,21,30, 2,5,10))
clocs.is.disjoint(clocs)
clocs <- clocations(c(1,1,10, 1,5,20, 1,10,30, 2,5,10))
clocs.is.disjoint(clocs)
clocs.is.disjoint(clocs, strong=FALSE)
clocs <- clocations(c(1,1,10, 1,5,30, 1,10,30, 2,5,10))
clocs.is.disjoint(clocs, strong=FALSE)
```

`clocs.is.empty` *test if clocations is empty*

Description

Test if clocations is empty (either null or no rows)

Usage

```
clocs.is.empty(clocations)
```

Arguments

`clocations`	nx3 matrix of relative clocations (1-based)

Value

boolean TRUE if empty.

Examples

```
clocs.is.empty(NULL)
clocs.is.empty(clocations())
clocs.is.empty(clocations(c(1,1,10)))
```

`clocs.is.unsorted` *test if clocations are **not** sorted*

Description

Test if clocations are not sorted without the cost of sorting it.

Usage

```
clocs.is.unsorted(clocations, decreasing.to = FALSE)
```

Arguments

`clocations`	nx3 matrix of relative clocations (1-based)
`decreasing.to`	boolean. should the sort order of `to` be increasing or decreasing? may be set to NA if you don't care

Details

the sort order is: first by increasing chromosome index (clocations[,1]) then, for equal chromosome index, by increasing from (clocations[,2]) then, for equal from, by increasing to if `decreasing.to==FALSE` else by decreasing to.

Value

boolean TRUE if unsorted, FALSE if sorted

Note

for coords you may use the R base function is.unsorted

See Also

clocs.sort

Examples

```
clocs <- clocations(c(1,1,5, 1,10,10, 1,10,20, 2,5,10))
clocs.is.unsorted(clocs)
clocs.is.unsorted(clocs, decreasing.to=TRUE)
clocs.is.unsorted(clocs, decreasing.to=NA)
clocs <- clocs.sort(clocs, decreasing.to=TRUE)
```

`clocs.is.valid` *clocations sanity check*

Description

Test if clocations matrix is valid i.e. that
`1 <= from <= to <= seq.len` and `1 <= chr <= nchr`.

Usage

```
clocs.is.valid(clocations, handle = NULL)
```

Arguments

`clocations`	nx3 matrix of relative clocations (1-based)
`handle`	basta/baf file handle (as returned by basta.open or baf.open). this parameter is optional (see details).

Details

if `handle` is provided then the function checks that `to <= seq.len` and `chr <= nchr` else these conditions are ignored.

Value

boolean TRUE if valid.

Note

an empty clocations is valid.

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs.is.valid(NULL)
clocs <- clocations(c(1,10,25, 2,30,10))
clocs.is.valid(clocs)
clocs <- clocations(c(1,10,25, 2,10,30))
clocs.is.valid(clocs)
clocs.is.valid(clocs, fh)
clocs <- clocations(c(1,10,24, 2,10,30))
clocs.is.valid(clocs, fh)
basta.close(fh)
```

`clocs.join` *join clocations*

Description

join consecutive clocations that are separated by at most `delta` bp and retains intervals above a specified width.

Usage

```
clocs.join(clocations, delta = 0L, minreg = 1L, .force.reduce = FALSE,
  use.threads = lx.use.threads())
```

Arguments

`clocations`	nx3 matrix of relative clocations (1-based)
`delta`	positive or zero integer. the maximal spacing between two consecutive intervals to be joined.
`minreg`	minimum interval width. all resulting intervals strictly smaller than `minreg` are discarded.
`.force.reduce`	(see details).
`use.threads`	(see lx.use.threads).

Details

This function implements two different algorithms. One (a) is very efficient if the clocations are (weakly) disjoints (see clocs.is.disjoint). The second algorithm (b) does not have this requirement but works more slowly on the average. The function will switch to the most appropriate algorithm by using clocs.is.disjoint. If you know that the disjoint condition will not be satisfied, you may force the use of algorithm (b) immediately by using the `.force.reduce` parameter.
note that in all cases the result is the same, just the execution time may vary.
the `use.threads` parameter is only active with algorithm (b).
Results are always sorted by increasing chromosome index, from and to positions (see clocs.sort).

Note

If `delta == 0` this is equivalent to clocs.reduce.

See Also

clocs.reduce

Examples

```
clocs <- clocations(c(1,1,10, 1,11,20, 1,20,30, 1,40,50, 1,60,70, 1,70,80, 1,90,100, 2,1,10))
clocs.join(clocs)
clocs.join(clocs, delta=9)
clocs <- clocs.rbind(list(clocs, c(1,1,100)))
clocs.join(clocs)
```

`clocs.matrix` *reformat data to proper matrix of clocations*

Description

this function is mostly used within other functions to properly (re)format a clocs matrix. It is quite unusual to call it directly, please consider clocations instead.

Usage

```
clocs.matrix(x = NULL)
```

Arguments

`x`	data to reformat (see details)

Details

data can be :

- NULL : return empty matrix
- matrix : (should be nx3) then just setup colnames and storage mode
- dataframe : (should be nx3) then convert to matrix
- anything else: transform to nx3 matrix

Value

nx3 matrix of clocations with proper colnames and storage.

See Also

clocations

Examples

```
clocs.matrix() # empty clocs
clocs.matrix(list(c(1,1,10), c(2,1,10)))
clocs.matrix(c(1,1,10, 2,1,10))
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs.matrix(lapply(c("seq1:1-10", "seq2:1-10"), sloc2cloc, handle=fh))
basta.close(fh)
```

clocs.rbind *catenate a list of matrices of clocations into single matrix*

Description

catenate (by rows) a list of matrices of clocations into a single matrix of clocations.

Usage

```
clocs.rbind(submatrices)
```

Arguments

`submatrices`	list of submatrices of clocations

Value

matrix of clocations

Note

this is equivalent to `Reduce(rbind, submatrices, clocs.matrix(NULL))`

See Also

clocs.rsplit for the reverse operation

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs <- coords2clocs(fh, 1:25)
x <- clocs.rsplit(clocs)
y <- clocs.rbind(x)
z <- clocs2coords(fh, y)
identical(as.integer(z[,1]), 1:25)
basta.close(fh)
```

clocs.reduce *compactly re-represent clocations*

Description

a set of clocations represents intervals on chromosomes. in general these intervals may overlap (partially or completely) or may be strictly adjacent. this function computes the union of all intervals on each chromosome in order to compact the input clocations and to produce the minimal number of clocations. It also sorts the resulting clocations by increasing chromosome index, from and to positions. Finally only intervals above a specified width are retained.

Usage

```
clocs.reduce(clocations, minreg = 1L, use.threads = lx.use.threads())
```

Arguments

`clocations`	nx3 matrix of relative clocations (1-based)
`minreg`	minimum interval width. all resulting intervals strictly smaller than `minreg` are discarded.
`use.threads`	(see lx.use.threads)

Note

this is formally equivalent (and actually implemented as):
`clocs.join(clocations, delta=0L, minreg=minreg, use.thread=use.thread)`
this version has been kept for historical reasons and to keep open the possibility of a more efficient version in the future.

See Also

clocs.join

Examples

```
clocs <- clocations(c(1,1,10, 1,11,20, 1,20,30, 1,40,50, 1,60,70, 1,70,80, 1,90,100, 2,1,10))
clocs.reduce(clocs)
clocs <- clocs.rbind(list(clocs, c(1,1,100)))
clocs.reduce(clocs)
```

`clocs.rsplit` *split matrix of clocations into submatrices*

Description

split matrix of clocations (by rows) into submatrices according to by

Usage

```
clocs.rsplit(clocations, by = clocations[, 1])
```

Arguments

`clocations`	matrix of clocations
`by`	group factors (should be of length nrow(clocations))

Value

list of submatrices (named by factors)

Note

levels in by that do not occur are dropped.

See Also

clocs.rbind for the reverse operation

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs <- coords2clocs(fh, 1:25)
x <- clocs.rsplit(clocs)
y <- clocs.rbind(x)
z <- clocs2coords(fh, y)
identical(as.integer(z[,1]), 1:25)
basta.close(fh)
```

`clocs.sort` *sort clocations*

Description

sort clocations in increasing order of chr, from and increasing or decreasing order of to.

Usage

```
clocs.sort(clocations, decreasing.to = FALSE)
```

Arguments

`clocations`	nx3 matrix of relative clocations (1-based)
`decreasing.to`	boolean. should the sort order of `to` be increasing or decreasing?

Details

the sort order is: first by increasing chromosome index (clocations[,1]) then, for equal chromosome index, by increasing from (clocations[,2]) then, for equal from, by increasing to if `decreasing.to=FALSE` else by decreasing to.

Value

sorted nx3 matrix of relative clocations

Note

for coords you may simply use the R base function sort

See Also

clocs.is.unsorted

Examples

```
clocs <- clocations(c(1,10,20, 1,10,30, 2,5,10, 1,1,100))
clocs.sort(clocs)
clocs.sort(clocs, decreasing.to=TRUE)
```

`clocs.threshold` *filter clocations by size*

Description

keep only clocations which size (`w=to-from+1`) is greater or equal to `minreg`.

Usage

```
clocs.threshold(clocations, reduce = TRUE, minreg = 1L,
  use.threads = lx.use.threads())
```

Arguments

`clocations`	nx3 matrix of relative clocations (1-based)
`reduce`	perform a clocs.reduce first
`minreg`	minimum width
`use.threads`	(see lx.use.threads) (only used if reduce==TRUE)

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs <- coords2clocs(fh, c(1:5, 25:29))
clocs[,3] <- clocs[,2] + c(sample(1:5, 5), sample(1:5, 5))
clocs.threshold(clocs, 4)
clocs[,3] <- clocs[,2] + c(sample(1:5, 5), sample(1:2, 5, replace=TRUE))
basta.close(fh)
```

`clocs.union` *unions two sets of clocations*

Description

each set of clocations represents intervals on chromosomes. this function makes union (by chromosome) of all intervals from the first set and all intervals from the second set and retains intervals above a specified width.

Usage

```
clocs.union(clocations1, clocations2, minreg = 1L,
  use.threads = lx.use.threads())
```

Arguments

`clocations1`	nx3 matrix of relative clocations (1-based)
`clocations2`	mx3 matrix of relative clocations (1-based)
`minreg`	minimum interval width (see details)
`use.threads`	(see lx.use.threads)

Details

minreg parameter: all resulting intervals strictly smaller than `minreg` are discarded

Value

kx3 matrix of relative clocations (1-based)

Note

this is formally equivalent to concatenating (by rbind) the two sets and calling clocs.reduce

require library `intervals`

this function works chromosome by chromosome to allow more efficient multithreading.

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocsF <- coords2clocs(fh, c(1:10, 25:30))
clocsF[,3] <- clocsF[,2] + 5
clocs1 <- coords2clocs(fh, 1:3)
clocs1[,3] <- clocs1[,2] + 1
clocs2 <- coords2clocs(fh, 25)
clocs2[,3] <- clocs2[,2] + 2
identical(clocs.union(clocsF, clocsF), clocs.reduce(clocsF))
clocs.union(clocsF, clocs1)
clocs.union(clocsF, clocs2)
clocs.union(clocs1, clocs2)
basta.close(fh)
```

`clocs2bits` *transform a matrix of clocations to bitfield(s)*

Description

transform a matrix of clocations to bit bitfield(s) of allowed positions on specified chromosome(s). bitfield(s) are defined in `bit` library

Usage

```
clocs2bits(handle, clocations, chrs = unique(clocations[, 1]),
  save.mem = FALSE, use.threads = lx.use.threads())
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`clocations`	nx3 matrix of relative clocations (1-based)
`chrs`	vector (possibly scalar) of chromosome indexes (not names) to work on
`save.mem`	save memory at expense of speed (see details)
`use.threads`	(see lx.use.threads)

Details

by default, this function internally works using logicals. This requires N bytes of memory per chromosome, where N is the size of each chromosome. The `save.mem` parameter will force using bitfields internally. This results in a 30 fold reduction of memory size at expense of speed. If memory is short, also consider using `use.threads = FALSE` to proceed each chromosome sequentially.

Value

named list (possibly of size 0) of bitfields

Note

require library `bit`

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
slocs <- c("seq1:1-10", "seq1:15-20", "seq2:2-5")
clocs <- clocs.matrix(lapply(slocs, sloc2cloc, handle=fh))
bits <- clocs2bits(fh, clocs)
empty <- clocs2bits(fh, clocs.matrix(NULL))
basta.close(fh)
```

`clocs2coords` *transform matrix of relative clocations to matrix of absolute coordinates*

Description

transform a nx3 matrix of relative clocations to a nx2 matrix of absolute coordinates
see HELP.COORD for help on coordinates systems

Usage

```
clocs2coords(handle, clocations)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`clocations`	nx3 matrix of relative clocation (1-based)

Value

nx2 matrix of absolute coordinates (1-based)

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))

x <- sample(1:50, 10, replace=TRUE)
coo <- cbind(x, x)
clocs <- coords2clocs(fh, coo)
plocs <- coords2clocs(fh, x)
identical(clocs, plocs)

x <- sample(25:30, 10, replace=TRUE)
coo <- cbind(from=x, to=x+10)
clocs <- coords2clocs(fh, coo)
rcoo <-clocs2coords(fh, clocs)
identical(coo, rcoo)

clocs <- coords2clocs(fh, matrix(0, ncol=2, nrow=0))
clocs2coords(fh, clocs)
clocs <- coords2clocs(fh, matrix(1, ncol=2, nrow=1))
clocs2coords(fh, clocs)

basta.close(fh)
```

`clocs2llocs` *convert clocations to llocations*

Description

convert a nx3 matrix of clocations to a list of mx2 locations. (see HELP.COORD for help on coordinates systems)

Usage

```
clocs2llocs(clocations)
```

Arguments

`clocations`	nx3 matrix of relative clocations (1-based)

Value

named list of mx2 relative locations per chromosome

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs <- coords2clocs(fh, 1:25)
llocs <- clocs2llocs(clocs)
rclocs <- llocs2clocs(llocs)
identical(clocs, rclocs)
basta.close(fh)
```

`compl` *Generic method to complement a sequence*

Description

just complement sequence (not reverse complement)

Usage

```
compl(obj)
```

Arguments

`obj`	a Dna sequence to complement

See Also

compl.Dna, revcompl

`compl.Dna` *Complement Dna sequence*

Description

just complement sequence (not reverse complement)

Usage

```
## S3 method for class 'Dna'
compl(obj)
```

Arguments

`obj`	a Dna sequence to complement

See Also

revcompl.Dna

Examples

```
x <- Dna("acgtnry")
compl(x)
```

coord2cloc *transform absolute coordinates to relative clocation*

Description

see HELP.COORD for help on coordinates systems

Usage

```
coord2cloc(handle, coord, truncate = TRUE)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`coord`	absolute coordinates c(absfrom, absto) (1-based) or single absolute position
`truncate`	truncate 3’ to seq.size if needed

Value

relative clocation c(chrindex, from, to) (1-based), NULL on error

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
coord2cloc(fh, c(1, 10))
coord2cloc(fh, c(25, 34))
basta.close(fh)
```

coord2sloc *transform absolute coordinates to relative slocation*

Description

see HELP.COORD for help on coordinates systems

Usage

```
coord2sloc(handle, coord, zero.based.loc = FALSE, truncate = TRUE)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`coord`	absolute coordinates c(absfrom, absto) (1-based) or single absolute position
`zero.based.loc`	given slocation is 0-based
`truncate`	truncate 3’ to seq.size if needed

Value

relative slocation "chrname:from-to" (0 or 1-based), NULL on error

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
coord2sloc(fh, c(1, 10))
coord2sloc(fh, c(25, 34))
basta.close(fh)
```

`coords.sample` *sample absolute point locations on chromosomes*

Description

sample locations within regions specified by clocations.

Usage

```
coords.sample(handle, clocations, size = 1000000L, replace = FALSE)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`clocations`	regions where we can sample (endpoints included)
`size`	number of points to sample
`replace`	sample with replacement (see lx.sample)

Value

vector of `size` absolute point coordinates. (see coords2clocs to transform into clocations)

Note

if replace=FALSE and N, the number of points in the union of all regions, is less than 2*size then downsample to N/2 points

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs <- basta2clocs(fh)
samp <- coords.sample(fh, clocs, size=10)
coords2clocs(fh, sort(samp))
basta.close(fh)
```

`coords2clocs` *transform matrix of absolute coordinates to matrix of relative clocations*

Description

transform a nx2 matrix of absolute coordinates (or a vector of point coordinates) to nx3 matrix of relative clocations.
see HELP.COORD for help on coordinates systems

Usage

```
coords2clocs(handle, coords)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`coords`	nx2 matrix of absolute coordinates (1-based) or vector of n absolute point coordinates

Value

nx3 matrix of relative clocation (1-based)

Note

if some absolute coordinates span several chromosomes then the corresponding rows are discarded.

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))

x <- sample(1:50, 10, replace=TRUE)
coo <- cbind(x, x)
clocs <- coords2clocs(fh, coo)
plocs <- coords2clocs(fh, x)
identical(clocs, plocs)

x <- sample(25:30, 10, replace=TRUE)
coo <- cbind(from=x, to=x+10)
clocs <- coords2clocs(fh, coo)
rcoo <-clocs2coords(fh, clocs)
identical(coo, rcoo)

clocs <- coords2clocs(fh, matrix(0, ncol=2, nrow=0))
clocs <- coords2clocs(fh, matrix(1, ncol=2, nrow=1))

basta.close(fh)
```

countsymb *Generic method to count symbols in sequence*

Description

Generic method to count symbols in sequence

Usage

```
countsymb(obj, symb)
```

Arguments

obj	sequence object (usually Dna)
symb	a character string containing symbols to count

Value

count table

See Also

countsymb.Dna for Dna sequence

countsymb.Dna *Count symbols in Dna Sequence*

Description

Count symbols in Dna Sequence

Usage

```
## S3 method for class 'Dna'
countsymb(obj, symb = "acgt")
```

Arguments

obj	a Dna sequence
symb	a character string containing symbols to count

Note

To count using IUPAC degenerated codes, use uppercase symbols (eg symb='W' will count g + c). Therefore symb='r' will count the number of strict r's whereas 'R' will sum counts for r's and g's and a's.

Examples

```
seq <- paste(sample(c('a', 'c', 'g', 't'), 1e7, replace=TRUE), collapse='')
system.time(cnt <- length(gregexpr('a', seq)[[1]]))
x <- Dna(seq, 'strict')
system.time(cnt <- countsymb(x, 'a'))
system.time(tab <- countsymb(x))
countsymb(x, 'W')
```

Dna *Dna Class constructor*

Description

Dna Class constructor

Usage

```
Dna(x = 0L, code = c("standard", "strict", "pattern"), pattern = NULL)
```

Arguments

`x`	either an integer, a character string or another Dna object
`code`	encoding scheme (see details)
`pattern`	pattern to use (only used if code='pattern)

Details

x parameter: if x is an integer, creates an empty Dna sequence of size x
if x is a character string, creates a Dna sequence representing x
if x is a Dna object, same as `Dna(as.character(x), ...)`

code parameter: if code == 'standard' use 3 bits/symbol to represent symbols in [acgtryn]
if code == 'strict' use a 2 bits/symbol to represent symbols in [acgt]
if code == 'pattern' use a 2 bits/symbol to represent pattern by [x.]

Examples

```
x <- Dna("acgtnnactg")
x <- Dna("acgtacgt", 'strict')
x <- Dna("acgtnnnacgtg", 'pattern', '[gc]')
x <- Dna(10)
x[] <- 'accn'
x <- Dna(10, 'strict')
x <- Dna(10, 'pattern', '[gc]')
x[] <- 'gca'
```

`hamming` *Generic method to compute Hamming distance between two sequences*

Description

Generic method to compute Hamming distance between two sequences

Usage

```
hamming(obj1, obj2)
```

Arguments

`obj1`	first sequence object (usually Dna)
`obj2`	second sequence object (usually Dna)

See Also

hamming.Dna for Dna sequence

`hamming.Dna` *Hamming distance between two Dna sequences*

Description

Hamming distance between two Dna sequences

Usage

```
## S3 method for class 'Dna'
hamming(obj1, obj2)
```

Arguments

`obj1`	a Dna sequence
`obj2`	a Dna sequence of same size and encoding as obj1

Value

number of differences between obj1 and obj2 or -1 if obj1 and obj2 are not of same size or encoding

Note

sequence case and IUPAC codes are ignored

Examples

```
hamming(Dna('acgtacgtacgt'), Dna('acggacggacgg'))
```

HELP.BAF *Baf format*

Description

Baf is a binary format to compactly represents reads alignments from bam files. It basically only keeps the information about each allele counts at each position of the chromosomes (all other information such as alignement quality, CIGAR etc... is discarded). It can therefore be used to retrieve allelic frequencies, total cover (sum of all alleles) or GC content (sum of G and C alleles).

This is a little endian binary file composed of a header:

int32	0x62696d31	magic number ('baf1')
int32	nbseq	number of sequences
—-	————	——-
		repeat nbseq times
—-	————	——-
string	namei	name of sequence i (see 1)
int64	sizei	length of sequence i
int32	codei	encoding size in bytes for this sequence (0, 1,2,4)
—-	————	——-

followed by the concatenation of nbseq arrays each of 4 * sizei * codei bytes. each codei bytes (codei=1,2 or4) represent total number of read bases (i.e non counting deletions) covering each position.
(codei=1: unsigned char, codei=2: unsigned short, codei=4: unsigned int32, and codei=0 means that all counts on this chromosome are 0)
.

(1) string format is:

int32	size	string length
bytes	size + 1	NULL terminated char array

Note

When opening a baf file, the header is loaded into memory but not the count arrays. Counts will be directly accessed from disk when needed.

Conversion from bam to baf is performed by the external C executable `bam2baf` provided in Csrc directory.

Baf header is compatible with Basta header (see HELP.BASTA) and Baf handles can therefore be passed as the handle argument of most functions accepting a Basta handle (except of course for those that need to access to sequence).

of course when using together a Basta and a Baf file, you should ensure that sequences in both header are stricly identical. In practice this means that the Bam file from which the baf file was generated was built using the same fasta sequences from which the Basta file was generated.

HELP.BASTA *Basta format*

Description

Basta file format is similar to Fasta format but allow indexed access to sequences.
this is a little endian binary file composed of a header:

int32	0x62617332	magic number ('bas2')
int32	nbseq	number of sequences
—-	———	——
		repeat nbseq times
—-	———	——
string	namei	name of sequence i (see 1)
int64	sizei	length of sequence i
int32	crc32i	crc32 of sequence i
—-	———	——

followed by the concatenation of all sequences as character arrays (not NULL terminated).

(1) string format is:

int32	size	string length
bytes	size + 1	NULL terminated char array

Note

When opening a basta file, the header is loaded into memory but not the sequences. Sequences are directly accessed from disk.

Conversion from fasta to basta is performed by the external C executable `fasta2basta` provided in Csrc directory.

HELP.COORD *Coordinate systems*

Description

XLX tools use three coordinates system:

Relative string coordinates, called `sloc`: a `sloc` is a string of the form :
`"chrname:from-to"` or `"chrname:from:to"`
where `chrname` is the sequence name (not the sequence index) `from` and `to` can be either 1-based (default) or 0-based (by specifiying the zero.based.loc=TRUE option)

Relative coordinates, called `cloc`: a `cloc` is a 1-based `slocation` of the form:
`c(chrindex, from, to)` where `chrindex` is the (1-based) index of sequence entry in basta file.
`chrindex`, `from` and `to` are (32 bits) integers.

Absolute coordinates, called `coord`: a `coord` represents two absolute positions in the catenated chromosomes, of the form `c(from, to)`.
absolute coordinates are always 1-based
`from` and `to` are (64 bits) doubles actually representing integers (there is no loss of precision until 53 bits i.e. 9,007,199,254,740,992)

It is not memory nor speed efficient to manipulate large amount of clocations as lists (it uses about 64 bytes per clocation).
Instead, XLX manipulate sets of clocations by matrices or list of matrices (this uses 12 or 8 bytes per clocation and is much quicker to operate).

Matrix of clocations, called `clocs`: a `clocs` is a nx3 matrix. each row is a clocation chrindex, from, to. columns are named: "chr", "from", "to" respectively. if necessary you may convert it to a dataframe by: `as.data.frame(clocs)`. note that "chr" is (as in cloc) a chrindex **not** a chr name. use `basta.index2name` or `basta.name2index` to transform between names and indexes

List of matrices of locations, called `llocs`: a `llocs` is a more memory efficient version of `clocs`. this is a named list of matrices. each element of the list is named by a chromosome index (as character) and contains an mx2 matrix of from, to relative positions on this chromosome.

Matrix of coordinates, called `coords`: a `coords` is a nx2 matrix. each row is a coord absfrom, absto. columns are named: "from", "to" respectively. if necessary you may convert it to a dataframe by: `as.data.frame(coords)`

Summary:

single location

shortname	**name**	**definition**	**base**
sloc	relative slocation	"chrname:from-to"	0 or 1-based
cloc	relative clocation	c(chrindex, from, to)	1-based
coord	absolute coordinates	c(absfrom, absto)	1-based offset

multiple locations

shortname	**name**	**definition**	**base**
clocs	matrix of clocations	nx3 matrix	1-based
llocs	list of matrices of locations	n list of mx2 matrices	1-based
coords	matrix of coordinates	nx2 matrix	1-based

Note

in 1-based system endpoints are included : [from, to]
in 0-based system the right endpoint is excluded : [from, to[
the conversion between 0-based and 1-based is therefore
from0 = from1 - 1
to0 = to1

conversion between coordinate systems is performed by the <xxx>2<yyy> functions (e.g. coord2cloc) when extracting rows (or cols) from matrix, don't forget to add the `drop=FALSE` last subscript, in order to avoid spurious coercing to vector when selecting a single row (or col). (eg: `clocs[1,,drop=FALSE]`)

`HELP.DNA` *Dna Class*

Description

Dna is an S3 Class that let you manipulate DNA sequences with a more memory-efficient way than usual character strings.
More precisely **Dna** stores sequence with a 3, 2 or 1 bits/symbol instead of 8 bits for character strings.

Dna currently works with the following restrictions:

- the DNA alphabet is lowercase and restricted to 'acgtryn' or 'acgt' depending upon the storage mode.
- maximum sequence size is 2^31-1 (this is the same limitation as for character strings, until R internally goes to 64 bits vector indexes)

creation: Dna sequences are created by the Dna constructor or as.Dna coercion.

manipulation: Dna sequences can be transformed to strings by as.character
Access to sequence components is performed by subscripting (either as extracting or replacing): `dna[index]` and `dna[index] <- seq`

misc: other **Dna** operations include:
c, length, subseq, summary, rev, compl, revcompl, plot etc.

Note

Dna requires the bit library

the `print` S3 method has been redefined. For debugging purpose, you may use `unclass(obj)` to see the actual internal components.

Examples

```
# generate a 10 Mb sequence
n <- 1e7
seq <- paste(sample(c('a', 'c', 'g', 't'), n, replace=TRUE), collapse='')
x <- Dna(seq, 'strict')
length(x)
summary(x)
s <- as.character(x)  # identical to seq
# extract or replace subsequences
x[1:20]
x[20:1]
x[1:20] <- 'acgt'
x[1:20] <- 'acgtn'  # will complain
x[]  # same as Dna(x) or as.Dna(x) or x
# some operations
revcompl(x)
```

```
countsymb(x, 'gc')
countsymb(x, 'W')
plot(x) # guess what
# some funny constructors
x <- Dna(15)
x[] <- 'acg'
as.character(x[seq.int(1, length(x), by=3)])
```

HELP.DNA.ENCODING *Dna Internal Encoding Scheme*

Description

this section describes the internal Dna bits encoding schemes and is intended for developpers only.

code	**length(bits)**
standard	3
strict	2
pattern	1

standard encoding:

i	**bits[[i]]**
1	[acgt]
2	[cgny]
3	[gtnr]

symb	**bit1**	**bit2**	**bit3**
g	1	1	1
c	1	1	0
t	1	0	1
a	1	0	0
n	0	1	1
y	0	1	0
r	0	0	1
x	0	0	0

strict encoding:

i	**bits[[i]]**
1	[gc]
2	[gt]

symb	**bit1**	**bit2**
g	1	1
c	1	0
t	0	1
a	0	0

pattern encoding:

i	**bits[[i]]**
1	pattern

symb	**bit1**
pattern	1
!pattern	0

`HELP.OBO` *lx in-memory database parser for Obo format*

Description

read and parse Obo database in flat file format and hold results in memory

the main functions are : `mdb.obo.read` and `mdb.obo.load`

Note

these functions hold all the database in memory and are therefore not intented for large databases

Examples

```
db <- mdb.obo.load(lx.system.file('samples/test_obo', 'xlx'))

# get entry names

names(db)

# get info about specific entry :

db$GO.0000001
db$GO.0000001$id
db$GO.0000001$name

# search for entries matching pattern :

mdb.find(db, 'def', 'mitochondrial', ignore.case=TRUE)
```

HELP.SWISS *lx in-memory database parser for Uniprot/Swissprot format*

Description

read and parse Uniprot/Swissprot database in flat file format and hold results in memory

the main functions are : `mdb.swiss.read` and `mdb.swiss.load`

Note

these functions hold all the database in memory and are therefore not intented for large databases

Examples

```
db <- mdb.swiss.read(lx.system.file('samples/test_swiss.dat', 'xlx'))

# get entry names
names(db)

# get info about specific entry :
db$P04395
db$P04395$OC
db$P04395$DR$PROSITE

# search for entries matching pattern :
mdb.find(db, 'KW', 'gluconate', ignore.case=TRUE)
```

is.Dna *test for Dna Class*

Description

test for Dna Class

Usage

```
is.Dna(obj)
```

Arguments

obj	object to be tested

`length` *Length method for Dna*

Description

Length of Dna sequence

Usage

```
## S3 method for class 'Dna'
length(x)
```

Arguments

`x`	a Dna object

Value

length of Dna sequence

`llocs2clocs` *convert llocations to clocations*

Description

convert a named list of mx2 locations to a nx3 matrix of clocations. (see HELP.COORD for help on coordinates systems)

Usage

```
llocs2clocs(llocations)
```

Arguments

`llocations`	named list of mx2 relative locations per chromosome (see note)

Value

nx3 matrix of relative clocations (1-based)

Note

llocations must be named by chromosome indexes (as character)

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
clocs <- coords2clocs(fh, 1:25)
llocs <- clocs2llocs(clocs)
rclocs <- llocs2clocs(llocs)
identical(clocs, rclocs)
basta.close(fh)
```

mdb.find — *grep pattern in specified records of database*

Description

grep pattern in specified records of database

Usage

```
mdb.find(db, key, pat, regex = TRUE, ignore.case = FALSE,
  full.match = FALSE)
```

Arguments

`db`	a flat db or a structured db (in the later case the actual db used is db$db)
`key`	record key to search in
`pat`	pattern to search for
`regex`	pat is a regular expression
`ignore.case`	ignore case during search
`full.match`	pattern should span all the key value. equivalent to '^pattern$' but when used with regex=FALSE and ignore.case=FALSE the engine may use precompiled entries for speedup

Value

vector (possible of 0 length) of records id (as strings)

See Also

`mdb.swiss.load` `mdb.obo.load`

Examples

```
db <- mdb.swiss.load(lx.system.file('samples/test_swiss', 'xlx'))
mdb.find(db, 'KW', 'gluconate', ignore.case=TRUE)
```

mdb.gaf.filter — *filter gaf table and keep only geneID <-> termID associations*

Description

filter gaf table and keep only geneID <-> termID associations

Usage

```
mdb.gaf.filter(gaf, DB = c("User", "UniProtKB"), GN = NULL,
  gid.col = "DB.Object.Symbol", tid.col = "GO.ID", no.qual = TRUE)
```

Arguments

`gaf`	gaf dataframe (from mdb.gaf.read)
`DB`	source DBs to keep (keep all if NULL)
`GN`	gene IDs to keep (keep all if NULL)
`gid.col`	column index of geneID
`tid.col`	column index of termID
`no.qual`	if TRUE (dft) remove entries with (non empty) qualifiers else keep all entries, including the repugnant **NOT** qualifier, (this may therefore lead to plain wrong associations)

Value

a dataframe with two columns: 'gid', 'tid' specifying the association

See Also

mdb.gaf.read

Examples

```
tab <- mdb.gaf.read(lx.system.file('samples/test_gaf.dat', 'xlx'))
tab <- mdb.gaf.filter(tab)
tab[tab$gid=="VPS4A",]
```

`mdb.gaf.read` *read Gene Association (GAF) file*

Description

read Gene Association (GAF) file

Usage

```
mdb.gaf.read(pathname)
```

Arguments

`pathname`	pathname of GAF file to read

Details

GAF 1.0 or 2.0 specifications

Column	**Content**	**Required?**	**Cardinality**	**Example**	**Dataframe_Colnan**
1	DB	required	1	UniProtKB	DB
2	DB Object ID	required	1	P12345	DB.Object.ID
3	DB Object Symbol	required	1	PHO3	DB.Object.Symbol
4	Qualifier	optional	0 or greater	NOT	Qualifier
5	GO ID	required	1	GO:0003993	GO.ID
6	DB:Reference (|DB:Reference)	required	1 or greater	PMID:2676709	DB.Reference

7	Evidence Code	required	1	IMP	Evidence.Code
8	With (or) From	optional	0 or greater	GO:0000346	With.or.From
9	Aspect	required	1	F	Aspect
10	DB Object Name	optional	0 or 1	Toll-like receptor 4	DB.Object.Name
11	DB Object Synonym (|Synonym)	optional	0 or greater	hToll|Tollbooth	DB.Object.Synonyr
12	DB Object Type	required	1	protein	DB.Object.Type
13	Taxon(|taxon)	required	1 or 2	taxon:9606	Taxon
14	Date	required	1	20090118	Date
15	Assigned By	required	1	SGD	Assigned.By
16	Annotation Extension	optional	0 or greater	part_of(CL:0000576)	Annotation.Extensio
17	Gene Product Form ID	optional	0 or 1	UniProtKB:P12345-2	Gene.Product.Form.

Value

a dataframe with columns corresponding to GAF 1.0 or 2.0 specifications (see details)

Note

the GAF file may be provided in plain text or gzipped format. this is checked automaticaly, there is no need to add the .gz extension.

See Also

mdb.gaf.filter

Examples

```
tab <- mdb.gaf.read(lx.system.file('samples/test_gaf.dat', 'xlx'))
tab <- mdb.gaf.filter(tab)
tab[tab$gid=="VPS4A",]

tac <- mdb.gaf.read(lx.system.file('samples/test_gaf_compressed.dat', 'xlx'))
identical(tab, tac)
```

`mdb.obo.get.ancestors` *get ancestors of go.id(s)*

Description

get ancestors of go.id(s)

Usage

```
mdb.obo.get.ancestors(db, go.id, max.depth = Inf)
```

Arguments

`db`	go db opened by mdb.obo.read or mdb.obo.load
`go.id`	entry id
`max.depth`	maximum depth of ancestor

Value

list of ancestors go.id's

See Also

mdb.obo.index.ancestors

Examples

```
db <- mdb.obo.load(lx.system.file('samples/test_obo', 'xlx'))
mdb.obo.get.ancestors(db, 'GO.0000083')
```

`mdb.obo.index.ancestors` *get indexed array of ancestors*

Description

get a list indexed by goids, giving for each entry the list of ancestors for this goid.
with goids=names(db) or restrict=TRUE, this is formally equivalent to but much quicker than :
`sapply(goids, function(x) mdb.obo.get.ancestors(db, x))`
with restrict=FALSE the resulting list includes entries for goids as well their ancestors ids.

Usage

```
mdb.obo.index.ancestors(db, goids = names(db), restrict = TRUE)
```

Arguments

`db`	go db opened by mdb.obo.read or mdb.obo.load
`goids`	set of goid's to index
`restrict`	result to goids only (do not include entries for their ancestors)

Value

named list of ancestors for each goids (and optionally their ancestors)

See Also

mdb.obo.get.ancestors

Examples

```
db <- mdb.obo.load(lx.system.file('samples/test_obo', 'xlx'))
anc <- mdb.obo.index.ancestors(db)
anc['GO.0000083']
mdb.obo.get.ancestors(db, 'GO.0000083')
```

`mdb.obo.load` *quick load obo db*

Description

this is a quicker version of mdb.obo.read
`mdb.obo.load` try to recover a previously loaded and serialized file called : dbname.rds
if it does not exist then it reads the flat file called : 'dbname.dat' and further serialize the result into dbname.rds
you may force to ignore the serialized version by using force=TRUE

Usage

```
mdb.obo.load(dbname, force = FALSE, local = TRUE)
```

Arguments

`dbname`	filename (without extension) of obo flatfile
`force`	force read and serialize even if serialized file already exists
`local`	if TRUE (default), serialized DB is saved and/or loaded in current directory else in `dirname(dbname)`.

Value

a list indexed by GO terms (under the form GO.<number> not GO:<number>)
each element is a list indexed by the record key
each recordkey element is either the raw line(s) or the result of a specific parser
current parsers are provided for : id, is_a, relationship and xref

Note

you may add your own function `.mdb.obo.parse.<key>(rec)` to parse other keys than (id, is_a, relationship and xref).

See Also

`mdb.obo.read`

Examples

```

db <- mdb.obo.load(lx.system.file('samples/test_obo', 'xlx'))

# get entry names
names(db)

# get info about specific entry :

db$GO.0000001
db$GO.0000001$id
db$GO.0000001$name
```

```
# search for entries matching pattern :

mdb.find(db, 'def', 'mitochondrial', ignore.case=TRUE)
```

`mdb.obo.parse` *obo main parsing driver (internal use)*

Description

obo main parsing driver (internal use)

Usage

```
mdb.obo.parse(key, rec)
```

Arguments

`key`	key to parse (currently AC,OC,KW,DR,seq)
`rec`	record to process

Note

call function `.mdb.obo.parse.<key> if it exists`

`mdb.obo.read` *read obo flat file and parse fields*

Description

read obo flat file and parse fields

Usage

```
mdb.obo.read(pathname)
```

Arguments

`pathname`	pathname of obo file to read

Value

a list indexed by GO terms (under the form GO.<number> not GO:<number>)
each element is a list indexed by the record key
each recordkey element is either the raw line(s) or the result of a specific parser
current parsers are provided for : id, is_a, relationship and xref.
in addition a pseudo-key named 'parent_of' is added, representing the reverse of 'is_a' relationship.

Note

you may add your own function `.mdb.obo.parse.<key>(rec)` to parse other keys than (id, is_a, relationship and xref).
`rec` is a string containing all lines of the current record to be parsed (with newlines as \n) and your function may return whatever is appropriate (usually a list).

the obo file may be provided in plain text or gzipped format. this is checked automaticaly, there is no need to add the .gz extension.

See Also

mdb.obo.load

Examples

```
db <- mdb.obo.read(lx.system.file('samples/test_obo.dat', 'xlx'))

# get entry names
names(db)

# get info about specific entry :

db$GO.0000001
db$GO.0000001$id
db$GO.0000001$name

# search for entries matching pattern :

mdb.find(db, 'def', 'mitochondrial', ignore.case=TRUE)

dc <- mdb.obo.read(lx.system.file('samples/test_obo_compressed.dat', 'xlx'))
identical(db, dc)
```

`mdb.swiss.load` *quick load swissprot db*

Description

this is a quicker version of mdb.swiss.read
`mdb.swiss.load` try to recover a previously loaded and serialized file called : `dbname.<sort_extra>.rds`
(where <sort_extra> is a '_' separated string of sorted extra, see below)
if it does not exist then it reads the flat file called : `dbname.dat` and further serialize the result into `dbname.<sort_extra>.rds`
you may force to ignore the serialized version by using force=TRUE

Usage

```
mdb.swiss.load(dbname, extra = "ALL", force = FALSE, local = TRUE)
```

Arguments

`dbname`	filename (without extension) of uniprot db
`extra`	string comma-separated list of additional lines to parse (e.g 'DE,OS,OC,KW') if empty only the default ID, AC, and ' ' (sequence) lines are parsed if 'ALL' then all keys are parsed
`force`	force read and serialize even if serialized file already exists
`local`	if TRUE (default), serialized DB is saved and/or loaded in current directory else in `dirname(dbname)`.

Value

a list indexed by records primary AC
each element is a list indexed by the record key (except sequence that is indexed by 'seq')
each recordkey element is either the raw line(s) or the resutl of a specific parser
current parsers are provided for : AC, OC, KW, DR

Note

you may add your own function `.mdb.swiss.parse.<key>(rec)` to parse other keys than (AC, OC, KW, DR and seq).

See Also

mdb.swiss.read

Examples

```
db <- mdb.swiss.load(lx.system.file('samples/test_swiss', 'xlx'))

# reload serialized version
db <- mdb.swiss.load(lx.system.file('samples/test_swiss', 'xlx'))

# get entry names
names(db)

# get info about specific entry :
db$P04395
db$P04395$OC
db$P04395$DR$PROSITE

# search for entries matching pattern :
mdb.find(db, 'KW', 'gluconate', ignore.case=TRUE)

# remove local serialized DB
unlink("test_swiss.ALL.rds")
```

`mdb.swiss.parse` *swiss parse driver*

Description

parse the content of key in record.

Usage

```
mdb.swiss.parse(key, rec)
```

Arguments

`key`	key to parse (currently AC,OC,KW,DR,seq)
`rec`	record to process

Details

this function just acts as a selector to call function `.mdb.swiss.parse.<key>` if it exists or return record if it does not.
you may add your own function `.mdb.swiss.parse.<key>(rec)` to parse other keys than (AC, OC, KW, DR and seq).
`rec` is a string containing all lines of the current record to be parsed (with newlines as \n) and your function may return whatever is appropriate (usually a list).

Value

anything that should be stored under key in record.

`mdb.swiss.read` *read swissprot db*

Description

read swissprot db and parse ID,AC,seq + extra fields as requested

Usage

```
mdb.swiss.read(pathname, extra = "ALL")
```

Arguments

`pathname`	pathname of uniprot file to read
`extra`	string comma-separated list of additional lines to parse (e.g 'DE,OS,OC,KW') if empty only the default ID, AC, and seq (sequence) lines are parsed if 'ALL' then all keys are parsed

Value

a list indexed by records primary AC
each element is a list indexed by the record key (sequence is indexed by 'seq')
each recordkey element is either the raw line(s) or the result of a specific parser
current parsers are provided for : AC, OC, KW, DR (see note)

Note

you may add your own function `.mdb.swiss.parse.<key>(rec)` to parse other keys than (AC, OC, KW, DR and seq).
`rec` is a string containing all lines of the current record to be parsed (with newlines as \n) and your function may return whatever is appropriate (usually a list).

the uniprot file may be provided in plain text or gzipped format. this is checked automaticaly, there is no need to add the .gz extension.

See Also

`mdb.swiss.load`

Examples

```
db <- mdb.swiss.read(lx.system.file('samples/test_swiss.dat', 'xlx'))

# get entry names
names(db)

# get info about specific entry :
db$P04395
db$P04395$OC
db$P04395$DR$PROSITE

# search for entries matching pattern :
mdb.find(db, 'KW', 'gluconate', ignore.case=TRUE)

dc <- mdb.swiss.read(lx.system.file('samples/test_swiss_compressed.dat', 'xlx'))
identical(db, dc)
```

`patbits` *get bitfield of pattern matches in sequence*

Description

get bitfield of pattern matches in sequence

Usage

```
patbits(seq, pat, regex = FALSE)
```

Arguments

`seq`	sequence string
`pat`	pattern to match
`regex`	pattern `pat` is a regular expression (see details)

Details

if `regex==FALSE` then match any char in string `pat`

Value

bitfield (see package bit) of same size as sequence where TRUE's indicate start positions of pattern.

Examples

```
seq <- "ACGTACGTAC"
x <- patbits(seq, 'GC')
bit::as.which(x)
sum(x)
x <- patbits(seq, '[GC]', regex=TRUE)
x <- patbits(seq, 'TAC', regex=TRUE)
```

`plot` *Plot method for Dna*

Description

just for fun... this is for Jean ;-)

Usage

```
## S3 method for class 'Dna'
plot(x, step = ceiling(length(x)/100), compass = list(x =
  c(0L, -1L, 1L, 0L), y = c(1L, 0L, 0L, -1L)), ...)
```

Arguments

`x`	Dna object
`step`	walking step (in bp)
`compass`	integer vector of length 4 giving the direction for (a c g t)
`...`	additional arguments to plot

`print` *Print method for Dna*

Description

Print Dna sequence

Usage

```
## S3 method for class 'Dna'
print(x, ...)
```

Arguments

`x`	a Dna object
`...`	further arguments passed to or from other methods.

`regions.bincover` *get binned coverage in regions*

Description

loop over given `regions` and collect mean coverage in adjacent windows of size `binsize`.

Usage

```
regions.bincover(handle, regions = baf2clocs(handle), binsize = 10000L,
  use.threads = lx.use.threads())
```

Arguments

`handle`	baf file handle (as returned by baf.open)
`regions`	clocations regions to bin (default is regions spanning all chromosomes)
`binsize`	size of bins
`use.threads`	(see lx.use.threads)

Value

a list of length nrow(regions), each element is a numerical vector of mean coverage in adjacent windows of size `binsize` in the region.

See Also

regions.bycover.range, regions.bycover.band

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
x <- regions.bincover(baf, binsize=1000, use.threads=FALSE)
baf.close(baf)
```

`regions.byacgt` *get regions with only [agct] symbols*

Description

loop over given `init` regions. foreach of them split and keep regions containing only a,c,g or t's.

Usage

```
regions.byacgt(handle, init = basta2clocs(handle), minreg = 10000L,
  use.threads = lx.use.threads())
```

Arguments

`handle`	basta file handle (as returned by basta.open)
`init`	regions to start with (default is regions spanning all chromosomes)
`minreg`	minimum region size
`use.threads`	(see lx.use.threads)

Value

a nx3 matrix of (1-based) clocations

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
x <- regions.byacgt(fh, minreg=1, use.threads=FALSE)
basta.close(fh)
```

`regions.bybed` *get user's defined regions from bed files*

Description

loop over provided bed files, intersect their regions, filter out small regions and returns clocations.

Usage

```
regions.bybed(handle, filenames, init = basta2clocs(handle), minreg = 1L,
  check = TRUE, file.stop = FALSE, use.threads = lx.use.threads())
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`filenames`	a character vector of filenames
`init`	regions (clocations) to start with (default is regions spanning all chromosomes)
`minreg`	minimum region size (see clocs.inter)
`check`	check that region boundaries are correct (see bed2clocs)
`file.stop`	boolean, stops if a bed file is not found
`use.threads`	(see lx.use.threads)

Value

a nx3 matrix of (1-based) clocations

Note

the function also checks if each file exists and will skip over or stop on non-existing file(s)

See Also

bed.read, bed2clocs and basta2clocs

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
bedfile <- lx.system.file('samples/test.bed', 'xlx')
clocs <- regions.bybed(fh, bedfile)
# this is the same as:
clocs2 <- clocs.reduce(bed2clocs(fh, bed.read(bedfile)))
#
identical(clocs, clocs2)
basta.close(fh)
```

`regions.bycover.band` *select regions in band of coverage distribution*

Description

compute the distribution of (mean) coverage in all `init` regions, select a band in this distribution according to different models (see details).
then loop over given `init` regions. foreach of them split and keep regions with coverage in that band.

Usage

```
regions.bycover.band(handle, init = baf2clocs(handle), binsize = 10000L,
  model = c("poisson", "median", "peak"), smooth.k = c(3L, 5L, 15L, 35L,
  55L), alpha = 1, minreg = binsize, keep.bins = TRUE, ...,
  use.threads = lx.use.threads())
```

Arguments

`handle`	baf file handle (as returned by baf.open) (ignored if `bins` is provided)
`init`	clocations regions to start with (default is regions spanning all chromosomes). if `bins` is provided it should be the regions used to compute bins (thru regions.bincover).
`binsize`	size of bins
`model`	one of "median", "poisson" or "peak" or user-defined function. see details.
`smooth.k`	k parameter to lx.smooth.median to smooth bins before computing distribution. (use NULL or 0 to disable smoothing)
`alpha`	width factor (see details).
`minreg`	minimum region size (should be >= binsize)
`keep.bins`	if TRUE, `bins`, binsize and binrange are kept as attributes in the result (set to FALSE to save memory)
`...`	additional parameters to user-defined function if specified
`use.threads`	(see lx.use.threads)

Details

let us call `dist` is the distribution of coverage in all bins of size `binsize`. the band of coverage [a, b] is defined by various models:
model="median": (a,b)=median(dist)-/+alpha*mad(dist)
model="poisson": (a,b)=mode(dist)+/-alpha*sqrt(mode(dist))
where mode(dist) is the coverage value associated to the first maximum of the distribution. this model correspond roughly to a poisson distributed coverage (when coverage is large enough).
model="peak": a=pos-alpha*left; b=pos+alpha*right
where pos, left and right are the maximum peak parameters returned by lx.peaks.
model=function: a and b are defined by a user-provided function called as fun(bins, alpha, ...) that should returns c(a, b)

Value

a nx3 matrix of (1-based) clocations

See Also

regions.bycover.range

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
x <- regions.bycover.band(baf, binsize=1000, smooth.k=3)
x <- regions.bycover.band(baf, binsize=1000, model="median")
y <- attr(x, 'binsize')
baf.close(baf)
```

`regions.bycover.range` *select regions in range of coverage*

Description

loop over given `init` regions. foreach of them split and keep regions with mean coverage in range [mincover, maxcover].

Usage

```
regions.bycover.range(handle, init = baf2clocs(handle), bins = NULL,
  binsize = 10000L, mincover = 0, maxcover = Inf, minreg = binsize,
  keep.bins = TRUE, use.threads = lx.use.threads())
```

Arguments

`handle`	baf file handle (as returned by baf.open) (ignored if `bins` is provided)
`init`	clocations regions to start with (default is regions spanning all chromosomes). if `bins` is provided `init` should be the regions used to compute bins (thru regions.bincover).
`bins`	list (of length `nrow(init)`) of binned coverage in regions (as returned by regions.bincover). if NULL this will be computed using regions.bincover with same parameters. (see notes)
`binsize`	size of bins
`mincover`	minimum coverage (default 0)
`maxcover`	maximum coverage (default +Inf).
`minreg`	minimum region size (should be >= binsize)
`keep.bins`	if TRUE, `bins`, binsize and binrange are kept as attributes in the result (set to FALSE to save memory)
`use.threads`	(see lx.use.threads)

Value

a nx3 matrix of (1-based) clocations

Note

the `bins` != NULL form is provided to avoid recomputation and is mostly used internally (by regions.bycover.band).

See Also

regions.bycover.band

Examples

```
baf <- baf.open(lx.system.file('samples/test.baf', 'xlx'))
x <- regions.bycover.range(baf, binsize=1000, mincover=1)
y <- attr(x, 'binsize')
baf.close(baf)
```

regions.bygc *get regions with specified gc content*

Description

loop over given `init` regions. foreach of them split and keep regions of specified

Usage

```
regions.bygc(handle, init = basta2clocs(handle), winsize = 1000L,
  gcrange = c(0, 1), minreg = winsize, use.threads = lx.use.threads())
```

Arguments

`handle`	basta or baf file handle (as returned by basta.open or baf.open). see details
`init`	regions to start with (default is regions spanning all chromosomes)
`winsize`	window size to compute gc content
`gcrange`	percent gc range (should be in [0, 1])
`minreg`	minimum final region size (should be >= winsize)
`use.threads`	(see lx.use.threads)

Details

there is a slight difference in the way the gc content is computed depending whether you pass a basta or baf file handle.
if a basta file handle is provided then the gc content is computed on the basis of the reference genome.
if a baf file handle is provided then the gc content is computed on the basis of the actual observed alleles (see baf.bin.cloc). note that this may lead to 0 counts in region with no mapping.
basta mode is (about 10 times) quicker than baf mode.

Value

a nx3 matrix of (1-based) clocations

See Also

regions.strata.bygc for a stratified version

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
x <- regions.bygc(fh, winsize=3, gcrange=c(0., 0.5))
basta.close(fh)
```

`regions.exclude` *exclude locations from regions*

Description

remove locations (+/- margin) from regions and keep only regions whose size is >= minreg

Usage

```
regions.exclude(handle, coords, init = basta2clocs(handle), spaceleft = 0L,
  spaceright = spaceleft, minreg = 1L, use.threads = lx.use.threads())
```

Arguments

`handle`	basta or baf file handle (as returned by basta.open or baf.open
`coords`	nx2 matrix of absolute coordinates (1-based) or vector of n absolute point coordinates to remove
`init`	regions to start with (default is regions spanning all chromosomes)
`spaceleft`	size to remove on left side of coords
`spaceright`	size to remove on right side of coords
`minreg`	minimum final region size
`use.threads`	(see lx.use.threads)

Value

a nx3 matrix of (1-based) clocations

`regions.strata.bygc` *stratify regions by gc content*

Description

stratify subregions from `init` regions into gc content

Usage

```
regions.strata.bygc(handle, init = basta2clocs(handle), winsize = 1000L,
  nbins = 5L, minreg = winsize, use.threads = lx.use.threads())
```

Arguments

`handle`	basta or baf file handle (as returned by basta.open or baf.open). see details
`init`	regions to start with (default is regions spanning all chromosomes)
`winsize`	window size to compute gc content
`nbins`	number of %gc bins (bins go from 0. to 1. by 1/nbins)
`minreg`	minimum final region size (should be >= winsize)
`use.threads`	(see lx.use.threads)

Details

there is a slight difference in the way the gc content is computed depending whether you pass a basta or baf file handle.
if a basta file handle is provided then the gc content is computed on the basis of the reference genome.
if a baf file handle is provided then the gc content is computed on the basis of the actual observed alleles (see baf.bin.cloc). note that this may lead to 0 counts in region with no mapping.
basta mode is (about 10 times) quicker than baf mode.

Value

a vector of size nbins. each element is a nx3 matrix of (1-based) clocations

See Also

regions.bygc for a non stratified version

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
x <- regions.strata.bygc(fh, winsize=3)
basta.close(fh)
```

`regions.trim` *trim regions*

Description

remove `trim` on both sides of regions and keep only regions whose size is >= minreg

Usage

```
regions.trim(regions, trim = 1000L, minreg = 10000L)
```

Arguments

`regions`	regions to trim
`trim`	size to remove on both ends
`minreg`	minimum final region size (should be >= trim)

Value

a nx3 matrix of (1-based) clocations

`rev` *Rev method for Dna*

Description

Reverse method for Dna

Usage

```
## S3 method for class 'Dna'
rev(x)
```

Arguments

`x`	Dna object

Value

Dna sequence reversed (but not complemented)

See Also

revcompl

Examples

```
x <- Dna("acgtnry")
rev(x)
```

`revcompl` *Generic method to reverse complement Dna subsequence*

Description

Generic method to reverse complement Dna subsequence

Usage

```
revcompl(obj)
```

Arguments

`obj`	a Dna sequence to reverse complement

See Also

revcompl.Dna, compl

revcompl.Dna *Reverse Complement Dna subsequence*

Description

Reverse Complement Dna subsequence

Usage

```
## S3 method for class 'Dna'
revcompl(obj)
```

Arguments

obj	a Dna sequence to reverse complement

See Also

compl.Dna

Examples

```
x <- Dna("acgtnry")
revcompl(x)
```

runs2clocs *find runs of TRUE's in bitfield*

Description

considering a single bitfield (usually representing allowed positions on a chromosome), this function will recover all runs of TRUE (larger than the given threshold) and return them as a nx3 matrix of clocations (with specified chromosome index `chr`).

Usage

```
runs2clocs(bit, chr = 0, minreg = 1L, p0 = 1L, delta = 1L)
```

Arguments

bit	a bitfield (see package bit)
chr	default chrindex
minreg	minimum number of consecutive TRUE to report (see details)
p0	region origin (see details)
delta	region size factor (see details)

Details

`p0` and `delta` are two parameters to transform indices in bitfileds into actual positions on chromosomes according to:
`pos = p0 + (i-1) * delta`
this is useful when indices actually correspond to binned values (delta=binsize) or to regions that do not start at 1 (p0 = from).
when using delta!=1, the minreg parameter is interpreted with the transformation applied (e.g with delta=1000 and minreg=1000, a single TRUE will actually pass the test)

Value

nx3 matrix of clocations

Note

require library `bit`

See Also

bits2clocs for a list version

Examples

```
b <- bit::as.bit(c(TRUE,FALSE,TRUE,TRUE,TRUE,FALSE,TRUE,TRUE,FALSE))
runs2clocs(b)
runs2clocs(b, minreg=3)
runs2clocs(b, delta=1000, minreg=1000)
runs2clocs(b, delta=1000, minreg=2000)
```

`sloc2cloc` *transform relative slocation to relative clocation*

Description

see HELP.COORD for help on coordinates systems

Usage

```
sloc2cloc(handle, slocation, zero.based.loc = FALSE)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`slocation`	relative slocation ("chrname:from-to")
`zero.based.loc`	given slocation is 0-based

Value

relative clocation (1-based), NULL on error

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
sloc2cloc(fh, "seq1:1-10")
basta.close(fh)
```

---

sloc2coord *transform relative slocation to absolute coordinates*

Description

see HELP.COORD for help on coordinates systems

Usage

```
sloc2coord(handle, slocation, zero.based.loc = FALSE, truncate = TRUE)
```

Arguments

`handle`	basta/baf file handle (as returned by basta.open or baf.open)
`slocation`	relative slocation ("chrname:from-to")
`zero.based.loc`	given slocation is 0-based
`truncate`	truncate 3' to seq.size if needed

Value

absolute coordinates c(absfrom, absto) (1-based), NULL on error

Examples

```
fh <- basta.open(lx.system.file('samples/test.bst', 'xlx'))
sloc2coord(fh, "seq1:1-10")
sloc2coord(fh, "seq2:1-10")
basta.close(fh)
```

---

smooth.kalman *smooth data using Kalman filter*

Description

smooth data using Kalman filter with SSMtrend model

Usage

```
smooth.kalman(x, f = 1, ...)
```

Arguments

`x`	vector of equally spaced data (time series)
`f`	parameter which controls the degree of smoothing (see details)
`...`	other parameters to KFS

Details

the model used is `KFAS::SSMtrend` of degree 1 (local level) and Q (variance) equals to `f` parameter. Namely `KFAS::SSMtrend(1, Q=list(matrix(f)))`

Value

named list with following fields
x : the smoothed values
kfs : the detailled kalman filter results (see KFS)

Note

will force load of package KFAS (if available) because of a nasty bug in KFAS::SSMtrend which prevents from using namespace.

See Also

SSModel, fitSSM, KFS

`smooth.loess` *smooth data using local polynomial regression*

Description

this is an alias of lx.loess

Usage

```
smooth.loess(x, y = NULL, span = 0.75, ...)
```

Arguments

`x`	vector of abscissa if y != NULL or time series values if y == NULL
`y`	vector of values
`span`	parameter which controls the degree of smoothing (see loess)
`...`	other parameters to loess

Details

if just x is provided (i.e. y == NULL) then use x as values and seq_along(x) as abscissa.

Value

named list with following fields
x : the original abscissa (see Details)
y : the smoothed values
loess : raw result from loess

See Also

loess

Examples

```
x <- hist(rnorm(5000), breaks='fd', plot=FALSE)
smooth.loess(x$mids, x$counts)
```

`subseq` *Generic method to extract subsequence*

Description

extract subsequence in range [from, to] (endpoints included)
`to` may be omitted (in which case it equals the length of obj)
`from` and `to` may be negative. they are interpreted as `length - from + 1` or `length - to +1`.

Usage

```
subseq(obj, from, to)
```

Arguments

`obj`	object to extract a subsequence
`from`	start index (1:based, endpoint included)
`to`	end index (1:based, endpoint included)

Note

the term 'subsequence' is a misnommer, this is actually a substring. So this function should be renamed 'substr' or 'substring'.
see [.Dna for an actual subsequence

See Also

subseq.Dna, subseq.character

subseq.character *Extract subsequence from character string*

Description

this is equivalent to substring

Usage

```
## S3 method for class 'character'
subseq(obj, from, to)
```

Arguments

`obj`	object to extract a subsequence
`from`	start index (1:based, endpoint included)
`to`	end index (1:based, endpoint included)

subseq.Dna *Extract Dna subsequence*

Description

see subseq

Usage

```
## S3 method for class 'Dna'
subseq(obj, from, to)
```

Arguments

`obj`	object to extract a subsequence
`from`	start index (1:based, endpoint included)
`to`	end index (1:based, endpoint included)

See Also

[.Dna

Examples

```
x <- Dna("acgtnacgtn")
subseq(x, 1, 3)
subseq(x, 1, -3)
```

summary — *Summary method for Dna*

Description

Make a summary of Dna sequence

Usage

```
## S3 method for class 'Dna'
summary(object, ...)
```

Arguments

`object`	Dna object
`...`	additional arguments affecting the summary produced.

xlx — *eXtended LX library*

Description

Extensions to the LX base library.

These utilities are currently subdivised in different subpackages:

General programming utilities: tbd

Basta format: tbd

Bim format: tbd

Details

Package:	xlx
Type:	Package
Version:	1.0
Date:	2013-12-12
License:	GPL

Author(s)

Alain Viari

`[.Dna` *Subscript extract method for Dna*

Description

extract subscript from Dna sequence

Usage

```
## S3 method for class 'Dna'
obj[index]
```

Arguments

`obj`	Dna object
`index`	any indexing expression (except for negative indices - see details)

Details

negative indices are not interpreted the usual way (i.e. as tail) but as **drop** (like in Python). this is more convenient to delete symbols.

Value

Dna subsequence

See Also

subseq [<-.Dna

Examples

```
x <- Dna("acgtnacgtn")
x[1:5]
x[5:1]
x[seq.int(1, 10, by=2)]
x[-3]
x[-3:-5]
```

`[<-.Dna` *Subscript replace method for Dna*

Description

replace subscript in Dna sequence

Usage

```
## S3 replacement method for class 'Dna'
obj[index] <- value
```

Arguments

`obj`	Dna object to be subscripted
`index`	any indexing expression (except for negative indices - see details)
`value`	a character string (recycled if necessary)

Details

if `value` is shorter than index range, then it is recycled.
if `value` is larger than index range, then it is truncated.

negative indices are not allowed here (since they are interpreted as **drop** see [.Dna)

you cannot currently use this to **insert** symbol within sequence (since value is truncated - I'll improve this in next versions). For the moment, use c instead.

See Also

[.Dna

Examples

```
x <- Dna("acgtnacgtn")
x[1:5] <- 'a'
x[5:1] <- 'cga'
x[seq.int(1, 10, by=2)] <- 'n'
```

Index