plyranges: a grammar for manipulating genomics data

Stuart Lee [1] , Michael Lawrence [2] , Di Cook [1]

1 Department of Econometrics and Business Statistics, Clayton, Victoria, Australia
2 Bioinformatics and Computational Biology, Genentech, Inc., South San Francisco, California, United States of America

Abstract

The Bioconductor project has created many powerful abstractions for reasoning about genomics data, such as the *Ranges* data structures for representing genomic intervals. By recognising that these data structures follow 'tidy' data principles we have created a grammar of genomic data manipulation that defines verbs for performing actions on and between genomic interval data. This grammar simplifies performing common genomic data analysis tasks via a consistent interface to existing Bioconductor infrastructure which results in creating human readable analysis pipelines. We have implemented this grammar as an Bioconductor/R package called plyranges.

Introduction

Genomic data may be naturally represented as sets of pairs of integers corresponding to the start and end points of sequences. Further information such as strandedness and chromosome name may be added to these sets to provide biological context. Because of the flexibility of this representation supplemental information such as measurements obtained from an experimental assay or annotations from a genome database can be joined to their relevant sequences. In the Bioconductor/R packages `IRanges` and `GenomicRanges` these representations have been implemented as a suite of data structures called *Ranges* [1]; [2]. These data structures cover many common data types encountered in bioinformatics analyses. For example, a gene can be represented with its coordinates, along with its identifier and the identifiers of its exons; or an RNA-seq experiment may be represented as sets of genes with a matching count column.

The Bioconductor infrastructure for computing with genomic ranges are highly efficient and powerful, however the application programming interface (API) for performing analysis tasks with *Ranges* is complex due to its large number of methods and classes. It also makes resulting scripts written difficult for a non-programmer to read and reason about. An alternative approach would be to implement a domain specific language (DSL) as a fluent interface built on top *Ranges* [3]. The goal of a fluent interface is to enable users to write human-readable code via method chaining and consistent function returns. Fluent interfaces fit naturally in the context of bioinformatics workflows because they enable writing succinct pipelines.

Several other DSLs have been implemented to reason about genomics data. Broadly, these are either implemented as query languages or as command line tools embedded in the Unix environment. 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 from genomic databases and BAM files [4]; [5]. Another

example is the Genometric Query Language (GMQL) which implements a relational algebra for combining big genomic datasets [6].

The command line application BEDtools develops an extensive algebra for performing arithmetic between two or more sets of genomic regions [7]. It also has a python interface which simplifies constructing scripts for performing analyses based on BEDTools [8].

The abstraction provided by the *Ranges* data structures aligns with the concept of tidy data [9]. A tidy dataset is defined to be to a tabular data structure that has observations as rows and columns as variables, and each tidy dataset represents measurements from a single observational unit. The tidy data pattern is useful because it allows us to see how the data relates to the design of an experiment and the variables measured. Consequently, it makes both the modelling and manipulation of data more systematic. The *Ranges* data structure follows this abstraction: it is a rectangular table corresponding to a single biological context. Each row contains a single observation and each column is a variable about that observation.

The *plyranges* API implements a domain specific language using the existing *IRanges* and *GenomicRanges* packages in Bioconductor as a back-end. Consequently, our API still has the speed and efficiency of the aforementioned packages but with a more coherent interface. The API also extends the grammar elements in *dplyr* [10] for performing genomic specific manipulations such as finding overlapping regions or nearest neighbour regions between many *Ranges* . The *plyranges* API is specifically designed to enable fast interactive analysis of *Ranges* objects but can also be used for scripting genomic data workflows.

Design and Implementation

We have designed the API to be fluent. Every function call corresponds to an action on a *Ranges* object (they are named verbs) and where possible functions have few arguments. Each verb is constructed to enable a tab completion based workflow. Both of these aspects reduce the cognitive load on a new user since most manipulations can be performed with a vocabulary of several verbs, rather than having to memorise functions with many arguments that are nouns (as is required in the existing Bioconductor packages). This is also has the advantage of allowing users to write human readable code because verbs describe what the code is doing rather than how its doing it.

Workflows can be composed by chaining verbs together via the forward pipe operator,`%>%` (exported from the R package *magrittr* [11]). This is possible because every function call is an endomorphism: when the input is *Ranges* object the output will also be a *Ranges* object. One advantage of endomorphism is that it does not require any additional learning of classes beyond *Ranges* and the *DataFrame* classes. This strongly deviates from the design of the *Ranges* Bioconductor packages, where many methods return a new class upon return. The Bioconductor design enables efficient computing as users are exposed to low-level features of its API which *plyranges* abstracts away. Method chaining via the pipe operator can also be difficult to debug, as there multiple points of failure.

In order to provide a compatible API with *dplyr*, *plyranges* makes extensive use of non-standard evaluation in R via the *rlang* package [12]. Simply, this means that computations are performed and evaluated in the context of the *Ranges* objects, which emphasises the interactive nature of the API. This has the disadvantage that programming with *plyranges* becomes more difficult because a user needs to capture expressions inside function calls and then unquote them.

Actions on Ranges

The *plyranges* API exports the six core verbs from the *dplyr* package and modifies them for use with *Ranges* objects. The verb `mutate()` takes a Ranges object and a set of name-value pairs and generates a new Ranges object that with modified or new metadata columns or modified core components (start, end, width, seqnames, strand). The use of `mutate()` means that a user no longer needs knowledge of the accessors of the Ranges object, as they can modify them in place. The `filter()` function takes a Ranges object and logical expressions and restricts Ranges object to where the logical expression evaluates to true. The `summarise()` function takes a Ranges object and a set of name-value pairs and aggregates the Ranges according to functions evaluated in the name-value pairs. As `summarise()` is an aggregation it may break the structure of the of a Ranges object, hence it returns a *DataFrame* object. The `select()` function determines which metadata columns are returned and the order they are returned in. The `arrange()` function sorts a Ranges object by named variables. The `group_by()` function creates an implicit grouping of Ranges object according to variables in the Ranges object. This modifies the actions of `mutate()`, `summarise()` and `filter()`, so they are performed on each partition created by the grouping. The `group_by()` operation acts as a replacement for the *GenomicRangesList* class in the original *GenomicRanges* API.

The *plyranges* API introduces additional summary verbs, `reduce_ranges()` and `disjoin_ranges()`that return Ranges objects after being returned. The `reduce_ranges()` operation merges overlapping and neighbour ranges, while `disjoin_ranges()` expands ranges by taking the union of end points.

Arithimetic on Ranges

The *plyranges* API has an expressive algebra for performing arithmetic on Ranges via the verbs `set_width()` and `stretch()`. As the names suggest `set_width()` modifies the width of a Ranges object, while `stretch` extends the start and end of a Ranges object. These can be chained with the anchoring functions `anchor_start()`, `anchor_end()`, `anchor_center()`, `anchor_3p()` or `anchor_5p()`, which fix the coordinates of a Ranges object in place. Moreover, the `shift_` and `flank_` family of functions can be used to shift all coordinates in a Ranges object or generate flanking regions from a Ranges object to the left, right, upstream or downstream of the input. Unlike, the Bioconductor API, *plyranges* makes it explicit via function calls whether to take into account the strand information of a *Ranges* object.

Overlapping Ranges

A common operation to perform between two *Ranges* objects is to find overlaps or nearest neighbours. The *plyranges* API recasts these operations as 'joins' or 'pairing' operations. For overlaps, there are three join operations: `join_overlap_intersect()`, `join_overlap_inner()` and `join_overlap_left()` which are shown in figure (1).

These operations consider any overlap between two input ranges and return any corresponding metadata from both Ranges objects as metadata. The intersect join takes the intersect of the start and end coordinates of overlapping intervals of the query and subject *Ranges* (for *GenomicRanges* it also accounts for sequence name), when there is a overlap the metadata corresponding to the query and subject *Ranges* are returned. Similarly, inner join takes the start and end coordinates of the query *Ranges* that overlap the subject *Ranges* and returns metadata of the overlapping query and subject *Ranges*. Finally, the left join performs a left outer join between the query and subject

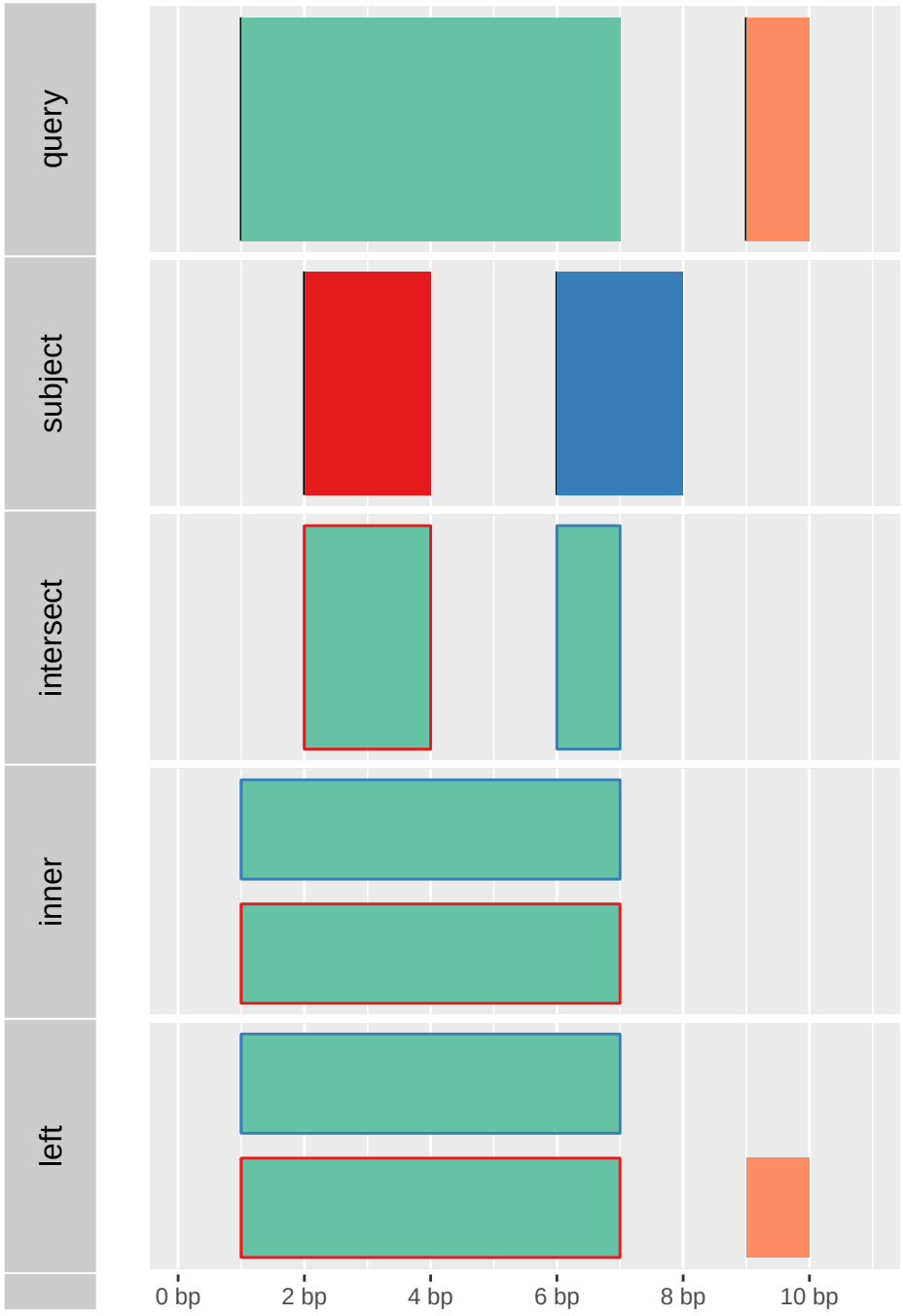

Fig 1. The three overlap joins: the query and subject ranges are coloured by their metadata. When an overlap is performed the resulting range is filled by the query metadata and the metadata from the subject colours the outside of the range.

Ranges, it returns all genomic intervals from the query ranges, and returns missing values in metadata columns when there is no overlap.

A user may also restrict or group by overlaps with the `filter_by_overlaps()`, `filter_by_non_overlaps()` and `group_by_overlaps()`. All overlap methods can be modified with the `within` suffix (which changes the type of overlap from 'any' to 'within') or the `directed` suffix (which takes into account the strand of a *GenomicRanges* object.).

For nearest neighbours, the *plyranges* API provides `join_nearest()`, `join_precede()`, and `join_follow()` functions. These functions are similar to the overlapping functions, in that they return the query ranges that are nearest (or precede or follow) the subject ranges and add metadata from the subject ranges when the query is a nearest neighbour of the subject. Like the overlap joins, these functions can modified with suffixes to find nearest neighbours that are left, right, upstream or downstream of the subject.

The pairing operations, `pair_overlap()`, `pair_nearest()`, `pair_follow()`, and `pair_precede()` are similar to the join operation but instead of returning a *Ranges*, they pair up the subject and query *Ranges* objects into a *DataFrame*, alongside their metadata columns. This data structure is similar to the *Pairs* data structure in the *S4Vectors* [13] package or the BED-PE file format.

The combination of verbs we have defined above encapsulate all operations that can be performed in the original *IRanges* and *GenomicRanges* packages without the user be exposed to new classes. In those packages to perform most operations requires users to learn many classes and perform additional manipulations to return the results of their computation back to a *Ranges* object.

Construction and Import/Output

The methods `as_granges()` and `as_iranges()` for constructing *Ranges* from tabular data structures, such as the *data.frame* in base R. These methods use non-standard evaluation so columns in a *data.frame* can be modified before a *Ranges* object is created. The API also has convenience methods for annotating or extracting annotations from *GRanges* objects with the `set_genome_info()` and `get_genome_info()` functions.

There is a consistent framework for reading and writing files from and to common genomic data formats, using the *rtracklayer* package as a back-end [14]. The methods are implemented in the `read_/write_` family of functions, currently *plyranges* can read and write BAM, BED, BEDPE, narrowPeaks, GFF/GTF, WIG and BigWig files.

Results

As an example, we use the Bioconductor package *AnnotationHub* [15] to search for BigWig files from ChIP-Seq experiments from the Human Epigenome Roadmap project [16]. We choose to focus on assays for primary T CD8+ memory cells from peripheral blood. We can then read the BigWig file corresponding to the H3 lysine 27 trimethylation (H3K27Me3) methylation mark over chromosome 10.

First, we gather the BigWig file and extract its annotation information and filter it to chromosome 10.

```
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. The annotation information from the file is automatically included (in this case the hg19 genome build).

```
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
```

The `reduce_ranges()` operation is used to find coverage peaks across chromosome 10. We can manually set a threshold to restrict genomic regions to have a coverage score of greater than 8, and then merge nearby regions. The maximum coverage is computed over all the coverage scores in regions that were reduced.

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

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

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

Finally, the overlap inner join could be used to restrict the chromosome 10 normalised coverage scores that are within the 5000nt region that contains the max peak on chromosome 10 (visualised in figure 2).

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

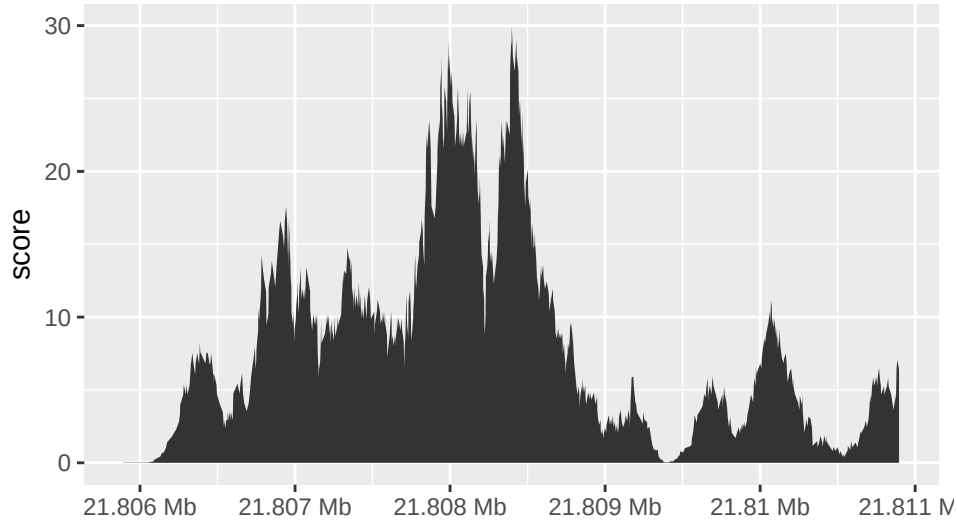

Fig 2. Visualisation of normalised coverage scores accross a 5000nt region of chromosome 10 from H3K27Me3 ChIP-Seq assay from the Human Epigenome Roadmap project.

Availablilty and Future Work

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`. We aim to continue developing the *plyranges* package and extend it for use with more complex data structures such as the *SummarizedExperiment* class, which can be used for analysing transcriptomic and variant data. As the *plyranges* interface encourages tidy data practices it integrates well with the principles of the grammar of graphics, we aim to use it for the visualisation of multimodal biological datasets.

References

1. Lawrence M, Huber W, Pagès H, Aboyoun P, Carlson M, Gentleman R, et al. Software for computing and annotating genomic ranges. PLoS Comput Biol. 2013;9. doi:10.1371/journal.pcbi.1003118

2. Huber W, Carey VJ, Gentleman R, Anders S, Carlson M, Carvalho BS, et al. Orchestrating high-throughput genomic analysis with bioconductor. Nat Methods. Springer Nature; 2015;12: 115–121. doi:10.1038/nmeth.3252

3. Fowler M. Domain-Specific languages [Internet]. Pearson Education; 2010. Available: `https://market.android.com/details?id=book-ri1muolw_YwC`

4. Kozanitis C, Heiberg A, Varghese G, Bafna V. Using genome query language to uncover genetic variation. Bioinformatics. 2014;30: 1–8. doi:10.1093/bioinformatics/btt250

5. Kozanitis C, Patterson DA. GenAp: A distributed SQL interface for genomic data. BMC Bioinformatics. 2016;17: 63. doi:10.1186/s12859-016-0904-1

6. Kaitoua A, Pinoli P, Bertoni M, Ceri S. Framework for supporting genomic operations. IEEE Trans Comput. 2017;66: 443–457. doi:10.1109/TC.2016.2603980

7. Quinlan AR, Hall IM. BEDTools: A flexible suite of utilities for comparing genomic features. Bioinformatics. 2010;26: 841–842. doi:10.1093/bioinformatics/btq033

8. Dale RK, Pedersen BS, Quinlan AR. Pybedtools: A flexible python library for

manipulating genomic datasets and annotations. Bioinformatics. 2011;27: 3423–3424. doi:10.1093/bioinformatics/btr539

9. Wickham H. Tidy data. Journal of Statistical Software, Articles. 2014;59: 1–23. doi:10.18637/jss.v059.i10

10. Wickham H, Francois R, Henry L, Müller K. Dplyr: A grammar of data manipulation [Internet]. 2017. Available: `https://CRAN.R-project.org/package=dplyr`

11. Bache SM, Wickham H. Magrittr: A forward-pipe operator for r [Internet]. 2014. Available: `https://CRAN.R-project.org/package=magrittr`

12. Henry L, Wickham H. Rlang: Functions for base types and core r and 'tidyverse' features [Internet]. 2017. Available: `http://rlang.tidyverse.org`

13. Pagès H, Lawrence M, Aboyoun P. S4Vectors: S4 implementation of vector-like and list-like objects. 2017.

14. Lawrence M, Gentleman R, Carey V. Rtracklayer: An R package for interfacing with genome browsers. Bioinformatics. 2009;25: 1841–1842. doi:10.1093/bioinformatics/btp328

15. Morgan M. AnnotationHub: Client to access annotationhub resources. 2017.

16. Roadmap Epigenomics Consortium, Kundaje A, Meuleman W, Ernst J, Bilenky M, Yen A, et al. Integrative analysis of 111 reference human epigenomes. Nature. 2015;518: 317–330. doi:10.1038/nature14248