plyranges: a grammar for transforming genomics data

Stuart Lee *Monash University*

Michael Lawrence *Genentech*

Di Cook *Monash University*

The Bioconductor project has created many useful data abstractions for analysing high-throughput genomics experiments. However, there is a cognitive load placed on a user in learning a data abstraction and understanding its appropriate use. Throughout a standard workflow, a user must navigate and know many of these abstractions to perform an genomic analysis task, when a single data abstraction, a GRanges object will suffice. The GRanges class naturally represent genomic intervals and their associated measurements. By recognising that the GRanges class follows 'tidy' data principles we have created a grammar of genomic data transformation. The grammar defines verbs for performing actions on and between genomic interval data. It provides a principled way of performing common genomic data analysis tasks through a coherent interface to existing Bioconductor infrastructure, resulting in human readable analysis workflows. We have implemented this grammar as a Bioconductor/R package called plyranges.

Keywords: Bioconductor, Grammar, Genomes, Data Analysis

Introduction

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 implementations 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

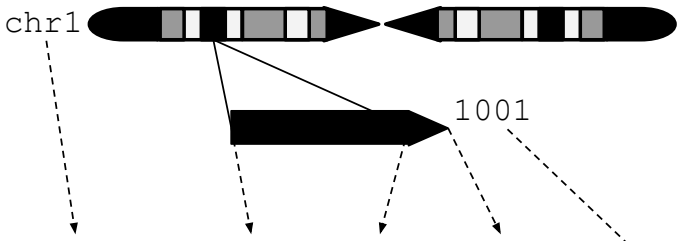
Language (GQL) and its distributed implementation GenAp which use an SQL-like syntax for fast retrieval of information of unprocessed sequencing data [1, 2]. Similarly, the Genometric Query Language (GMQL) implements a relational algebra for combining genomic datasets [3]. The command line application 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 analysing 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 API that maps notions from the general relational algebra to operations on tidy data. It expresses each operation as a cohesive, endomorphic verb. 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.



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).

Genomic relational algebra

Data model

The `plyranges` DSL is built on the core Bioconductor data structure GRanges, which is essentially 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 follows 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
Aggregation	***summarise()***	aggregate over column(s)
	disjoin_ranges()	aggregate column(s) over the union of end coordinates
	reduce_ranges()	aggregate column(s) by merging overlapping and neighbouring ranges
Arithmetic (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
Arithmetic (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
Merging	*join_overlap_*()*	merge by overlapping ranges
	join_nearest	merge by nearest neighbour 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
Modifier	*anchor_direction()*	fix coordinates at direction
	group_by()	partition by column(s)
	group_by_overlaps()	partition by overlaps
Restriction	***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 categorised into one of: aggregation, unary or binary arithmetic, merging, modifier, or restriction. 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. The parallel operations map to infix operators, which we surround with % symbols, a convention borrowed from R syntax. For example, the parallel intersection (x *%intersect%* y) finds the intersecting range between xi and yi for i in $1 \ldots n$, 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. We use the infix syntax for the parallel operations, since it is the conventional syntax for parallel, binary operations in R. 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 neighbours between two genomic regions as variants of the relational join operator. A join acts on two GRanges objects, a query and a subject. The join operator is relational in the sense that metadata from the query and subject ranges is 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* (figures 2

and 3). 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).

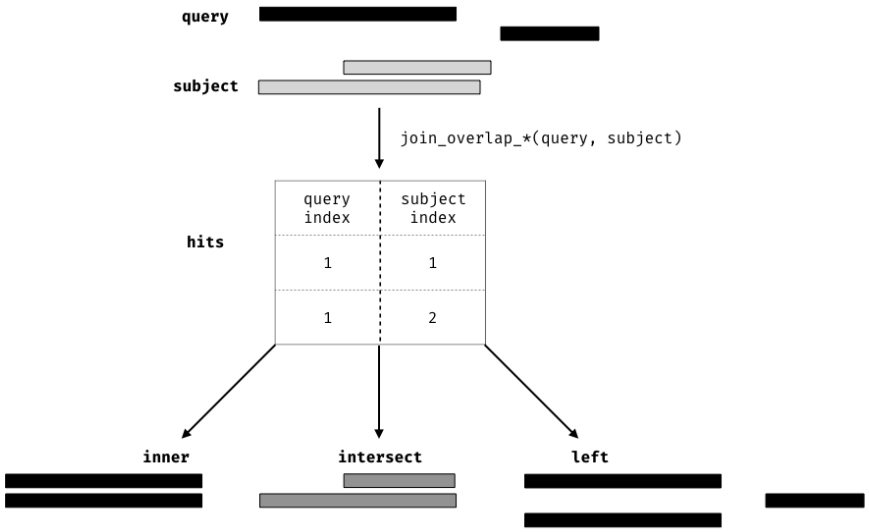


Figure 2: Illustration of the three overlap join operators. Each join takes query and subject range as input (black and light gray rectangles, respectively). An index for the join is computed, returning a Hits object, which contains the indices of where the subject overlaps the query range. This index is used to expand the query ranges by where it was 'hit' by the subject ranges. The join semantics alter what is returned: for an extbfinner join the query range is returned for each match, for an extbfintersect the intersection is taken between overlapping ranges, and for a extbfleft join all query ranges are returned even if the subject range does not overlap them. This principle is gnerally applied through the 'plyranges' DSL for both overlaps and nearest neighbour operations.

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 Cobb's relational algebra: it performs an overlap inner join but also returns all query ranges that are not hit by the subject.

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.

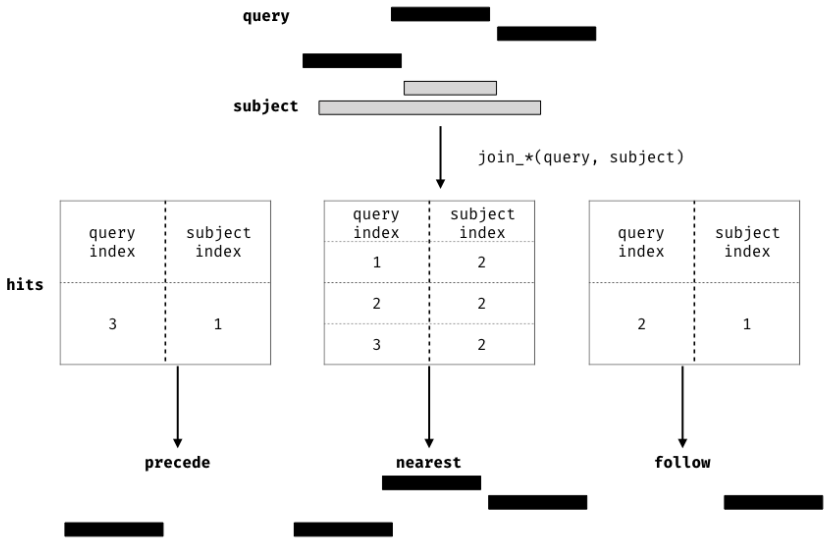


Figure 3: Illustration of neighbour finding joins. Each join takes a query and subject range and computes a 'Hits' object. For the extbfnearest join all query ranges are returned as they are all nearest neighbours of the second subject range. For the extbffollow join there is only one query range that follows any subject range. Likewise for the extbfprecede join, there is only one query range that precedes a subject range.

Design principles

The design of `plyranges` adheres to well understood principles of language and API design: cognitive consistency, cohesion, expressiveness and endomorphism [12]. To varying degrees, these principles also underlie the design of `dplyr` and the Bioconductor infrastructure.

Cognitive consistency and fluency

We aim for our interfaces 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.

Like `dplyr`, `plyranges` verbs are functional: they are free of side effects and return their result. This enables chaining of verbs through syntax like the forward pipe operator (`%>%`, read as "then") of the `magrittr` package [13]. This has 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.

Cohesion

A function is cohesive if it performs a singular task without producing any side-effects. Singular tasks are not necessarily atomic; they 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. Since functions are superordinate to their arguments, flattening the API at the function level enables the user to conceptualize the API as a catalog of functions, without having to descend further. A flat function catalog also enhances API discoverability, particularly through auto-completion in 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.

Expressiveness

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 unintuitive behavior.

Results

Here we provide illustrative examples of using the `plyranges` DSL to show how our grammar could be integrated into genomic data workflows. We also highlight how interoperability with existing Bioconductor infrastructure, enables easy access to public datasets and methods for analysis and visualisation.

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` [14] we can download and read BigWig files from ChIP-Seq experiments from the Human Epigenome Roadmap project [15]. 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.

```
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.

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

After filtering for regions with a coverage score greater than 8, we can reduce individual runs to ranges representing the islands of coverage by using the `reduce_ranges()` function. This function allows a summary to be computer 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))
```

Returning to the GRanges object containing normalised 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 normalised coverage scores that are within the 5000nt region that contain the max peak on chromosome 10 (figure 4).

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

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. To generate bins of fixed width 10000nt we apply the `tile_ranges()` function to the genomic coordinates in locations.

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

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

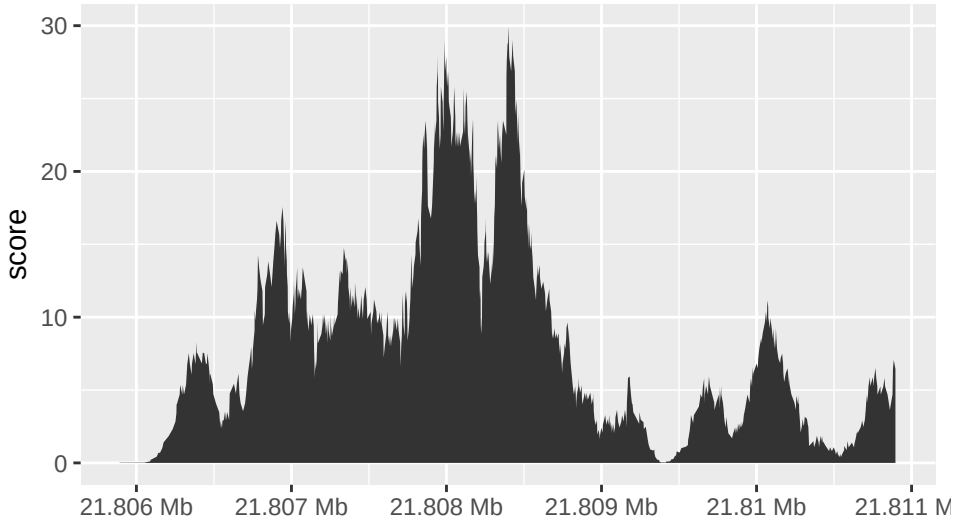


Figure 4: Visualisation of normalised coverage scores accross a 5000nt region of chromosome 10 from H3K27Me3 ChIP-Seq assay from the Human Epigenome Roadmap project.

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 [16].

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

Finally, we use `group_by_overlaps()` to see where the alignments overlap the genomic windows and then apply `summarize()` to compute the total number of reads and average GC content within each window.

```
alignments_summary <- bins %>%
  group_by_overlaps(alignments %>% select(score)) %>%
  summarize(n_reads = n(), avg_gc = mean(score))
```

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))
```

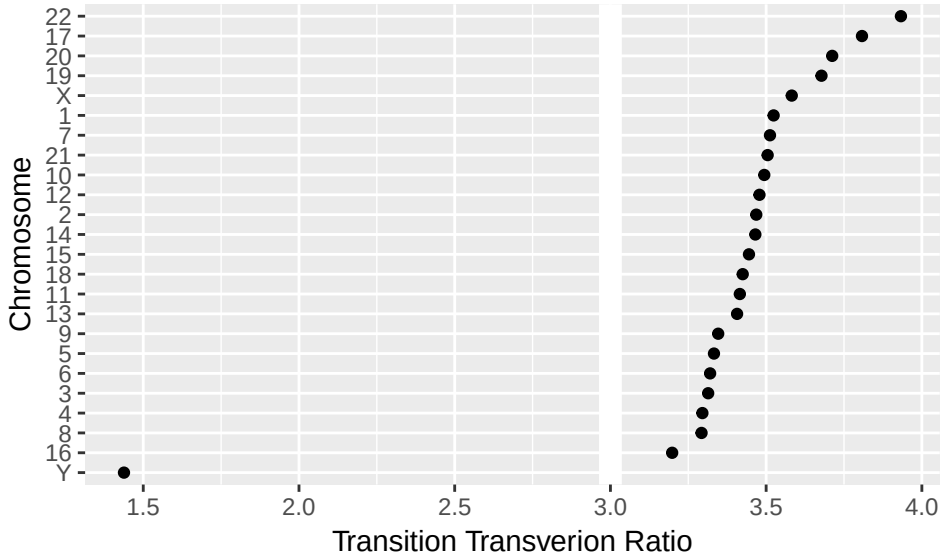


Figure 5: Dot plot of chromosomes ordered by estimated transition-transversion ratio. A white reference line is drawn at the expected ratio for a human exome.

Discussion and 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.

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.

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. The grammar and design of the `plryanges` DSL are naturally extensible to 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 is available on the Bioconductor project website https://bioconductor.org or can be accessed via Github https://github.com/sa-lee/plyranges.

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 report was written with `knitr` [19] and the figures were made with `ggbio` [20]. All code required to reproduce this article is available at https://github.com/sa-lee/plyranges-paper.

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] 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.

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

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

[15] 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.

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

[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.