Using the rbambools package

Wolfgang Kaisers, CBiBs HHU Dusseldorf

4. Februar 2016

1 Introduction

BAM files are an important and powerful file format in Bioinformatics. This package pursues several objectives:

- Provide a technical (reading and writing) access to BAM files from within R.
- Give an authentic representation of the informational structure inside BAM files as programming interface.
- Provide a fast, C-based access to special (cumulative) aspects of the stored information.

These objectives transform into three implementational layers:

- The samtools C-library (written by Heng Li).
- C-based alignment and align-gap and gap-sites container.
- A R S4 class library.

The samtools library is an adapted version of samtools. Samtools version 0.1.18 (last modified 02 Sept 2011) had been downloaded on 07 Sept 2011 from the samtools homepage [1]. The then current version of the samtools file format description was (v1.4-r985, 0.1.18).

All file interactions are done via samtools. There is C-code which handle alignment data for whole ranges and C-code for accumulation of information about splice-sites from gapped alignments. Samtools and container C-structures are connected to S4 objects by external pointers (EXTPTRSXP)

The R-part of the source code calls C-functions which directly communicate with the samtools library.
Align-gaps are emphasized here because they are highly informative representations of genomic splice-sites in RNA-seq data.

[1] `http://sourceforge.net/projects/samtools/files/samtools/0.1.18/`

2 SAM file format

Data in BAM files is compressed and optionally indexed data in SAM file format. The current definition of the SAM file format [3] can be found on the samtools homepage[2].

BAM files contain sequence alignment data which is the result of potentially incomplete matching sequence snippets to a reference sequence. In practice the snippets are DNA sequences which come from short read sequencing of DNA or RNA extracted from a biological sample and the reference sequence is a genome reference.

Usually one BAM file contains alignments data from one biological sample where the read number is in range of 10 to 100 million. The size of the corresponding compressed files usually is 1 to 15 Gbyte. A very important feature of BAM files is that sorted BAM files can be indexed and indexed files allow random access. This allows very fast access to alignments that are located in arbitrary regions of the reference genome.

The content of BAM files is divided in a header section and an alignment section.

2.1 The header section

The header section contains the following information:

Tag	Description	Explanation
HD	Header line	Format version and sorting
SQ	Reference sequence dictionary	Indexed reference sequences*
RG	Read group	Sequencing technology
PG	Program	Alignment program
CO	Comment	

*Entries in the reference sequence dictionary usually are Chromosomes (e.g. 'chr1')

There are accessor functions in this package for reading and writing the listed fields. The header section is stored and retrieved as binary structure (`bamHeader`) which is converted into a tag delimited string representation (`bamHeaderText`). All processing steps on BAM-header data work on the string representation. `rbamtools`-objects parse and compose strings from and to object slots which then can be accessed via script code.

[2] `http://samtools.sourceforge.net/SAM1.pdf`

2.1.1 The reference sequence dictionary

The reference sequence dictionary section contains a list of reference sequences (usually chromosomes). Only two of six fields (declared in the SAM file format specification) usually only two are used:

Tag	Description
SN	Reference sequence name
LN	Reference sequence length

The reference sequence dictionary section misses an index entry (refid) which is used in alignment structures and is described below in 2.2.1.

2.2 The alignment section

The align section contains a series of alignments datasets. Each alignment describes the coordinates of the identified sequence matches in the reference sequence. The information for each alignment basically consists of:

Field	Content
QNAME	alignment name (read identifier)
RNAME	Reference sequence identifier
POS	Mapping position: 0-based
CIGAR	Matching type string
FLAG	A set of bitwise flags.

2.2.1 The RNAME identifier: refid

Although RNAME associates with a textual entry, usually this field contains a number which identifies a sequence in the header section. To make things complicated, RNAME is a *0-based* sequential identifier which is not explicitly included in the *Reference sequence dictionary* (SQ). So, RNAME=0 means the first SQ entry and the '0' is not present in the header.
We call this missing value *refid* throughout this document and there are functions in this package that automatically generate and use this id. The refid value is used by the `samtools` library as sequence identifier in alignment structures and for defining ranges in index based random access.

2.2.2 Position

The position entry gives the alignment start position. For checking the similarity between query and reference sequence it may be feasible to compare the

sequences manually in single cases.

In order to find the exact matching position it's necessary to notice the base of the position notation. We distinguish *0-based* and *1-based* position notations. They differ by the index of the starting position (and therefore all positions).

The first position in a *0-based* notation is 0 whereas the first position in a *1-based* notation is 1:

0-based	0	1	2
1-based	1	2	3

Both notations appear in samtools. The SAM file format specification says (see [3], section 1.4): 'POS: 1-based leftmost mapping POSition of the first machting base'". Samtools source code comments (bam.h, line 164) state the contrary: 'pos 0-based leftmost coordinate'. Experiences with aligners (tophat 2.0.0) and annotation data (Ensembl and UCSC) indicate that the latter seems to be true (i.e. position entries are 0-based).
In order to reflect the technical file content, two functions (`position` on `bamAlign` objects and `as.data.frame` on `bamRange` objects) return the file contained value (which is 0-based). In order to get values that are congruent with annotation (and IGV genome-browser data) or positions in DNAStringSet (Bioconductor Biostrings package) objects, the position values have to be increased by one.
The `bamGapList` objects which operate on alignment gaps contain *1-based* positions. So, overlapping with annotation data, can be done without correction.

2.2.3 Navigation on reference sequence

Printing the reference sequence results in characters that are ordered from left to right in ascending order of their position coordinate (consistent with ordinary reading succession). We refer to this image when two or more locations are compared. Lower coordinates are assumed to be on the *left* side and higher coordinates are assumed to be on the *right* side.
So, for genes coding on the (+) strand, *left* would be synonymous to *upstream* and *right* would be synonymous to *downstream*.

2.2.4 CIGAR string

Alignment algorithms usually tolerate to some extend inexact matching. The type of matching is described in the CIGAR string. For a complete list of CIGAR operations see [3] 1.4, Nr. 6. The CIGAR string is made up of CIGAR-items. A CIGAR-item consists of a integer number and a character. The number counts

the affected positions (cigar-length). The character describes the type of operation (cigar-type). The following table shows relevant operations:

Operation	Label	Description
M	Match	Exact match of x positions
N	Alignment gap	Next x positions on ref don't match
D	Deletion	Next x positions on ref don't match
I	Insertion	Next x positions on query don't match
		(x = cigar-length)

The operations 'N' and 'D' are mechanistic identical but they describe biological different entities: 'D' means genomic deletions, where few nucleotides on the genome get lost whereas 'N' means gaps which occur in RNA-seq alignments. These gaps are due to DNA-splicing events and their size can achieve magnitude of $10^3 - 10^5$.

First example: The shown alignment is an exact match and will give `position = 2` (0-based!) and `CIGAR = 6M`:

```
AAGTCTAGAA (ref)
  GTCTAG   (query)
```

Second example: We consider an alignment with two nucleotides ("GA") inserted into the reference. The alignment entries will be `position=3` (0-based!) and `CIGAR=3M2I2M`:

```
AAAGTCGATGAA (ref)
   GTC  TG   (query)
```

Third example: The next exampls contains a deletion on the reference. The 'C' in the query sequence has no match. The alignment entries will be `position=3` and `CIGAR=2M1D3M`:

```
AAGT TAGAA (ref)
  GTCTAG   (query)
```

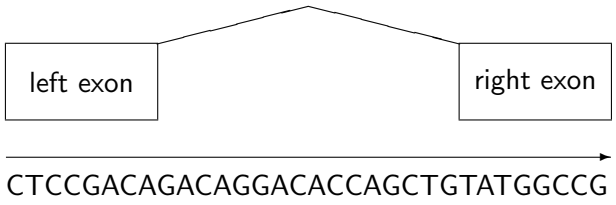
Fourth example: This is a gapped alignment (due to a splicing event in RNA-seq). It will give the entries `position=3` and `CIGAR=3M7N4M`:

```
CCCTACGTCCCAGTCAC (ref)
   TAC       TCAC (query)
```

We see the alignment gap (GTCCCAG). From the 'GT' and 'AG' at the gap boundaries, it can be assumed that this splice-site is on the (+) strand.

2.3 Gapped alignments

A special focus of functionality inside this package are Alignment gaps. Alignment gaps in RNA-seq experiments are viewed as phenomenons that rely on biological splicing mechanisms during protein-biosynthesis and the resulting exon-intron structure of the genome.



2.4 Gap-sites

Gap-sites are alignment gaps (=gap-regions) that are shared by one or more alignments. The nucleotides on the reference sequence that are skipped in the alignment (i.e. the reference region which is depicted by "N"cigar items) form the gap-region.
Gap-sites are also characterized by the fact that they are bordered by M-segments on either side. The amount of information about the existence of gap-site in the alignment is proportional to the number of matching nucleotides that make up the framing M-segments. The calculated derived values on gap-sites therefore center on three measures:

- The number of alignments that define the gap-site.
- The Length of the framing M-segments.
- The number of different length values in the framing M-segments.
- The number of alignments (probes, number of BAM-files) in which the gap-site is found.

Gap-sites are of special interest in RNA-seq experiments because they arise from mRNA sequence which spans a processed splice site (splicing results in removal of intronic sequence ranges from pre-mRNA). Gapped alignments contain highly specific information about splicing events. One of the central objectives in RNA-seq experiments is the identification and quantification of splice events.

In order to describe and illustrate the parameters that are calculated and kept within this package we show the following example:

2.4.1 Example

The following table example shows a short reference sequence and three different alignments that define a gap-site. The reference nucleotides that consitute the gap-region are printed in red:

		qname	position	CIGAR
AG	CCTTGATG	align1	3	2M6N8M
CAG	CCTTGAT	align2	2	3M6N7M
CCAG	CCT	align3	1	4M6N3M
CCCAGGTCCAGCCTTGATGTCC		(reference)	(0-based)	

For each gapped alignments from which the gap-site is constituted, three values concerning the number of matching nucleotides are kept:

- **lcl** (left cigar length) is the length of the left adjacent match in the CIGAR string.
- **rcl** (right cigar length) is the length of the right adjacent match in the CIGAR string.
- **mcl** (minimum cigar length) is the minimum of the `lcl` and `rcl` value for each alignments.

For these parameters we have values in the example:

qname	position	CIGAR	lcl	rcl	mcl
align1	3	2M6N8M	2	8	2
align2	2	3M6N7M	3	7	3
align3	1	4M6N3M	4	3	3

2.4.2 Gap-site coordinates

For each gap-site, `localisation-coordinates` are defines as:

- **refid** (reference sequence identifier)

- **lend** (left-end) is the (1-based) coordinate of the last matching nucleotide on the left side: CCCAGGTCCAGCCTTGATGTCC
- **rstart** (right-start) is the (1-based) coordinate of first matching nucleotide on the right side: CCCAGGTCCAGCCTTGATGTCC

We call alignments sharing identical gap-localisation-coordintates `gap-site-defining-aligns`. In order to derive a lower boundary for the size of the adjacent exons are calculated:

- **lstart** (left-start) is the (1-based) coordinate of the leftmost nucleotide for which a match exsists in the set of left adjacent matching regions: CCCAGGTCCAGCCTTGATGTCC.
  The position is calculated by $lstart = lend - max(lcl) + 1$.
- **rend** (right-end) is the (1-based) coordinate of the rightmost nucleotide for which a match exists in the set of right adjacent matching regions: CCCAGGTCCAGCCTTGATGTCC.
  The position is calculated by $rend = rstart + max(rcl) - 1$.

As derivative, the number of nucleotides in the gap-region (denoted **gaplen**) is calculated as $gaplen = rend - lstart - 1$. Alltogether, the gap-site and the adjacent putative matching regions in this example are:

CCCAGGTCCAGCCTTGATGCCTTGATGTCC.

The associated numeric values for the shown example are:

Name	value	base	
refid	0	0	We assume, there is only one reference sequence
lstart	2	1	Leftmost match position (C)
lend	5	1	Last match on left side (G)
rstart	12	1	First match on right side (C)
rend	20	1	Rightmost match position (G)
gaplen	6		Number of nucleotides in gap

2.4.3 Quantification of alignment numbers

The number of gap-site-defining-aligns are quantified in:

- **nAligns**, the number of alignments that define the gap-site.
- **nProbes**, the number of alignments (BAM-files) in which this gap-site is found.

In the present example, the resulting values are nAligns=3 and nProbes=1.

2.4.4 Quantification of informational support for gap-site's

In order to quantify the information content for each gap-site `lcl` and `mcl` values are stored as single byte values inside of an unsigned long long integer. We define `n` as the number of bytes they contain.

On a 32-bit operating system there is $n = 4$ and on a 64-bit operating system $n = 8$. With that, we van view `lcl` and `mcl` as n-dimensional vectors: $lcl = (lcl_i)_{i=1,\ldots,n}$ and $mcl = (mcl_i)_{i=1,\ldots,n}$ in which values are placed in descending order.

- **nlstart**, the number of different match start positions, which equals the number of different values in the `lcl` vector.
  $nlstart := \#\{lgl_i : i = 1, \ldots, n\}$.
- **lm_sum**, the number of matching nucleotides on the left side of the gap.
  $lm\_sum := \sum_{i=1}^{l} gl_i$.
- **qsm**, the sum of the four largest `mcl` values (quartet sum of minimal cigar length): $\sum_{i=1}^{4} mcl_i$

2.4.5 Gap quality score (gqs)

The gap quality score is calculated as

$$gqs = 10 \times \frac{nlstart}{n} \quad \frac{2qsm}{4} \tag{1}$$

$$= 10 \times \frac{\#\{lgl_i : i = 1, \ldots, n\}}{n} \times \frac{2\sum_{i=1}^{4} mcl_i}{4} \tag{2}$$

The score quantifies number of alignment start positions and matching nucleotides in order to distinguish biological existing splice-sites from alignment phenomenons.

The stored information accumulates with increasing the number of included alignments (BAM-files). The score is given as a positive integer value and the maximum reachable number is 10 * read-length.

The higher the score the more likely is the fact that a gap-site represents a splice-site. gqs is not intended quantify gene expression although the two values correlate.

3 Object types inside rbamtools package

The description of object types in this section starts with reading and writing access to BAM files, proceeds to objects which elementary data inside BAM files and ends with the description of more complex containers.

3.1 Included example files within rbamtools

There are two example files included which are located in the '/inst/extdata' subdirectory of the package installation site. The directory contains a sorted BAM file 'accepted_hits.bam' and the corresponding index file 'accepted_hits.bam.bai'. They were produced (using the `extractRanges` function) from a RNA-seq experiment. A human probe was sequenced using an Illumina Hiseq sequencer. Fastq-reads were aligned with tophat against homo sapiens UCSC reference genome. Complex alignments (i.e. alignments with nCigar > 1) were extracted for genes KLHL17 (chr1) and SNRNP25 (chr16). The BAM file contains 3333 alignments.

3.2 Reading and writing access

Immediate reading and writing access is provided by `bamReader` and `bamWriter` Objects.

3.3 bamReader

An object of class bamReader is constructed and returnd by the function `bamReader` in the following way:

```
> bam <- system.file("extdata",
+                    "accepted_hits.bam", package="rbamtools")
> # Open bam file
> reader <- bamReader(bam)
```

An opened bamReader can be used to access the BAM header section and to read alignments sequenitally. `bamReader` can also be used to sort and index BAM files.
Sorting large BAM files requires some time and produces intermediate files. So the recommended way of sorting large BAM files is to use the samtools command line version. Sorting BAM files within R can be done with:

```
> bamSort(reader, prefix="my_sorted",
+              byName=FALSE, maxmem=1e+9)
```

Sorted BAM files can be indexed. Indexing results in a second file which is usually named as the BAM file itself with an added suffix ".bai". An index file can be created with:

```
> createIndex(reader, idx_filename="index_file_name.bai")
```

Omitting the `idx_filename` argument results in adding the ".baißuffix to the filename of the BAM file which is then automatically located in the same directory as the BAM file itself:

```
> createIndex(reader)
```

The creation of indexes for large BAM files (10 GB) takes some minutes time but can readily be done with this routine and of course has to be done only once per file.

The index files must be loaded before they can be used:

```
> idx <- system.file("extdata", "accepted_hits.bam.bai", package="rbamtools")
> loadIndex(reader, idx)
```

The reader object can be checked for for loaded index with:

```
> indexInitialized(reader)
```

```
[1] TRUE
```

A shortcut for opening a BAM file and loading the *standard* index at the same time is:

```
> reader <- bamReader(bam, idx=TRUE)
```

3.4 Tabled reference sequences: `getRefData`

A data.frame with the reference sequences contained in the BAM header can be obtained with:

```
> getRefData(reader)
```

```
  ID    SN        LN
1  0  chr1 249250621
2  1 chr16  90354753
```

The returned data.frame contains in the first column (ID) the mentioned refid 2.2.1 value which is not part of the header but uses as identifier for alignments and ranges.

3.5 bamWriter

For creation of a `bamWriter` object, a `bamHeader` and a filename must be given. The most convenient way of obaining a `bamHeader` class is to obtain one from an opened `bamReader` object.

```
> header <- getHeader(reader)
> writer <- bamWriter(header,"test.bam")
> # Write alignments using bamSave
> bamClose(writer)
```

alignments can be written to a BAM file either from single instances of `bamAlign`'s or from whole `bamRange` objects.

4 Elementary data structures

The content of BAM files can be divided in `header` section and `alignment` section.

4.1 Structures for header section

The complete header information (in binary representation) can be retrieved from a BAM file with the function `getHeader`. An object of this type is needed for creation of a `bamWriter` object.

In order to get Access to the data itself, the binary data has to be converted into a string representation which is maintained inside an object of class `bamHeaderText`:

```
> header <- getHeader(reader)
> htxt <- getHeaderText(header)
```

The header section is divided into several seqments (as described above) with data tags that describe the origin of the contained alignments. For each segment there is a class which can be be obtained by calling the appropriate function on a `bamHeaderText` object:

Segment	Description	S4 class	Accessor
HD	The header line	headerLine	headerLine
SQ	Reference sequence dictionary	refSeqDict	refSeqDict
RG	Read group		
PG	Program	headerProgram	header Program
CO	Comment		

A complete `bamHeader` object can be created from scratch with the following code:

```
> bh <- new("bamHeaderText")
> headl <- new("headerLine")
> setVal(headl, "SO", "coordinate")
> dict <- new("refSeqDict")
> addSeq(dict, SN="chr1",  LN=249250621)
> addSeq(dict, SN="chr16", LN=90354753)
> dict

An object of class "refSeqDict"
     SN        LN AS M5 SP UR
1  chr1 249250621     0
2 chr16  90354753     0
```

```
> prog <- new("headerProgram")
> setVal(prog, "ID", "TopHat")
> setVal(prog, "PN", "tophat")
> setVal(prog, "CL",
+     "tophat --library-type fr-unstranded hs_ucsc_index reads.fastq")
> setVal(prog, "DS", "Description")
> setVal(prog, "VN", "2.0.0")
> bh <- bamHeaderText(head=headl, dict=dict, prog=prog)
> header <- bamHeader(bh)
```

4.2 Structures for alignment section

Single alignments can be retreaved from opened reader via `getNextAlign`:

```
> align <- getNextAlign(reader)
```

The alignment section in BAM files is a series of alignment (align) records. The data inside of each record is represented by a `bamAlign` object. Section 1.4 [3] describes the information content for each alignments in detail. The fields and the corresponding `bamAlign` accessors are listed below:

Field	Description	Accessor
QNAME	Name	name
FLAG	Multiple Flags	flag
RNAME	refid	2.2.1 refID
POS	Mapping position	2.2.2 position (0-based!)
MAPQ	Mapping quality	mapQuality
CIGAR	CIGAR string	cigarData
	Number of cigar entries	nCigar
RNEXT	Ref name of mate segment	mateRefID
PNEXT	Position of mate segment	matePosition
SEQ	segment sequence	alignSeq
QUAL	Pred-scaled Quality String	alignQual

The accessors can be used in the following way:

```
> name(align)
> flag(align)
> refID(align)
> position(align)
> mapQuality(align)
> cigarData(align)
> nCigar(align)
> mateRefID(align)
```

```
> matePosition(align)
> alignSeq(align)
> alignQual(align)
```

Flag segments: The flag field contains multiple bit-coded flags which are kept together inside an integer value:

Bit	Description	Accessor
0x1	Paired alignments	paired
0x2	Proper pair	properPair
0x4	Unmapped	unmapped
0x8	Mate umapped	mateUnmapped
0x10	Reverse Strand	reverseStrand
0x20	Mate reverse Strand	mateReverseStrand
0x40	First in pair	firstInPair
0x80	Second in pair	secondInPair
0x100	Secondary align	secondaryAlign
0x200	Not passing quality control	failedQC
0x400	PCR or optical duplicate	`pcrORopt_duplicate`

The following code demonstrates the usage of the flag-accessors:

```
> paired(align)
> properPair(align)
> unmapped(align)
> mateUnmapped(align)
> reverseStrand(align)
> mateReverseStrand(align)
> firstInPair(align)
> secondInPair(align)
> secondaryAlign(align)
> failedQC(align)
> pcrORopt_duplicate(align)
```

The same accessors can also be used for setting values:

```
> unmapped(align) <- TRUE
```

4.2.1 Creating bamAlign objects from scratch

The bamAlign function can be used to create `bamAlign` objects from scratch:

```
> align <- bamAlign("HWUSI-0001", "ATGTACGTCG", "Qual/Strng",
+                   "4M10N6M", refid=0, position=100)
> align
```

```
Class       :    bamAlign
refId       :           0
Position    :         100

Cigar Data  :
  Length Type
0      4    M
1     10    N
2      6    M
```

```
> name(align)
```

```
[1] "HWUSI-0001"
```

```
> alignSeq(align)
```

```
[1] "ATGTACGTCG"
```

```
> alignQual(align)
```

```
[1] "Qual/Strng"
```

```
> cigarData(align)
```

```
  Length Type
0      4    M
1     10    N
2      6    M
```

```
> refID(align)
```

```
[1] 0
```

```
> position(align)
```

```
[1] 100
```

The created bamAlign objects can be added to a `bamRange` list or be written to a BAM-file via `bamWriter`.

5 Complex and cumulative container

5.1 alignments lists for specific reference regions: bamRange

`bamRange` objects manage a list of `bamAlign`'s. As BAM files usually contain alignment results against a reference-genome, `bamRange` objects contain list of all alignments that match between a given start and stop position on a given chromosome. Region coordinates are thereby defined by a refid 2.2.1 and a start and stop position.

5.1.1 Reading bamRange from bamReader

In order to create a bamRange object, an index-initialized `bamReader` object and a numeric coordinate-vector of length three are passed to the `bamRange` function.

There are several ways to provide genomic coordinates from which the alignments shall be retrieved. The first way is to specify a circumscribed genomic region (e.g. where a gene of interest is located). The names for the coordinates are not required and only added for explanational purposes:

```
> coords <- c(0, 899000, 900000)
> names(coords) <- c("refid","start","stop")
> range <- bamRange(reader,coords)
> size(range)

[1] 0
```

The second way is to specify coordinates for a whole reference sequence (chromosome). As can be seen from the output of the `getRefData` function, the coordinates for the whole first chromosome should be given as:

```
> getRefData(reader)

  ID    SN        LN
1  0  chr1 249250621
2  1 chr16  90354753

> coords <- c(0,0,249250621)
> names(coords) <- c("refid","start","stop")
> range <- bamRange(reader,coords)
> size(range)

[1] 2216
```

The function `getRefCoords` is used here as shortcut:

```
> coords <- getRefCoords(reader,"chr1")
> coords

[1]         0         0 249250621

> range <- bamRange(reader,coords)
> size(range)

[1] 2216
```

bamRange objects keep a pointer to a current align structure for iteration purposes. Addidionally there are some summarizing values stored (which are displayed by show) which describe the range inside the reference from which the bamRange object was read (seqid, qrBegin, qrEnd, complex) and some statistics (size, qSeqMinLen, qSeqMaxLen). Most of the values are printed by show:

```
> range

Class       :    bamRange
Size        :       2.216
Seqid       :           0
qrBegin     :           0
qrEnd       : 249.250.621
Complex     :           0
rSeqLen(LN) : 249.250.621
qSeqMinLen  :         101
qSeqMaxLen  :         101
Refname     :        chr1

> getCoords(range)

    seqid     begin       end
        0         0 249250621

> getSeqLen(range)

min max
101 101

> getParams(range)

     seqid    qrBegin      qrEnd    complex    rSeqLen
         0          0  249250621          0  249250621
qSeqMinLen qSeqMaxLen
       101        101

> getRefName(range)

[1] "chr1"
```

The (0-based) positions of the leftmost and rightmost matching nucleotides in the align-list are not included by default but can be separately calculated:

```
> getAlignRange(range)

min_pos max_end
     -1   29867
```

5.1.2 Accessing alignments in bamReader

bamReader objects keep a list of bamAlign objects. The objects can sequentially accessed or a data.frame with the alignments data can be retrieved. Therefore bamRange objects internally keep a pointer to the current alignment.

When no current align object is set, the next call to getNextAlign will set the current to the first alignment in list. When the last alignment in list is reached, the next call to getNextAlign will return NULL.

The bamAlign objects in a bamRange container can be sequentially accessed with the getNextAlign function:

```
> align <- getNextAlign(range)
```

After rewinding the bamRange container, the next call to getNextAlign returns the first stored alignment:

```
> rewind(range)
> while(!is.null(align))
+ {
+   # Process align data here
+   align <- getNextAlign(range)
+ }
```

A fast way to get tabled alignment information on bamRange container is to use as.data.frame:

```
> rdf <- as.data.frame(range)
```

5.2 gapList

gapList objects represent a list of align-gaps. They contain one record for single each align-gap present in alignment data. Each align-gap can be linked to a single alignment in the BAM file (via refid and position coordinates).

The function gapList takes an open and indexed instance of bamReader and a set range coordinates (refid,start,stop). The function will scan all alignments that are overlap with the given range in the opened BAM file for gapped alignments.

For every contained align-gap, the refid and the position of the alignment, the match length on both sides (left_cigar_len, right_cigar_len) and the (1-based) positions of the last nucleotide the left side of the gap (left_stop) and the (1-based) position of the first nucleotide on the right side of the gap (right_start).

```
> coords <- getRefCoords(reader, "chr1")
> gl <- gapList(reader, coords)
> gl

An object of class 'gapList'. size: 2297
nAligns: 2216         nAlignGaps: 2297

> dfr <- as.data.frame(gl)
> dfr[1:6, c(1:3, 5:8)]

  refid position left_cigar_len left_stop gaplen
0     0    14729            100     14829    140
1     0    14729            100     14829    140
2     0    14729            100     14829    140
3     0    14729            100     14829    140
4     0    14729            100     14829    140
5     0    14729            100     14829    140
  right_start right_cigar_len
0       14970               1
1       14970               1
2       14970               1
3       14970               1
4       14970               1
5       14970               1
```

The columns 4 and 9 contain the type of the adjacent cigar items (which should always be 'M') are omitted.

The `size` function returns the number of gaps contained in the object. The functions `nAligns` and `nAlignGaps` return the total number of alignments and the number of gapped alignments in the scanned range respectively:

```
> size(gl)
> nAligns(gl)
> nAlignGaps(gl)
```

5.3 gapSiteList

`gapSiteList` objects contain pooled align-gap information. The single gaps are condensed by refid, left-stop and right-start. So each combination of coordinates appears only once in the list. The number of alignments in which each gap has been found is counted into the value `nAligns`.

Two `gapSitList` objects can be merged to one. The basic coordinates of the contained gap-sites (refid, lend, rstart) are compared. Gap-sites with no counterpart are just copied into the new list whereas gap-sites with couterpart are

merged into one record. In this merging process, the core coordinates are just copied. The following table gives an overview over the calculations which are done for merging:

Column name	Site identificator	Resulting value
id		New running index will be created
refid	+	Copied
lstart		Minimum
lend	+	Copied
rstart	+	Copied
rend		Maximum
gaplen		Copied
nAligns		Sum
nProbes		Sum
nlstart		(See text)
`lm_sum`		(See text)
lcl		(See text)
mcl		(See text)

For `lm_sum`, `lcl` and `mcl`, there are specialiced merging operations.

```
> coords <- getRefCoords(reader, "chr1")
> sl <- siteList(reader, coords)

[gap_site_list_fetch] Fetched list of size 32.

> size(sl)

[1] 32

> nAligns(sl)

[1] 2216

> nAlignGaps(sl)

[1] 2297

> sl

An object of class 'gapSiteList'. size: 32
nAligns: 2216         nAlignGaps: 2297

> df <- as.data.frame(sl)
> head(df)
```

```
  id refid lstart  lend rstart  rend gaplen nAligns nProbes
1  1     0  14730 14829  14970 15052    140     553       1
2  2     0  14944 15038  15796 15888    757     201       1
3  3     0  15909 15947  16607 16702    659      29       1
4  4     0  15953 16027  16607 16669    579       4       1
5  5     0  16730 16765  16854 16941     88       5       1
6  6     0  16682 16765  16858 16957     92      34       1
  nlstart lm_sum        lcl       mcl
1       8    772 1633837924 842150450
2       8    601 1163550303 757935406
3       8    196  387456295 387456295
4       4    220  640172875 438445608
5       5    108  236198180 236198180
6       8    358  690630740 690563632
```

5.4 bamGapList

`bamGapList` Objects are designed to contain information about gap-sites for a complete BAM file (i.e. for all refid's). `bamGapList`'s can be merged, so it's possible to cumulate information about gap-sites from a large number of BAM files (e.g. 50). As the whole collection and merging process is done in C, the whole process usually runs with a processing rate $>$ 1.000.000 alignments/sec (on a desktop machine).

```
> bsl <- bamGapList(reader)
> bsl

An object of class 'bamGapList'. size: 39
nAligns: 3.230         nAlignGaps: 3.443

> size(bsl)

[1] 39

> nAligns(bsl)

[1] 3230

> nAlignGaps(bsl)

[1] 3443

> summary(bsl)

  ID    SN        LN start size nAligns nAlignGaps
1  0  chr1 249250621     0   32    2216       2297
2  1 chr16  90354753     0    7    1014       1146
```

```
> dfr <- as.data.frame(bsl)
> head(dfr)
```

```
  id seqid lstart  lend rstart  rend gaplen nAligns nProbes
0  1  chr1  14730 14829  14970 15052    140     553       1
1  2  chr1  14944 15038  15796 15888    757     201       1
2  3  chr1  15909 15947  16607 16702    659      29       1
3  4  chr1  15953 16027  16607 16669    579       4       1
4  5  chr1  16730 16765  16854 16941     88       5       1
5  6  chr1  16682 16765  16858 16957     92      34       1
  nlstart qsm nmcl  gqs
0       8 200    8 1000
1       8 181    8  905
2       8 115    8  575
3       4 138    4  345
4       5  95    5  296
5       8 172    8  860
```

5.4.1 readPooledBamGaps

The two functions

- readPooledBamGaps
- readPooledBamGapDf

provide functionality for extraction and quantification of alignment gap positions over multiple BAM files.

```
> bam<-system.file("extdata","accepted_hits.bam",package="rbamtools")
> rpb<-readPooledBamGaps(bam)
> rpdf<-readPooledBamGapDf(bam)
```

The important part of the resulting information is shown below as example. The boundaries of the covered alignment ranges are denoted *lstart* and *lend* for the left side and *rstart* and *rend* for the right side of the alignmnent gap. Additionally, the number of alignments (nAligns) and the number of samples (nProbes) which provide support for the alignment gap position are given.

```
> xtable(head(rpdf)[, c(1:6, 8, 9, 13)])
```

Figure 1, although only contains data on 40 gap sites, already shows the *gqs* distribution at larger samples sizes: There are many sites with high and low gqs values and the intermediate range is only sparsely filled.

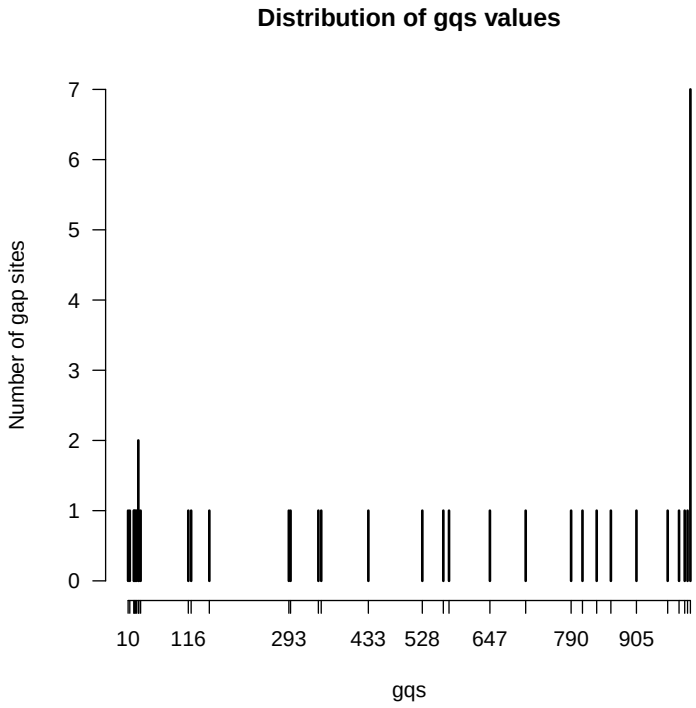


Abbildung 1: Distribution of gqs values

	id	seqid	lstart	lend	rstart	rend	nAligns	nProbes	gqs
0	1	chr1	14730	14829	14970	15052	553	1	1000
1	2	chr1	14944	15038	15796	15888	201	1	905
2	3	chr1	15909	15947	16607	16702	29	1	575
3	4	chr1	15953	16027	16607	16669	4	1	345
4	5	chr1	16730	16765	16854	16941	5	1	296
5	6	chr1	16682	16765	16858	16957	34	1	860

6 Alternative approaches for analyis of splicing events

6.1 Bioconductor alternative to 'readPooledBamGaps'

Using Bioconductor alignment numbers on gap sites as well as number of samples in which gap sites are found can be counted. We show an example implementation. The function can be evoked by simply providing a vector of BAM file names (but assumes that associated *.bam.bai BAM index files are present).

```
> scanGapSites <- function(bam, yieldSize=1e6, mc.cores=2)
+ {
+     require(Rsamtools)
+     require(GenomicAlignments)
+
+     mc.cores <- as.integer(mc.cores)
+
+     # Function will be called by mclapply
+     doScanBam <- function(bamFile)
+     {
+         open(bamFile)
+
+         # Create empty container.
+         gPos <- GRanges()
+
+         # Fill container by processing 'yieldSize' reads at a time
+         while(
+             sum(
+                 elementLengths(
+                     records <- scanBam(
+                                 bamFile,
+                                 param=ScanBamParam(
+                                     flag=scanBamFlag(isUnmappedQuery=FALSE),
+                                     what=scanBamWhat()[c(3, 5, 8)]
+                                     )
+                                 )[[1]]
+                     )
```

```
+             ) > 0
+         ){
+
+             nOps <- cigarRangesAlongReferenceSpace(records$cigar,ops="N")
+             sel <- elementLengths(nOps) != 0
+             gPos <- c(gPos,
+                         GRanges(seqnames=rep(records$rname[sel],
+                                              elementLengths(nOps)[sel]),
+                                              ranges=unlist(shift(nOps[sel],
+                                              records$pos[sel]))
+                                 )
+                     )
+         }
+         close(bamFile)
+         # Return all gap positions
+         return(gPos)
+     }
+
+     cat("[scanGapSites] Processing", length(bam), "Files.\n")
+
+
+     bamFileList <- BamFileList(bam, yieldSize = yieldSize)
+     gList <- mclapply(bamFileList, doScanBam)
+
+     sz <- object.size(gList)
+     bm<-Sys.localeconv()[7]
+     cat("[scanGapSites] Collected object of size",
+             format(as.numeric(object.size(gList)), big.mark=bm),
+             "bytes.\n")
+
+     # - - - - - - - - - - - - - - - - - - - - - - - - - - - #
+     # Get all unique positions across all samples
+     # - - - - - - - - - - - - - - - - - - - - - - - - - - - #
+     uPos <- unique(Reduce("c", gList))
+
+     # - - - - - - - - - - - - - - - - - - - - - - - - - - - #
+     # Create the count table by
+     # transforming the ranges into character strings.
+     # - - - - - - - - - - - - - - - - - - - - - - - - - - - #
+     ref <- paste(seqnames(uPos), start(uPos), end(uPos), sep="-")
+
+     # Will be called by mclapply
+     doTable <- function(grng, ref)
+     {
```

```
+         tab <- table(paste(seqnames(grng), start(grng), end(grng),sep="-"))
+         tab[match(ref,names(tab))]
+     }
+
+     count.table <- do.call("cbind",
+                     mclapply(gList, doTable, ref, mc.cores=mc.cores))
+     rownames(count.table) <- ref
+
+     cat("[scanGapSites] Number of sites:",
+             format(nrow(count.table), big.mark=bm),
+             ".\n")
+
+     cat("[scanGapSites] Finished.\n")
+     return(count.table)
+ }
```

The *scanGapSites* function offers speed enhancement by parallel processing and by that may reach similar processing times as the *readPooledBamGapDf* function. While multiple chunks of alignment data are loaded into the working memory demand will be in the range of some Gigabyte (depending on yieldSize).
This function does not provide information on the size of the exonic range which is covered by the alignment. Also, a quality value similar to the *gqs* can not be calculated retrospectively.

6.2 Direct import of Junction positions from Aligner

The RNA-seq aligners TopHat and STAR provide information on identified splice sites in their output files. The Bioconductor *GenomicAlignments* package provides functions for direct import of this information.

6.2.1 STAR aligner

The STAR aligner [2] produces a tab separated file where positions of alignment gaps as well as number of uniquely and multi-mapping reads and a code for the intron motif is produced (SJ.out.tab file).
The content of these files can directly be imported into R usign the *readSTARJunctions* method.

6.2.2 TopHat aligner

TopHat [4] produces a *junctions.bed* file where each junction is represented in form of two connected BED blocks. The content of the *junctions.bed* file can be read using the *readTopHatJunctions* function.

Both tables can be used to obtain candidate locations for splice sites which also allows fast collection. For further validation of putative splicing events, additional information may be necessary, e.g. more than just two intronic nucleotide positions or qualification of the alignment gap postion by the gqs.

7 Miscellaneous functions

7.1 bamCount and bamCountAll

The bamCount counts alignments and CIGAR-items in alignment ranges defined by coordinates. The function returns a named integer vector of length 10.

The bamCountAll counts alignments and CIGAR-items for whole BAM-files (represented by a `bamReader`). The function optionally takes a `verbose` argument which controls the textual output during runtime. The function returns a `data.frame`. Each line contains counts for one reference sequence, each column contains data for one CIGAR-OP type. Columns with total counts, referene sequence id (ID) and reference sequence length (LN) are added.

```
> coords <- c(0, 0, 14730)
> count <- bamCount(reader, coords)
> xtable(matrix(count, nrow=1))
```

	1	2	3	4	5	6	7	8	9	10
1	30	0	2	13	0	0	0	0	0	15

```
> count <- bamCountAll(reader, verbose=TRUE)

[bamCountAll] Counting chr1         [ 1/2]
[bamCountAll] Counting chr16         [ 2/2]
[bamCountAll] Finished.
```

	M	I	D	N	S	H	P	=	X	nAligns	ID	LN
chr1	4577	18	46	2297	0	0	0	0	0	2216	0	249250621
chr16	2164	4	0	1146	0	0	0	0	0	1014	1	90354753

7.2 countNucs

The countNucs counts occurrence of the nucleotides ACGT in `bamAlign` and `bamRange` objects. An integer vector of length 4 is returned. The names give the nucleotide which is counted at each position. The syntax is identical for `bamAlign`

```
> align <- bamAlign("HWUSI-0001", "ACCGGGTTTT","Qual/Strng",
+                                "4M10N6M", refid=0, position=100)
> countNucs(align)

A C G T N
1 2 3 4 0
```

and bamRange

```
> reader <- bamReader(bam, idx=TRUE)
> coords <- c(0, 0, 14730)
> range <- bamRange(reader, coords)
> countNucs(range)

  A   C   G   T   N
237 490 533 255   0
```

objects.

7.3 nucStats

nucStats for bamReader The nucStats function counts occurrence of the nucleotides ACGT in whole BAM files via opened `bamReader` objects. Any other character values are subsumed in the value N. The last two columns contain values for GC content and AG/GC ratio.

The function returns a `data.frame` with one row for each reference sequence which is listed in the BAM-header section.

```
> ncs <- nucStats(reader)
```

	nAligns	A	C	G	T	N	gcc	at_gc_ratio
chr1	2216	37756	72232	61721	52102	5	0.58	0.71
chr16	1014	28090	25298	31102	17921	3	0.58	0.71

nucStats for BAM file names The nucStats function counts occurrence of the the nucleotides ACGT for a given list of BAM file names. The last two columns contain values for GC content and AG/GC ratio. The function returns a `data.frame` with one row for each given BAM file name.

```
> ncs <- nucStats(bam)
```

	nAligns	A	C	G	T	N	gcc	at_gc_ratio
1	3230	65846	97530	92823	70023	8	0.58	0.71

7.4 createIdxBatch

The createIdxBatch is intended to create index files for a multiple BAM-files. The names of the created BAM-index files can optionally be added. The standard name for BAM-index files is the name of the BAM file plus an added suffix ".bai".

The third (optional) argument is `rebuild`. When `rebuild` is `FALSE` the function will only create not already existing BAM-index files. When `rebuild` is `TRUE` the function will build BAM-index for all given BAM-files.

Sometimes (especially when BAM-files have been copied), they content may be corrupt. Rebuilding index files is a way to check the integrity of a BAM-file.

```
> createIdxBatch(bam)
```

7.5 readerToFastq, rangeToFastq

readerToFastq: The readerToFastq and rangeToFastq take (optionally random subsets) of whole BAM-files (via `bamReader`) or selected ranges (via `bamRange`) and copy alignments to fastq files.

For handling of alignments inside whole BAM-files, use the readerToFastq function. Alignments are read from BAM files via getNextAlign. For an opened file, there is a pointer to the last retrieved alignment kept. Multiple calls to getNextAlign will retrieve subsequent alignments.

When a logical vector is given, there will be a call to getNextAlign for every entry in the vector. The function then returns the number of checked alignments.

When EOF is reached before the vector is processed, the number of checked alignments is smaller than the length of the given logical vector. When no logical vector is given, the function returns the number of written alignments.

```
> reader <- bamReader(bam)
> readerToFastq(reader, "out.fastq")
> bamClose(reader)
> # Reopen in order to point to first alignment
> reader <- bamReader(bam)
> index <- sample(1:100, 20)
> readerToFastq(reader, "out_subset.fastq", which=index)
```

rangeToFastq: The function rangeToFastq writes all alignments in a `bamRange` object into a compressed fastq file. Optionally, a logical vector (where length must be equal to size of range) can be given. In this case only the depicted alignments are copied into the fastq file and the remaining alings are skipped.

```
> reader <- bamReader(bam, idx=TRUE)
> coords <- as.integer(c(0,0,249250621))
> range <- bamRange(reader,coords)
> rangeToFastq(range,"rg.fq.gz")
> index <- sample(1:size(range),100)
> rangeToFastq(range,"rg_subset.fq.gz",which=index)
```

7.6 Functions for reading and displaying Phred quality scores

Phred quality scores Q are defindes as $Q = -10log_{10}P$ where P is the base calling error probability.

getQualDf takes a `bamReader` and returns a data.frame. The `data.frame` has 94 rows which represent values from 0 to 93 ([1]). The number of columns equals the maximum sequence length in the given `bamRange`.

```
> qdf <- getQualDf(range)
> qdf[32:38,1:10]
```

```
   1 2 3 4 5 6 7 8 9 10
31 2 2 1 0 1 0 0 1 1  1
32 0 2 0 3 0 2 0 0 0  0
33 0 0 1 1 3 0 2 1 1  1
34 1 3 1 0 0 0 0 0 0  0
35 0 7 7 7 7 8 7 7 7  6
36 0 0 0 0 0 0 0 0 0  0
37 0 0 0 2 2 2 2 3 0  1
```

```
> qdr <- getQualDf(range,prob=TRUE)
> qrr <- round(qdr,2)
> qrr[32:38,1:10]
```

```
      1    2    3    4    5    6    7    8    9   10
32 0.13 0.13 0.07 0.00 0.07 0.00 0.00 0.07 0.07 0.07
33 0.00 0.13 0.00 0.20 0.00 0.13 0.00 0.00 0.00 0.00
34 0.00 0.00 0.07 0.07 0.20 0.00 0.13 0.07 0.07 0.07
35 0.07 0.20 0.07 0.00 0.00 0.00 0.00 0.00 0.00 0.00
36 0.00 0.47 0.47 0.47 0.47 0.53 0.47 0.47 0.47 0.40
37 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
38 0.00 0.00 0.00 0.13 0.13 0.13 0.13 0.20 0.00 0.07
```

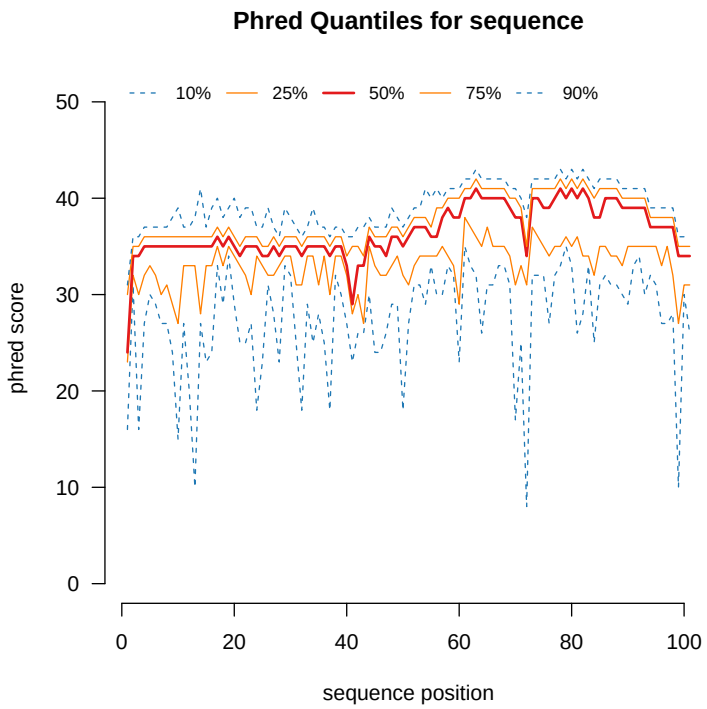
getQualQuantiles takes a `bamReader` and a vector of quantiles (must be between 0 and 1) and returns a data.frame. The data.frame contains one row for each quantile and also as many columns as the maximum sequence length.

```
> qt <- getQualQuantiles(range,c(0.25,0.5,0.75))
> qt[,1:10]
```

```
      1  2  3  4  5  6  7  8  9 10
q_25 23 32 30 32 33 32 30 31 29 27
q_50 24 34 34 35 35 35 35 35 35 35
q_75 30 35 35 36 36 36 36 36 36 36
```

plotQualQuant takes a `bamReader` and plots the 0.1, 0.25, 0.5, 0.75 and 0.9 quantiles for all occurring sequence positions.

```
> plotQualQuant(range)
```



7.7 Functions for calculation and displaying alignment-depth

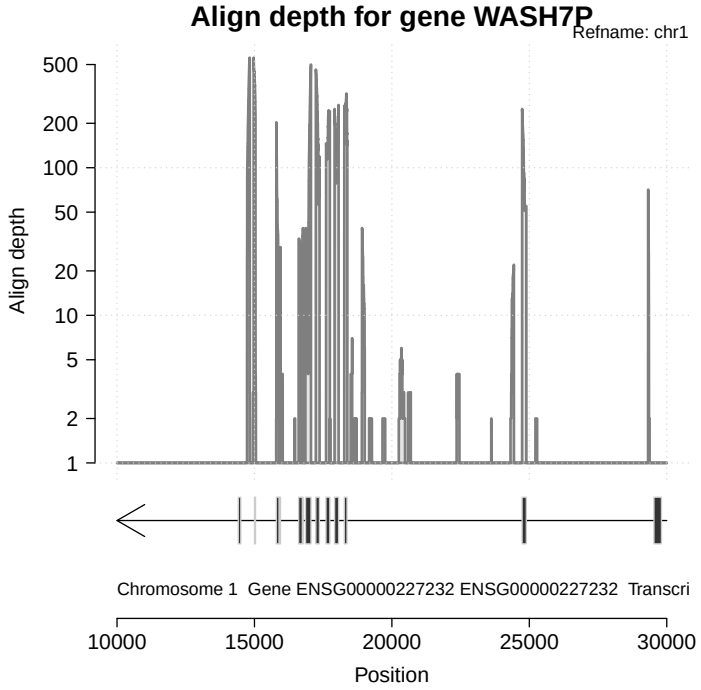
Alignment depth means quantification of present matches for each nucleotide position in a given range.

The alignDepth member function calculates alignment depth for a given bamRange object. From the `bamRange` object, the range is extracted and for each nucleotide position whithin this range the numbers of alignment matches are calculated. When `alignDepth` is called wich `gap=TRUE`, the function counts alignments solely for gap-adjacent match regions (cigar-op's).

Whe extract a `bamRange` for the WASH7:

```
> # WASH7P coordinates
> xlim <- c(10000, 30000)
> coords <- c(0,xlim[1], xlim[2])
> range <- bamRange(reader, coords)
> bamClose(reader)
> ad <- alignDepth(range)
> ad

Class       :  alignDepth
Seqid       :           0
qrBegin     :       10.000
qrEnd       :       30.000
Complex     :           0
rSeqLen(LN) :   249.250.621
qSeqMinLen  :         101
qSeqMaxLen  :         101
refname     :        chr1
10001 10002 10003 10004 10005 10006
    0     0     0     0     0     0

> getParams(ad)

     seqid    qrBegin      qrEnd    complex    rSeqLen
         0      10000      30000          0  249250621
qSeqMinLen qSeqMaxLen        gap
       101        101          0

> # Identifier
> gene <- "WASH7P"
> ensg_id <- "ENSG00000227232"
> enst_id <- "ENST00000538476"
> # Get exon positions
> start <- c(14411, 15000, 15796, 15904, 16607, 16748, 16858, 17233,
+                     17602, 17915, 18268, 24737, 29534)
> end <-   c(14502, 15038, 15901, 15947, 16745, 16765, 17055, 17364,
+                     17742, 18061, 18366, 24891, 29806)
```

```
> plotAlignDepth(ad, lwd = 2, xlim = xlim,
+              main = paste("Align depth for gene",gene),
+              ylab = "Align depth", start = start,
+              end = end, strand = "-",
+              transcript = paste("Chromosome 1",
+                  "\tGene ENSG00000227232", ensg_id,
+                  "\tTranscript ",enst_id
+ ))
```



7.8 Functions for counting alignmnents in genomic segments

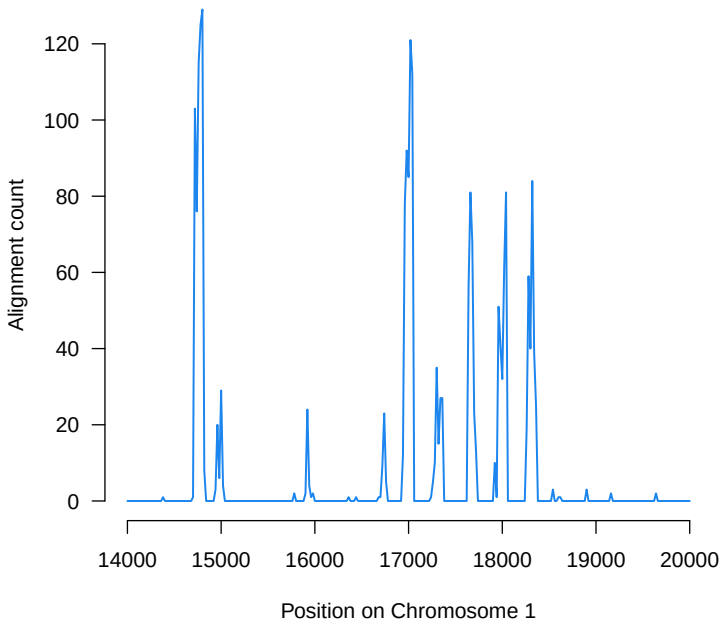
The `rangeSegCount` class counts alignment numbers in specified genomic segments. The `rangeSegCount` function takes three arguements:

- reader: An opened instance of bamReader which must contain an initialized bam index.
- coords: A numeric vector of length 3 which contains (seqid, start, end) as specified for bamRange object. As for bamRange's, this defines the genetic region from which alignments are read from a BAM file.
- segments: A numeric vector of arbitrary length which should divide the range defined by `coords` into segments. In the segments vector, two adja-

cent values define a right open interval in which the genomic alignments are counted.

The function only considers the alignment start postions. This reflects the difference to the alignment depth construction where aligned positions are counted separately for each nucleotide.

```
> # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
> # B) Count range segment
> # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - #
> reader <- bamReader(bam, idx=TRUE)
> coords <- c(0, 0, 2e4)
> segments <- seq(14000, 20000, 20)
> segcount<-rangeSegCount(reader, coords, segments)
> segcount

An object of class 'rangeSegCount'.
Refname :            chr1
Seqid   :               0
LN      :     249.250.621
qrBegin :               0
qrEnd   :          20.000
Complex :           FALSE
Size    :             301

  position count
1    14000     0
2    14020     0
3    14040     0
4    14060     0
5    14080     0
6    14100     0

> dfr<-as.data.frame(segcount)
> sum(dfr$count)

[1] 2112

>

> plot(count~position, dfr, type="l",
+         las=1, bty="n", lwd=1.5, col="dodgerblue2",
+         xlab="Position on Chromosome 1",
+         ylab="Alignment count",
+         main="Number of alignments in genomic segments of 20 nucleotides size")
```



8 Misalign errors

8.1 Cause of misalign errors

In the samtools C library, alignment data is contained in `bam1_t` structures. Cigar data is stored in the `data` segment of `bam1_t` structures which has type `unsigned char *` (1 byte). The data data segments usually are accessed using pointers to `unsigned char` as shown below.

```
bam1_t *align = bam_init1();
unsigned char * data = align->data;
```

Cigar data has type `unsigned int` (4 bytes) which differs from the type size of `unsigned char`. In order to retrieve cigar data from an alignment structure a type change has to be performed. On pointers, these type changes are called pointer casts. The accessor for cigar data `bam1_cigar` performs an implicit pointer cast in the following statment:

```
unsigned int *cigar=align->data + align->core.l_qname;
```

Because there are usually multiple cigar items for each alignment, a cigar structure actually contains more than one value. Therefore the value pointed to is an array rather than a single value. Values inside a C array are accessed via the

index operator '[x]' (similar to R). The first value inside a cigar array is accessed using

```
unsigned int c = cigar[0];
```

(unlike R where the first element has index 1). In other words, the index operator k[x] refers to a memory location x-units away from the base address k. The unit size is the memory demand of one element of the stored type. By putting the example together we see that after

```
bam1_t *align = bam_init1();
unsigned char * data = align->data + align->core.l_qname;
unsigned int *cigar = data;
```

the two expressions

```
data[4];
cigar[4];
```

point to different memory locations due to the implicit conversion during the assignment from data to cigar.
For this ambiguity, there is no general rule defined in the C language definition, a so called **undefined behaviour** (UB) situation. The result may depend on implementation and may vary between different operating systems for example between Debian Linux and SPARC.
The accountable structures currently are present in all C implementations of samtools, for example in the current (Bioconductor) Rsamtools version[3] and in the current htslib version (0.2.0-rc8) on GitHub[4].

8.2 Correction of misalign errors

In rbamtools, we have corrected these misalign errors by a new `cigar` data member into `bam1_t` structures and maintaining a copy of cigar data there. The implementation of this workaround requires changes in five functions: The creator (`bam_init1`) and destructor (`bam_destroy1`), the functions for copying (`bam_copy1`) and duplication (`bam_dup1`) and the accessor for cigar data (`bam1_cigar`). Additionally, the samtools library has to be corrected at places where the own creator and destructor routines had not been deployed (e.g. in the `bam_sort_core_ext` function in bam_sort.c and in the `bam_index_core` function in bam_index.c). This workaround is associated with a slight speed decrease of 6.7 % but it does not change file access and the programming interface. It can be introduced into any current samtools version.

[3] http://www.bioconductor.org/packages/release/bioc/html/Rsamtools.html
[4] https://github.com/samtools/samtools

Literatur

[1] PJA Cock, CJ Fields, N Goto, ML Heuer, and Rice PM. The sanger fastq file format for sequences with quality scores and the solexa/illumina fastq variants. *Nucleic Acids Research*, 38:1767–1771, 2010.

[2] A. Dobin, C. A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T. R. Gingeras. Star: ultrafast universal rna-seq aligner. *Bioinformatics*, 29(1):15–21, Jan 2013.

[3] The SAM Format Specication Working Group. The sam format specication (v1.4-r985). `http://samtools.sourceforge.net/SAM1.pdf`.

[4] D. Kim, G. Pertea, C. Trapnell, H. Pimentel, R. Kelley, and S. L. Salzberg. Tophat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biol.*, 14(4):R36, 2013.