plyranges: A grammar of genomic data transformation

Stuart Lee [*][†]

Dianne Cook [*]

Michael Lawrence [‡][§]

The Bioconductor project provides many interoperable data abstractions for analyzing high-throughput genomics experiments; however implementing a typical genomic workflow with Bioconductor requires learning these abstractions and understanding them at an integrative level. This places a large cognitive burden on the user, especially for non-programmers. To reduce this burden we have created a grammar of genomic data transformation that operates on a single, central Bioconductor data structure, GRanges, which naturally represents genomic intervals and their associated measurements. The grammar defines verbs for performing actions on and between genomic interval data through a simplified, coherent interface to existing Bioconductor infrastructure, resulting in fluent analysis workflows. We have implemented this grammar as an R/Bioconductor package called `plyranges`.

Keywords: Bioconductor, Grammar, Genomes, Data Analysis

Background

High-throughput genomics promises to unlock new disease therapies, and strengthen our knowledge of basic biology. To deliver on those promises, scientists must derive a stream of knowledge from a deluge of data. Genomic data is challenging in both scale and complexity. Innovations in sequencing technology often outstrip our capacity to process the output. Beyond their common association with genomic coordinates, genomic data are heterogeneous, consisting of raw sequence read alignments, genomic feature annotations like genes and exons, and summaries like coverage vectors, ChIP-seq peak calls, variant calls, and per-feature read counts. Genomic scientists need software tools to wrangle the different types of data, process the data at scale, test hypotheses, and generate new ones, all while focusing on the biology, not the computation. For the tool developer, the challenge is to define ways to model and operate on the data that align with the mental model of scientists, and to provide an implementation that scales with their ambition.

Several domain specific languages (DSLs) enable scientists to process and reason about heterogeneous genomics data by expressing common operations, such as range manipulation and overlap-based joins, using the vocabulary of genomics. Their imple-

[*] Department of Econometrics and Business Statistics, Monash University, Clayton, Australia

[†] Molecular Medicine Division, Walter and Eliza Hall Institute, Parkville, Australia

[‡] Bioinformatics and Computational Biology, Genentech Research and Early Development, South San Francisco, United States of America

[§] corresponding author

mentations either delegate computations to a database, or operate over collections of files in standard formats like BED. An example of the former is the Genome Query Language (GQL) and its distributed implementation GenAp which use a SQL-like syntax for fast retrieval of information of unprocessed sequencing data [1, 2]. Similarly, the Genometric Query Language (GMQL) implements a DSL for combining genomic datasets [3]. The command line application BEDtools develops an extensive algebra for performing arithmetic between two or more sets of genomic regions [4]. All of the aforementioned DSLs are designed to be evaluated either at the command line or embedded in scripts for batch processing. They exist in a sparse ecosystem, mostly consisting of UNIX and database tools that lack biological semantics and operate at the level of files and database tables.

The Bioconductor/R packages `IRanges` and `GenomicRanges` [5, 6, 7] define a DSL for analyzing genomics data with R, an interactive data analysis environment that encourages reproducibility and provides high-level abstractions for manipulating, modelling and plotting data, through state of the art methods in statistical computing. The packages define object-oriented (OO) abstractions for representing genomic data and enable interoperability by allowing users and developers to use these abstractions in their own code and packages. Other genomic DSLs that are embedded in programming languages include pybedtools and valr [8, 9], however these packages lack the interoperability provided by the aforementioned Bioconductor packages and are not easily extended.

The Bioconductor infrastructure models the genomic data and operations from the perspective of the power user, one who understands and wants to take advantage of the subtle differences in data types. This design has enabled the development of sophisticated tools, as evidenced by the hundreds of packages depending on the framework. Unfortunately, the myriad of data structures have overlapping purposes and important but obscure differences in behavior that often confuse the typical end user.

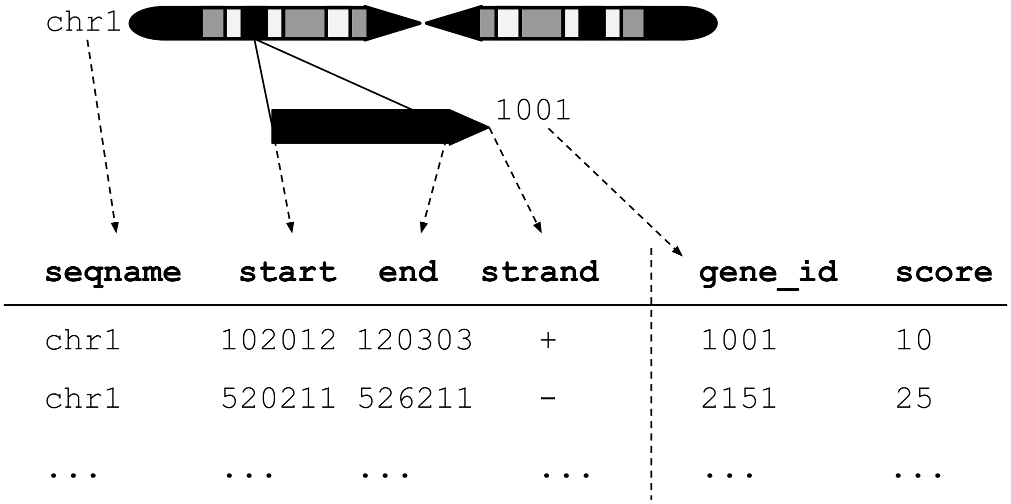
Recently, there has been a concerted, community effort to standardize R data structures and workflows around the notion of tidy data [10]. A tidy dataset is defined as a tabular data structure that has observations as rows and columns as variables, and all measurements pertain to a single observational unit. The tidy data pattern is useful because it allows us to see how the data relate to the design of an experiment and the variables measured. The `dplyr` package [11] defines an application programming interface (API) that maps notions from the general relational algebra to verbs that act on tidy data. These verbs can be composed together on one or more tidy datasets with the pipe operator from the `magrittr` package [12]. Taken together these features enable a user to write human readable analysis workflows.

We have created a genomic DSL called `plyranges` that reformulates notions from existing genomic algebras and embeds them in R as a genomic extension of `dplyr`. By analogy, `plyranges` is to the genomic algebra, as `dplyr` is to the relational algebra. The `plyranges` Bioconductor package implements the language on top of a key subset of Bioconductor data structures and thus fully integrates with the Bioconductor framework, gaining access to its scalable data representations and sophisticated statistical methods.

Results

Genomic Relational Algebra

Data Model



seqname	start	end	strand	gene_id	score
chr1	102012	120303	+	1001	10
chr1	520211	526211	-	2151	25
...	...	...	...	...	...

Figure 1: An illustration of the GRanges data model for a sample from an RNA-seq experiment. The core components of the data model include a seqname column (representing the chromosome), a ranges column which consists of start and end coordinates for a genomic region, and a strand identifier (either positive, negative, or unstranded). Metadata are included as columns to the right of the dotted line as annotations (gene_id) or range level covariates (score).

The `plyranges` DSL is built on the core Bioconductor data structure GRanges, which is a constrained table, with fixed columns for the chromosome, start and end coordinates, and the strand, along with an arbitrary set of additional columns, consisting of measurements or metadata specific to the data type or experiment (figure 1). GRanges balances flexibility with formal constraints, so that it is applicable to virtually any genomic workflow, while also being semantically rich enough to support high-level operations on genomic ranges. As a core data structure, GRanges enables interoperability between `plyranges` and the rest of Bioconductor. Adhering to a single data structure simplifies the API and makes it easier to learn and understand, in part because operations become endomorphic, i.e., they return the same type as their input.

GRanges follow the intuitive tidy data pattern: it is a rectangular table corresponding to a single biological context. Each row contains a single observation and each column is a variable describing the observations. GRanges specializes the tidy pattern in that the observations always pertain to some genomic feature, but it largely remains compatible with the general relational operations defined by `dplyr`. Thus, we define our algebra as an extension of the `dplyr` algebra, and borrow its syntax conventions and design principles.

	Verb	Description
Aggregate	***summarize()***	aggregate over column(s)
	disjoin_ranges()	aggregate column(s) over the union of end coordinates
	reduce_ranges()	aggregate column(s) by merging overlapping and neighboring ranges
Modify (Unary)	***mutate()***	modifies any column
	select()	select columns
	arrange()	sort by columns
	stretch()	extend range by fixed amount
	shift_(direction)	shift coordinates
	flank_(direction)	generate flanking regions
Modify (Binary)	*%intersection%*	row-wise intersection
	%union%	row-wise union
	compute_coverage	coverage over all ranges
	%setdiff%	row-wise set difference
	between()	row-wise gap range
	span()	row-wise spanning range
Merge	*join_overlap_*()*	merge by overlapping ranges
	join_nearest	merge by nearest neighbor ranges
	join_follow	merge by following ranges
	join_precedes	merge by preceding ranges
	union_ranges	range-wise union
	intersect_ranges	range-wise intersect
	setdiff_ranges	range-wise set difference
	complement_ranges	range-wise union
Operate	*anchor_direction()*	fix coordinates at direction
	group_by()	partition by column(s)
	group_by_overlaps()	partition by overlaps
Restrict	***filter()***	subset rows
	filter_by_overlaps()	subset by overlap
	filter_by_non_overlaps()	subset by no overlap

Table 1: Overview of the `plyranges` grammar. The core verbs are briefly described and categorized into one of the following higher level categories: aggregate, modify, merge, operate, or restrict. A verb is given bold text if its origin is from the `dplyr` grammar.

Algebraic operations

The plyranges DSL defines an expressive algebra for performing genomic operations with and between GRanges objects (see table 1). The grammar includes several classes of operation that cover most use cases in genomics data analysis. There are range arithmetic operators, such as for resizing ranges or finding their intersection, and operators for merging, filtering and aggregating by range-specific notions like overlap and proximity.

Arithmetic operations transform range coordinates, as defined by their *start*, *end* and *width*. The three dimensions are mutually dependent and partially redundant, so direct manipulation of them is problematic. For example, changing the *width* column needs to change either the *start*, *end* or both to preserve integrity of the object. We introduce the *anchor* modifier to disambiguate these adjustments. Supported anchor points include the start, end and midpoint, as well as the 3′ and 5′ ends for strand-directed ranges. For example, if we anchor the start, then setting the width will adjust the end while leaving the start stationary.

The algebra also defines conveniences for relative coordinate adjustments: *shift* (unanchored adjustment to both start and end) and *stretch* (anchored adjustment of width). We can perform any relative adjustment by some combination of those two operations. The *stretch* operation requires an anchor and assumes the midpoint by default. Since *shift* is unanchored, the user specifies a suffix for indicating the direction: left/right or, for stranded features, upstream/downstream. For example, *shift_right* shifts a range to the right.

The *flank* operation generates new ranges that are adjacent to existing ones. This is useful, for example, when generating upstream promoter regions for genes. Analogous to *shift*, a suffix indicates the side of the input range to flank.

As with other genomic grammars, we define set operations that treat ranges as sets of integers, including *intersect*, *union*, *difference*, and *complement*. There are two sets of these: parallel and merging. For example, the parallel intersection (*x %intersect% y*) finds the intersecting range between *xi* and *yi* for *i* in *1. . . n*, where *n* is the length of both *x* and *y*. In contrast, the merging intersection (*intersect_ranges(x, y)*) returns a new set of disjoint ranges representing wherever there was overlap between a range in *x* and a range in *y*. Finding the parallel union will fail when two ranges have a gap, so we introduce a *span* operator that takes the union while filling any gap. The *complement* operation is unique in that it is unary. It finds the regions not covered by any of the ranges in a single set. Closely related is the *between* parallel operation, which finds the gap separating *xi* and *yi*. The binary operations are callable from within arithmetic, restriction and aggregation expressions.

To support merging, our algebra recasts finding overlaps or nearest neighbors between two genomic regions as variants of the relational join operator. A join acts on two GRanges objects: *x* and *y*. The join operator is relational in the sense that metadata from the *x* and *y* ranges are retained in the joined range. All join operators in the plyranges DSL generate a set of hits based on overlap or proximity of ranges and use those hits to merge the two datasets in different ways. There are four supported matching algorithms: *overlap*, *nearest*, *precede*, and *follow* (figure 2). We can further restrict the matching by whether the query is completely *within* the subject, and adding the *directed* suffix ensures that matching ranges have the same direction (strand).

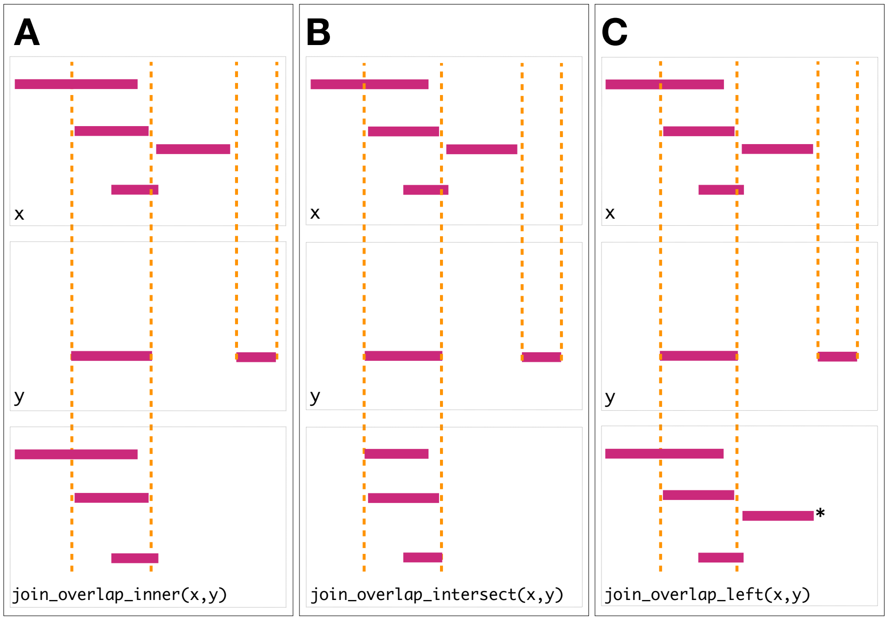


Figure 2: Illustration of the three overlap join operators. Each join takes two GRanges objects, x and y as input. A 'Hits' object for the join is computed which consists of two components. The first component contains the indices of the ranges in x that have been overlapped (the rectangles of x that cross the orange lines). The second component consists of the indices of the ranges in y that overlap the ranges in x. In this case a range in y overlaps the ranges in x three times, so the index is repeated three times. The resulting 'Hits' object is used to modify x by where it was 'hit' by y and merge all metadata columns from x and y based on the indices contained in the 'Hits' object. This procedure is applied generally in the `plyranges` DSL for both overlap and nearest neighbor operations. The join semantics alter what is returned: **A**: for an **inner** join the x ranges that are overlapped by y are returned. The returned ranges also include the metadata from the y range that overlapped the three x ranges. **B** An **intersect** join is identical to an inner join except that the intersection is taken between the overlapped x ranges and the y ranges. **C** For the **left** join all x ranges are returned regardless of whether they are overlapped by y. In this case the third range (rectangle with the asterisk next to it) of the join would have missing values on metadata columns that came from y.

For merging based on the hits, we have three modes: *inner*, *intersect* and *left*. The *inner* overlap join is similar to the conventional inner join in that there is a row in the result for every match. A major difference is that the matching is not by identity, so we have to choose one of the ranges from each pair. We always choose the left range. The *intersect* join uses the intersection instead of the left range. Finally, the overlap *left* join is akin to left outer join in Codd's relational algebra: it performs an overlap inner join but also returns all x ranges that are not hit by the y ranges.

A

```
library(plyranges)
gwas <- read_bed('snps.bed')
exons <- read_bed('exons.bed')
res <- exons %>%
          join_overlap_inner(snps) %>%
          group_by(rsID) %>%
          summarise(n = n_distinct(exonID))
```

B

```
library(GenomicRanges)
library(rtracklayer)
gwas  <- import('snps.bed')
exons <- import('exons.bed')
hits <- findOverlaps(exons, gwas,
                     ignore.strand = FALSE)
olap <- splitAsList(exons$name[queryHits(hits)],
                    gwas$name[subjectHits(hits)])
n <- lengths(unique(olap))
res <- DataFrame(rsID = names(n),
                 n = as.integer(n))
```

Figure 3: Idiomatic code examples for `plyranges` (**A**) and `GenomicRanges` (**B**) illustrating an overlap and aggregate operation that returns the same result. In each example, we have two BED files consisting of SNPs that are genome-wide association study (GWAS) hits and reference exons. Each code block counts for each SNP the number of distinct exons it overlaps. The `plyranges` code achieves this with an overlap join followed by partitioning and aggregation. Strand is ignored by default here. The `GenomicRanges` code achieves this using the 'Hits' and 'List' classes and their methods.

Since the GRanges object is a tabular data structure, our grammar includes operators to filter, sort and aggregate by columns in a GRanges. These operations can be performed over partitions formed using the *group_by* modifier. Together with our algebra for arithmetic and merging, these operations conform to the semantics and syntax of the `dplyr` grammar. Consequently, `plyranges` code is generally more compact than the equivalent `GenomicRanges` code (figure 3).

Developing workflows with `plyranges`

Here we provide illustrative examples of using the `plyranges` DSL to show how our grammar could be integrated into genomic data workflows. As we construct the workflows we show the data output intermittently to assist the reader in understanding the pipeline steps. The workflows highlight how interoperability with existing Bioconductor infrastructure, enables easy access to public datasets and methods for analysis and visualization.

Peak Finding

In the workflow of ChIP-seq data analysis, we are interested in finding peaks from islands of coverage over chromosome. Here we will use `plyranges` to call peaks from islands of coverage above 8 then plot the region surrounding the tallest peak.

Using `plyranges` and the the Bioconductor package `AnnotationHub` [13] we can download and read BigWig files from ChIP-Seq experiments from the Human Epigenome Roadmap project [14]. Here we analyse a BigWig file corresponding to H3 lysine 27 trimethylation (H3K27Me3) of primary T CD8+ memory cells from peripheral blood, focussing on coverage islands over chromosome 10.

First, we extract the genome information from the BigWig file and filter to get the range for chromosome 10. This range will be used as a filter when reading the file.

```r
library(plyranges)
chr10_ranges <- bw_file %>%
  get_genome_info() %>%
  filter(seqnames == "chr10")
```

Then we read the BigWig file only extracting scores if they overlap chromosome 10. We also add the genome build information to the resulting ranges. This book-keeping is good practice as it ensures the integrity of any downstream operations such as finding overlaps.

```r
chr10_scores <- bw_file %>%
  read_bigwig(overlap_ranges = chr10_ranges) %>%
  set_genome_info(genome = "hg19")
chr10_scores
```

```
#> GRanges object with 5789841 ranges and 1 metadata column:
#>             seqnames              ranges strand |              score
#>                <Rle>           <IRanges>  <Rle> |          <numeric>
#>         [1]    chr10             1-60602      * | 0.0422799997031689
#>         [2]    chr10         60603-60781      * |  0.163240000605583
#>         [3]    chr10         60782-60816      * |  0.372139990329742
#>         [4]    chr10         60817-60995      * |  0.163240000605583
#>         [5]    chr10         60996-61625      * | 0.0422799997031689
#>         ...      ...                 ...    ... .                ...
#>   [5789837]    chr10 135524723-135524734      * |  0.144319996237755
#>   [5789838]    chr10 135524735-135524775      * |  0.250230014324188
#>   [5789839]    chr10 135524776-135524784      * |  0.427789986133575
#>   [5789840]    chr10 135524785-135524806      * |  0.730019986629486
#>   [5789841]    chr10 135524807-135524837      * |   1.03103005886078
#>   -------
#>   seqinfo: 25 sequences from hg19 genome
```

We then filter for regions with a coverage score greater than 8, and following this reduce individual runs to ranges representing the islands of coverage. This is achieved with the `reduce_ranges()` function, which allows a summary to be computed over each island: in this case we take the maximum of the scores to find the coverage peaks over chromosome 10.

```
all_peaks <- chr10_scores %>%
  filter(score > 8) %>%
  reduce_ranges(score = max(score))
all_peaks
```

```
#> GRanges object with 1085 ranges and 1 metadata column:
#>           seqnames              ranges strand |            score
#>              <Rle>           <IRanges>  <Rle> |        <numeric>
#>        [1]    chr10     1299144-1299370      * | 13.2264003753662
#>        [2]    chr10     1778600-1778616      * | 8.20512008666992
#>        [3]    chr10     4613068-4613078      * | 8.76027011871338
#>        [4]    chr10     4613081-4613084      * | 8.43659973144531
#>        [5]    chr10             4613086      * | 8.11507987976074
#>        ...      ...                 ...    ... .              ...
#>     [1081]    chr10 135344482-135344488      * | 9.23237991333008
#>     [1082]    chr10 135344558-135344661      * |  11.843409538269
#>     [1083]    chr10 135344663-135344665      * | 8.26965999603271
#>     [1084]    chr10 135344670-135344674      * | 8.26965999603271
#>     [1085]    chr10 135345440-135345441      * | 8.26965999603271
#>   -------
#>   seqinfo: 25 sequences from hg19 genome
```

Returning to the GRanges object containing normalized coverage scores, we filter to find the coordinates of the peak containing the maximum coverage score. We can then find a 5000 nt region centered around the maximum position by anchoring and modifying the width.

```
chr10_max_score_region <- chr10_scores %>%
  filter(score == max(score)) %>%
  anchor_center() %>%
  mutate(width = 5000)
```

Finally, the overlap inner join is used to restrict the chromosome 10 coverage islands, to the islands that are contained in the 5000nt region that surrounds the max peak (figure 4).

```
peak_region <- chr10_scores %>%
  join_overlap_inner(chr10_max_score_region)
peak_region
```

```
#> GRanges object with 890 ranges and 2 metadata columns:
#>          seqnames            ranges strand |            score.x
#>             <Rle>         <IRanges>  <Rle> |          <numeric>
#>       [1]    chr10 21805891-21805988      * | 0.0206599999219179
#>     [2]    chr10 21805989-21806000      * | 0.0211200006306171
#>     [3]    chr10 21806001-21806044      * |  0.022069999948144
#>     [4]    chr10 21806045-21806049      * | 0.0215900000184774
#>     [5]    chr10 21806050-21806081      * | 0.0211200006306171
#>     ...      ...               ...    ... .                ...
#>   [886]    chr10          21810878      * |   5.24951982498169
#>   [887]    chr10          21810879      * |   5.83534002304077
#>   [888]    chr10 21810880-21810884      * |   6.44267988204956
#>   [889]    chr10 21810885-21810895      * |   7.07054996490479
#>   [890]    chr10 21810896-21810911      * |   6.44267988204956
#>                   score.y
#>                 <numeric>
#>     [1] 29.9573001861572
#>     [2] 29.9573001861572
#>     [3] 29.9573001861572
#>     [4] 29.9573001861572
#>     [5] 29.9573001861572
#>     ...              ...
#>   [886] 29.9573001861572
#>   [887] 29.9573001861572
#>   [888] 29.9573001861572
#>   [889] 29.9573001861572
#>   [890] 29.9573001861572
#>   -------
#>   seqinfo: 25 sequences from hg19 genome
```

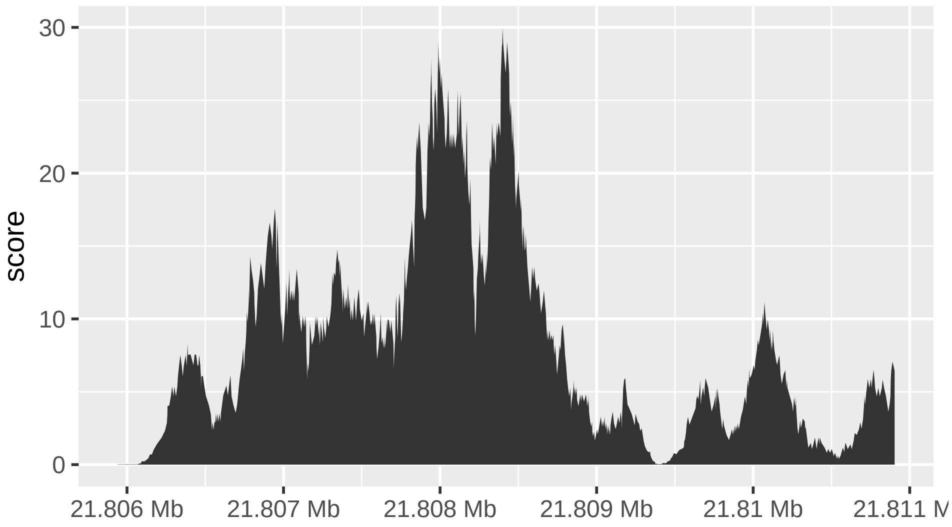


Figure 4: The final result of the `plyranges` operations to find a 5000nt region surrounding the peak of normalised coverage scores over chromosome 10, displayed as a density plot.

Computing Windowed Statistics

Another common operation in genomics data analysis is to compute data summaries over genomic windows. In `plyranges` this can be achieved via the `group_by_overlaps()`

operator. We bin and count and find the average GC content of reads from a H3K27Me3 ChIP-seq experiment by the Human Epigenome Roadmap Consortium.

We can directly obtain the genome information from the header of the BAM file: in this case the reads were aligned to the hg19 genome build and there are no reads overlapping the mitochondrial genome.

```
locations <- h1_bam_sorted %>%
  read_bam() %>%
  get_genome_info()
```

Next we only read in alignments that overlap the genomic locations we are interested in and select the query sequence. Note that the reading of the BAM file is deferred: only alignments that pass the filter are loaded into memory. We can add another column representing the GC proportion for each alignment using the `letterFrequency()` function from the `Biostrings` package [15]. After computing the GC proportion as the score column, we drop all other columns in the GRanges object.

```
alignments <- h1_bam_sorted %>%
  read_bam() %>%
  filter_by_overlaps(locations) %>%
  select(seq) %>%
  mutate(
    score = as.numeric(letterFrequency(seq, "GC", as.prob = TRUE))
  ) %>%
  select(score)
alignments
```

```
#> GRanges object with 8275595 ranges and 1 metadata column:
#>             seqnames            ranges strand |             score
#>                <Rle>         <IRanges>  <Rle> |         <numeric>
#>         [1]    chr10       50044-50119      - | 0.276315789473684
#>         [2]    chr10       50050-50119      + |              0.25
#>         [3]    chr10       50141-50213      - | 0.447368421052632
#>         [4]    chr10       50203-50278      + | 0.263157894736842
#>         [5]    chr10       50616-50690      + | 0.276315789473684
#>         ...      ...               ...    ... .               ...
#>   [8275591]     chrY 57772745-57772805      - | 0.513157894736842
#>   [8275592]     chrY 57772751-57772800      + | 0.526315789473684
#>   [8275593]     chrY 57772767-57772820      + | 0.565789473684211
#>   [8275594]     chrY 57772812-57772845      + |              0.25
#>   [8275595]     chrY 57772858-57772912      + | 0.592105263157895
#>   -------
#>   seqinfo: 24 sequences from an unspecified genome
```

Finally, we create 10000nt tiles over the genome and compute the number of reads and average GC content over all reads that fall within each tile using an overlap join and merging endpoints.

```
bins <- locations %>%
  tile_ranges(width = 10000L)

alignments_summary <- bins %>%
  join_overlap_inner(alignments) %>%
  disjoin_ranges(n = n(), avg_gc = mean(score))
alignments_summary
```

```
#> GRanges object with 286030 ranges and 2 metadata columns:
#>             seqnames            ranges strand |         n            avg_gc
#>                <Rle>         <IRanges>  <Rle> | <integer>         <numeric>
#>        [1]     chr10       49999-59997      * |        88 0.369019138755981
#>        [2]     chr10       59998-69997      * |        65 0.434210526315789
#>        [3]     chr10       69998-79996      * |        56 0.386513157894737
#>        [4]     chr10       79997-89996      * |        71  0.51297257227576
#>        [5]     chr10       89997-99996      * |        64 0.387746710526316
#>        ...       ...               ...    ... .       ...               ...
#>   [286026]      chrY 57722961-57732958      * |        36 0.468201754385965
#>   [286027]      chrY 57732959-57742957      * |        38 0.469529085872576
#>   [286028]      chrY 57742958-57752956      * |        38 0.542936288088643
#>   [286029]      chrY 57752957-57762955      * |        42 0.510651629072682
#>   [286030]      chrY 57762956-57772954      * |       504 0.526942355889723
#>   -------
#>   seqinfo: 24 sequences from an unspecified genome; no seqlengths
```

Quality Control Metrics

We have created a GRanges object from genotyping performed on the H1 cell line, consisting of approximately two million single nucleotide polymorphisms (SNPs) and short insertion/deletions (indels). The GRanges object consists of 7 columns, relating to the alleles of a SNP or indel, the B-allele frequency, log relative intensity of the probes, GC content score over a probe, and the name of the probe. We can use this information to compute the transition-transversion ratio, a quality control metric, within each chromosome in GRanges object.

First we filter out the indels and mitochondrial variants. Then we create a logical vector corresponding to whether there is a transition event.

```
h1_snp_array <- h1_snp_array %>%
  filter(!(ref %in% c("I", "D")), seqnames != "M") %>%
  mutate(transition = (ref %in% c("A", "G") & alt %in% c("G","A"))|
                      (ref %in% c("C","T") & alt %in% c("T", "C")))
```

We then compute the transition-transversion ratio over each chromosome using `group_by()` in combination with `summarize()` (figure 5).

```
ti_tv_results <- h1_snp_array %>%
  group_by(seqnames) %>%
  summarize(n_snps = n(),
            ti_tv = sum(transition) / sum(!transition))
ti_tv_results
```

```
#> DataFrame with 24 rows and 3 columns
#>     seqnames    n_snps             ti_tv
#>        <Rle> <integer>         <numeric>
#> 1          Y      2226   1.4381161007667
#> 2          6    154246 3.32013219807305
#> 3         13     83736 3.40669403220714
#> 4         10    120035 3.49400973418195
#> 5          4    153243 3.29528828096533
#> ...      ...       ...               ...
#> 20        16     77538 3.19827819589583
#> 21        12    113208 3.47851887016378
#> 22        20     57073  3.7121036988111
#> 23        21     32349 3.50480434479877
#> 24         X     55495 3.58219800181653
```

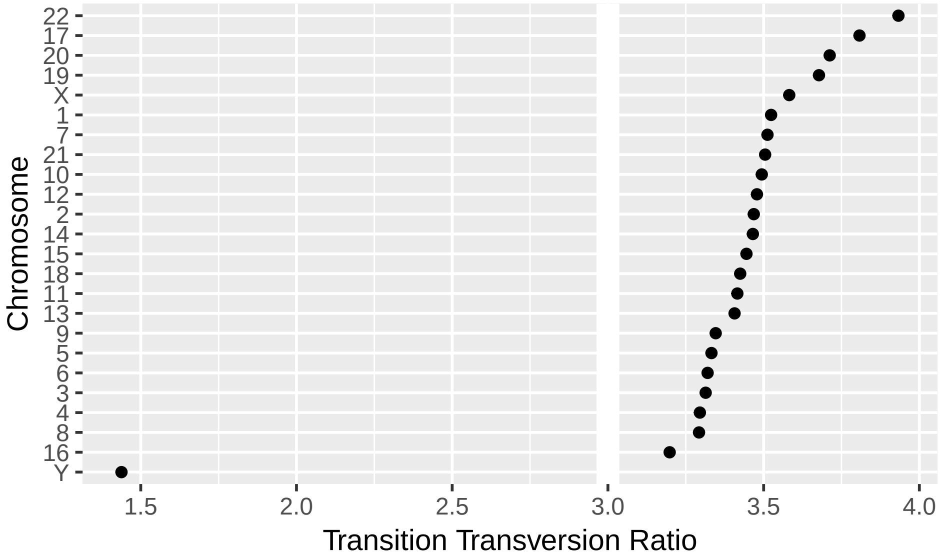


Figure 5: The final result of computing quality control metrics over the SNP array data with `plyranges`, displayed as a dot plot. Chromosomes are ordered by their estimated transition-transversion ratio. A white reference line is drawn at the expected ratio for a human exome.

Discussion

The design of `plyranges` adheres to well understood principles of language and API design: cognitive consistency, cohesion, expressiveness and endomorphism [16]. To varying degrees, these principles also underlie the design of `dplyr` and the Bioconductor infrastructure.

We have aimed for `plyranges` to have a simple and direct mapping to the user's cognitive model, i.e., how the user thinks about the data. This requires careful selection of the level of abstraction so that the user can express workflows in the language of genomics. This motivates the adoption of the tidy GRanges object as our central data structure. The basic data.frame and `dplyr` tibble lack any notion of genomic ranges and so could not easily support our genomic grammar, with its specific verbs for range-oriented data manipulation. Another example of cognitive consistency is how `plyranges` is insensitive to direction/strand by default when, e.g., detecting overlaps. `GenomicRanges` has the opposite behavior. We believe that defaulting to purely spatial overlap is most intuitive to most users.

To further enable cognitive consistency, `plyranges` functions are cohesive. A function is defined to be cohesive if it performs a singular task without producing any side-effects. Singular tasks can always be broken down further at lower levels of abstraction. For example, to resize a range, the user needs to specify which position (start, end, midpoint) should be invariant over the transformation. The `resize()` function from the `GenomicRanges` package has a `fix` argument that sets the anchor, so calling `resize()` coalesces anchoring and width modification. The coupling at the function call level is justified since the effect of setting the width depends on the anchor. However, `plyranges` increases cohesion and decouples the anchoring into its own function call.

Increasing cohesion simplifies the interface to each operation, makes the meaning of arguments more intuitive, and relies on function names as the primary means of expression, instead of a more complex mixture of function and argument names. This results in the user being able to conceptualize the `plyranges` DSL as a flat catalog of functions, without having to descend further into documentation to understand a function's arguments. A flat function catalog also enhances API discoverability, particularly through auto-completion in integrated developer environments (IDEs). One downside of pushing cohesion to this extreme is that function calls become coupled, and care is necessary to treat them as a group when modifying code.

Like `dplyr`, `plyranges` verbs are functional: they are free of side effects and are generally endomorphic, meaning that when the input is a GRanges object they return a GRanges object. This enables chaining of verbs through syntax like the forward pipe operator from the `magrittr` package. This syntax has a direct cognitive mapping to natural language and the intuitive notion of pipelines. The low-level object-oriented APIs of Bioconductor tend to manipulate data via sub-replacement functions, like `start(gr) <- x`. These ultimately produce the side effect of replacing a symbol mapping in the current environment and thus are not amenable to so-called fluent syntax.

Expressiveness relates to the information content in code: the programmer should be able to clarify intent without unnecessary verbosity. For example, our overlap-based join operations are more concise than the multiple steps necessary to achieve the same effect in the original `GenomicRanges` API. In other cases, the `plyranges` API increases verbosity for the sake of clarity and cohesion. Explicitly calling *anchor()* can require more typing, but the code is easier to comprehend. Another example is the set of routines for importing genomic annotations, including *read_gff()*, *read_bed()*, and *read_bam()*. Compared to the generic *import()* in `rtracklayer`, the explicit format-based naming in `plyranges` clarifies intent and the type of data being returned. Similarly, every `plyranges` function that computes with strand information indicates its intentions by including suffixes such as *directed*, *upstream* or *downstream* in its name, otherwise strand is ignored. The `GenomicRanges` API does not make this distinction explicit in its function naming, instead relying on a parameter that defaults to strand sensitivity, an arguably confusing behavior.

The implementation of plyranges is built on top of Bioconductor infrastructure, meaning most functions are constructed by composing generic functions from core Bioconductor packages. As a result, any Bioconductor packages that uses data structures that inherit from GRanges will be able to use plyranges for free. Another consequence of building on top of Bioconductor generics is that the speed and memory usage of plyranges functions are similar to the highly optimized methods implemented in Bioconductor for GRanges objects.

A caveat to constructing a compatible interface with dplyr is that plyranges makes extensive use of non-standard evaluation in R via the rlang package [17]. Simply, this means that computations are evaluated in the context of the GRanges objects. Both dplyr and plyranges are based on the rlang language, because it allows for more expressive code that is free of repeated references to the container. Implicitly referencing the container is particularly convenient when programming interactively. Consequently, when programming with plyranges, a user needs to generally understand the rlang language and how to adapt their code accordingly. Users familiar with the tidyverse should already have such knowledge.

Conclusion

We have shown how to create expressive and reproducible genomic workflows using the plyranges DSL. By realising that the GRanges data model is tidy we have highlighted how to implement a grammar for performing genomic arithmetic, aggregation, restriction and merging. Our examples show that plyranges code is succinct, human readable and can take advantage of the interoperability provided by the Bioconductor ecosystem and the R language.

We also note that the grammar elements and design principles we have described are programming language agnostic and could be easily be implemented in another language where genomic information could be represented as a tabular data structure. We chose R because it is what we are familiar with and because the aforementioned Bioconductor packages have implemented the GRanges data structure.

We aim to continue developing the plyranges package and to extend it for use with more complex data structures, such as the SummarizedExperiment class, the core Bioconductor data structure for representing experimental results (e.g., counts) from multiple sample experiments in conjunction with feature and sample metadata. Although, the SummarizedExperiment is not strictly tidy, it does consist of three tidy data structures that are related by feature and sample identifiers. Therefore, the grammar and design of the plryanges DSL is naturally extensible to the SummarizedExperiment.

As the plyranges interface encourages tidy data practices, it integrates well with the grammar of graphics [18]. To achieve responsive performance, interactive graphics rely on lazy data access and computing patterns, so the deferred mechanisms within plyranges should help support interactive genomics applications.

The plyranges package can be obtained via the Bioconductor project website https://bioconductor.org or accessed via Github https://github.com/sa-lee/plyranges.

Methods

Data Availability

The BigWig file for the H3K27Me3 primary T CD8+ memory cells from peripheral blood ChIP-seq data was downloaded from the `AnnotationHub` package (2.12.0) under accession AH33458. The BAM file corresponding to the H1 cell line ChIP-seq data is available at GEO under accession GSM433167. The SNP array data for the H1 cell line data is available at GEO under accession GSM1463263.

Software Versions

To produce the workflows as described in results section we used R version 3.5 with the development version of `plyranges` (1.1.10) and the `BioStrings` package (2.48.0) installed.

All code required to reproduce this article is available at https://github.com/sa-lee/plyranges-paper.

Acknowledgements

We would like to thank Dr Matthew Ritchie at the Walter and Eliza Hall Institute and Dr Paul Harrison for their feedback on earlier drafts of this work. We would also like to thank Lori Shepherd and Hèrve Pages for the code review they performed. This article was written with `knitr` [19] and the figures were made with `ggbio` [20].

References

[1] Kozanitis, Christos et al. "Using Genome Query Language to uncover genetic variation". en. In: *Bioinformatics* 30.1 (Jan. 2014), pp. 1–8. ISSN: 1367-4803, 1367-4811. DOI: 10.1093/bioinformatics/btt250.

[2] Christos Kozanitis and David A Patterson. "GenAp: a distributed SQL interface for genomic data". en. In: *BMC Bioinformatics* 17 (Feb. 2016), p. 63. ISSN: 1471-2105. DOI: 10.1186/s12859-016-0904-1.

[3] Kaitoua, A et al. "Framework for Supporting Genomic Operations". In: *IEEE Trans. Comput.* 66.3 (Mar. 2017), pp. 443–457. ISSN: 0018-9340. DOI: 10.1109/TC.2016.2603980.

[4] Aaron R Quinlan and Ira M Hall. "BEDTools: a flexible suite of utilities for comparing genomic features". en. In: *Bioinformatics* 26.6 (Mar. 2010), pp. 841–842. ISSN: 1367-4803, 1367-4811. DOI: 10.1093/bioinformatics/btq033.

[5] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2018.

[6] Michael Lawrence et al. "Software for Computing and Annotating Genomic Ranges". In: *PLoS Comput. Biol.* 9 (2013). ISSN: 1553-734X. DOI: 10.1371/journal.pcbi.1003118.

[7] Wolfgang Huber et al. "Orchestrating high-throughput genomic analysis with Bioconductor". en. In: *Nat. Methods* 12.2 (Feb. 2015), pp. 115–121. ISSN: 1548-7091, 1548-7105. DOI: 10.1038/nmeth.3252.

[8] Ryan K Dale, Brent S Pedersen, and Aaron R Quinlan. "Pybedtools: a flexible Python library for manipulating genomic datasets and annotations". en. In: *Bioinformatics* 27.24 (Dec. 2011), pp. 3423–3424. ISSN: 1367-4803, 1367-4811. DOI: 10.1093/bioinformatics/btr539.

[9] Kent A. Riemondy et al. "valr: Reproducible Genome Interval Arithmetic in R". In: *F1000Research* (2017). DOI: 10.12688/f1000research.11997.1.

[10] Hadley Wickham. "Tidy Data". In: *Journal of Statistical Software, Articles* 59.10 (2014), pp. 1–23. ISSN: 1548-7660. DOI: 10.18637/jss.v059.i10.

[11] Hadley Wickham et al. *dplyr: A Grammar of Data Manipulation*. R package version 0.7.4. 2017.

[12] Stefan Milton Bache and Hadley Wickham. *magrittr: A Forward-Pipe Operator for R*. R package version 1.5. 2014.

[13] Martin Morgan. *AnnotationHub: Client to access AnnotationHub resources*. R package version 2.10.1. 2017.

[14] Roadmap Epigenomics Consortium et al. "Integrative analysis of 111 reference human epigenomes". en. In: *Nature* 518.7539 (Feb. 2015), pp. 317–330. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14248.

[15] H. Pagès et al. *Biostrings: Efficient manipulation of biological strings*. R package version 2.48.0. 2018.

[16] T R G Green and M Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages & Computing* 7.2 (June 1996), pp. 131–174. ISSN: 1045-926X. DOI: 10.1006/jvlc.1996.0009.

[17] Lionel Henry and Hadley Wickham. *rlang: Functions for Base Types and Core R and 'Tidyverse' Features*. http://rlang.tidyverse.org, https://github.com/r-lib/rlang. 2017.

[18] Wickham, Hadley. *ggplot2: Elegant Graphics for Data Analysis*. en. Use R! Springer International Publishing, June 2016. ISBN: 9783319242750, 9783319242774. DOI: 10.1007/978-3-319-24277-4.

[19] Yihui Xie. *Dynamic Documents with R and knitr*. 2nd. ISBN 978-1498716963. Boca Raton, Florida: Chapman and Hall/CRC, 2015.

[20] Tengfei Yin, Dianne Cook, and Michael Lawrence. “ggbio: an R package for extending the grammar of graphics for genomic data”. In: *Genome Biology* 13.8 (2012), R77.