plyranges: a grammar for transforming 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 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 an Bioconductor/R package called plyranges.

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` [5–7] 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 in 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 behaviour 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.

Design and Implementation

The `plyranges` DSL is built on the most general Bioconductor data structure, GRanges, which is capable of representing all types of genomic data at a semantic level that roughly matches the intuition of most users. A GRanges is essentially a table, with 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. For example, a GRanges can represent a gene with its chromosomal coordinates, exonic structure, and a count column summarized from an RNA-seq experiment.

By definition GRanges follows the 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 about that observation. Hence, we have designed the `plyranges` DSL to extend the grammar and design principles of `dplyr`: cohesion, consistency, endomorphism, and fluency. All of these principles are defined and discussed below in the context of the GRanges class. Where applicable we contrast our design to the existing Bioconductor infrastructure.

Genomic Relational Algebra

The `plyranges` DSL defines an expressive algebra for performing genomic arithemtic withn and between GRanges objects. Often in analyses of genomic regions we are required to make adjustments to the start, end, and width columns. This is supported directly in `plyranges` via extending the `mutate()` function in `dplyr` (table 1). However, often adjustments need to be relative: there is a mutual dependence between the start, end and width columns over a genomic region. Changing the start,column needs to change either the width or the end to preserve integrity of the object. The GRanges object by design, causes a user to think of the range as a pair of endpoints, and hence we have developed operators that perform coordinate transfomations with the notion of an 'anchoring' combinator.

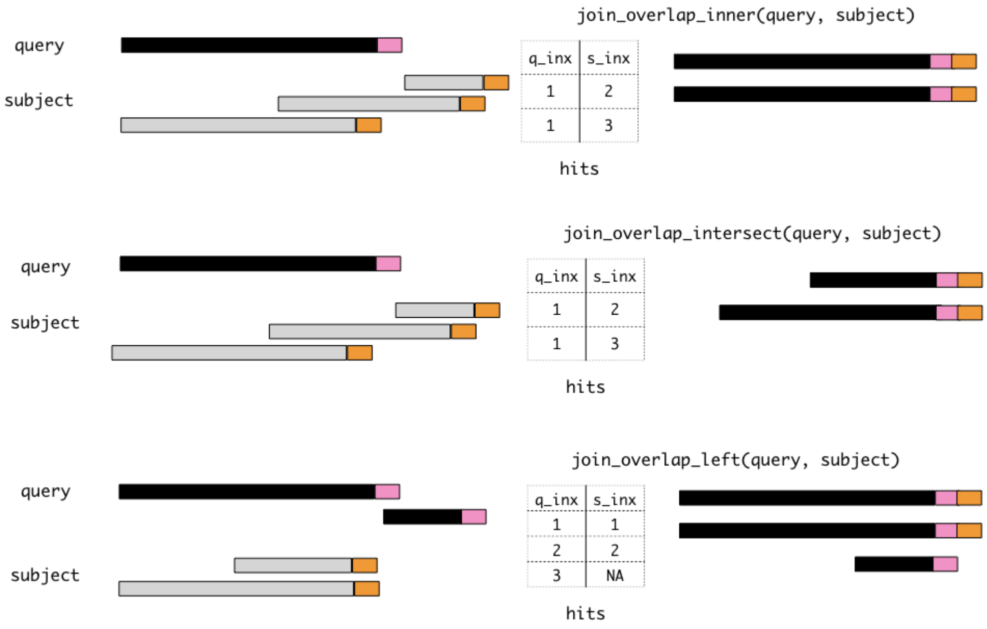
For width modification, we can stretch the range by a fixed amount or resize the width of a range. The 'anchoring' combinator preserves one of the endpoints or the midpoint of a range when altering performing width modification. It can also be used to preserve coordinates over strand, for example, anchoring over the three prime end of a genomic region, will preserve the start on the negative strand and end on the positive strand. Anchoring the width would not make sense, because it is not a point that can be intuitively anchored and stretching and resizing obviously modify the width, so it would not apply to those.

Transformations that are width invariant are designed to be explicit about the direction to modify and are performed via the shift operator. For GRanges objects strand modulates direction and we distinguish left/right (chromosomal coordinates) from the direction of transcription. To generate flanking regions, we use the flank operator which is also explicit about the direction of the input range.

Binary operations between two ranges mostly align with set operations that treat ranges as sets of integers. These are parallel/vectorized operations between two vectors/tables of ranges, and are not to be confused with the aggregating operations defined in table x. We make the binary nature of these operators by having in-fix functions in R for the parallel union, intersect and asymetric set difference. There are also operations that relate to the relative positions of two ranges such as the 'between' operator which returns the range in the gap between two ranges and the 'span' operator a range that spans both ranges. The binary operations can be expressed within the functions in table x.

Our algebra recasts the actions of finding overlaps or nearest neighbours between two genomic regions as a relational join operator. The join operator acts on two GRanges object, 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. The commonality between all join operators in the `plyranges` DSL is to use the index of where in the query range the subject range 'hits' the query range as a primary key (figure 1). The notion of 'hits' and hence the resulting key is determined by the type of join (preceding, nearest, follows overlaps).

We extend this idea by introducing three types of overlap joins: inner, intersect and left. The inner join considers a hit to be any overlap between subject and query and expands the query range to have length equal to the number of hits. The intersect join uses the same definition of a hit as the inner join but returns a range where the start and end coordinates are the intersect of coordinates in the subject that hit the query. Finally, the overlap left join is akin to left outer join in Cobb's relational algebra: it


join_overlap_inner(query, subject)
query
subject
hits
join_overlap_intersect(query, subject)
query
subject
hits
join_overlap_left(query, subject)
query
subject
hits

q_inx	s_inx
1	2
1	3

q_inx	s_inx
1	2
1	3

q_inx	s_inx
1	1
2	2
3	NA

Fig 1. Illustration of the three overlap join operators. Each join takes query and subject range as input (black and grey rectangles, respectively) with associated metadata (pink and orange rectangles respectively). An index for the join is computed via returning a Hits object, which contains the indices of where the subject overlaps the query range. This index is used to expand the query or (take the intersect) of the query ranges with subject ranges and return metadata associated with each input. This principle is gnerally applied through the 'plyranges' DSL for both overlaps and nearest neighbour operations.

performs an overlap inner join but also returns all query ranges that are not hit by the subject.

The behaviour of a join operator can be further altered with additional suffixes by restricting or expanding how a subject 'hits' a query. For example, for overlap joins we can use suffixes to encapsulate Allan's Interval Algebra: to consider subject 'hits' that are entirely 'within' a query range, the 'within' prefix is used. We also use the 'directed' suffix to explicitly include notions of strandedness in our algebra.

Cohesion

A function is cohesive if it performs a singular task and does not produce any side-effects. In the `plyranges` DSL our algebra for performing genomic arithmetic is a key example of cohesion. We define anchoring operators that decorate a GRanges object with an 'anchor' and modify the semantics of performing genomic arithmetic. The anchoring operators amount to fixing a GRanges object by its start, center, or end coordinates (or fixing these coordinates by strand). This enables any arithmetic function that performs a coordinate transformation to remain cohesive: that is they always perform the same operation regardless of whether a GRanges object is anchored or not. The only difference is the result of performing the arithmetic changes when contextual information is added to the GRanges object.

Another example of our algebra altering object semantics, while maintaining cohesion is through the 'group_by' operator. Like anchoring, this operator decorates a GRanges object with a column name (or names) that defines a partitioning of the GRanges by the unique values in the column(s). Functions defined in the `plyranges` DSL that perform restriction, aggregation, or column modification still perform those singular tasks, however grouping changes how those tasks are performed.

Consistency

A core design principle of the `plyranges` DSL is interface consistency: a user should not be surprised by the input or output of `plyranges` code. A key example of consistency is how `plyranges` handles strand information. Every 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. This strongly differs from the Bioconductor packages, which produces surprising output by assuming the user is always interested in using strand in the majority of circumstances (unless a user is computing coverage or finding flanking ranges).

Our use of suffixes in function names, also highlights are core tenet of the `plyranges` design: avoid complex generic functions with many arguments and instead use cohesive functions with a minimal number of arguments. As an example, we have written a consistent framework for reading and writing files from and to common genomic data formats as GRanges, using the `rtracklayer` package as a back-end [12]. We have replaced that packages generic functions for importing files with a family of reader functions that all take the exact same arguments.

Endomorphism

Most function calls in `plyranges` are endomorphisms: when the input is GRanges object the output will also be a GRanges object. The use of endomorphism in the `plyranges` DSL enables a user to predict the structure of the output of their computations and does not require them to learn any additional classes beyond GRanges and DataFrames.

- compute coverage as an example here

This design pattern strongly deviates from the design of the OO interface in `GenomicRanges`, where many methods return a new class that is unfamiliar to the user upon return. These low-level classes enable efficient computing at the cost of complexity and are abstracted away in `plyranges` interface.

Fluency

As a consequence of the design principles defined above every function in `plyranges` performs a single action on GRanges objects. The `plyranges` DSL implements the core verbs from the `dplyr` package and implements a genomic relational algebra for transforming GRanges objects (table x, y). Each verb preserves the semantics of GRanges object and works with derivatives of the GRanges class. 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 function names that are nouns.

This approach strongly contrasts the `GenomicRanges/IRanges` OO interface, which emphasises the use of setter and getter methods. In that interface, core components and metadata are updated via replacement methods (requiring knowledge of the class components), while our interface requires only a single call to `mutate()` to perform the modification.

Workflows can be composed by chaining verbs together into 'sentences' via the forward pipe operator,`%>%` (exported from the R package `magrittr` [13]), which can be read as the word 'then'. Overall, this allows users to write human readable code because workflows describe what the code is doing rather than how its doing it.

Opportunities

A caveat to constructing a compatible interface with `dplyr` is that `plyranges` makes extensive use of non-standard evaluation in R (achieved via the `rlang` package [14]). Simply, this means that computations are performed and evaluated in the context of the GRanges objects; emphasising the interactive nature of our API. Consequently, when programming with `plyranges` a user needs to be aware of how non-standard evaluation in R works and how to adapt their code accordingly. However, with the rise of R packages like `rlang` this process is becoming less difficult.

While GRanges are an intuitive representation for data measured on genomic regions, more flexible data structures are required to represent data from multiple sample experiments. The Bioconductor class SummarizedExperiment is the canonical data structure for representing data for combining multiomic measurements from multiple samples. The grammar and design of the `plryanges` DSL can be naturally extended to the SummarizedExperiment.

Results

Here we provide examples on how to use the `plyranges` DSL to construct genomic data workflows and highlight its interoperability with existing Bioconductor packages.

Peak Finding

The Bioconductor package `AnnotationHub` [15] can be used to search for BigWig files from ChIP-Seq experiments from the Human Epigenome Roadmap project [16]. Here we focus on assays for primary T CD8+ memory cells from peripheral blood. Using `plyranges` we will 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")
```

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 the regions that were reduced.

```
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 the width.

```
chr10_max_score_region <- chr10_scores %>%
  filter(score == max(score)) %>%
  anchor_center() %>%
  set_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 2).

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

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 (SNP) and short insertion/deletions (indel). 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

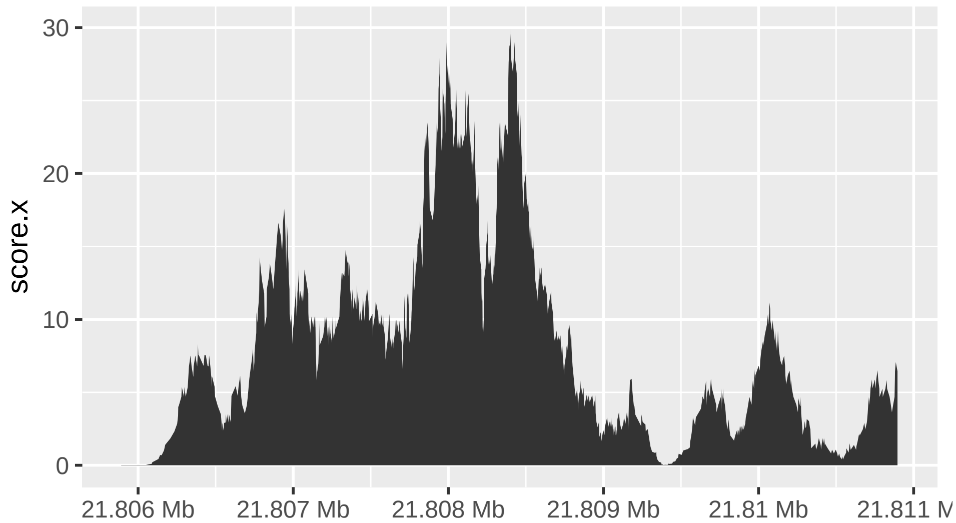


Fig 2. Visualisation of normalised coverage scores accross a 5000nt region of chromosome 10 from H3K27Me3 ChIP-Seq assay from the Human Epigenome Roadmap project.

to compute the transition-transversion ratio, a quality control metric, within each chromosome in GRanges object.

First we filter out any insertion or deletion alleles and the SNPs present on the mitochondria then 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 can then compute the transition-transversion ratio using the `group_by` and `summarise` pattern, it is computed as the total number of transition SNPs divided by the total number of transversion SNPs within each chromsome (figure 3).

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

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. Continuing with the data from the Human Epigenome Roadmap Consortium data, we can count the number of reads that fall into a fixed bins of size 10000bp over a BAM file of H3K27Me3 methylation marks from the H1 cell line. We extract reads that have a mapping quality score greater than 20:

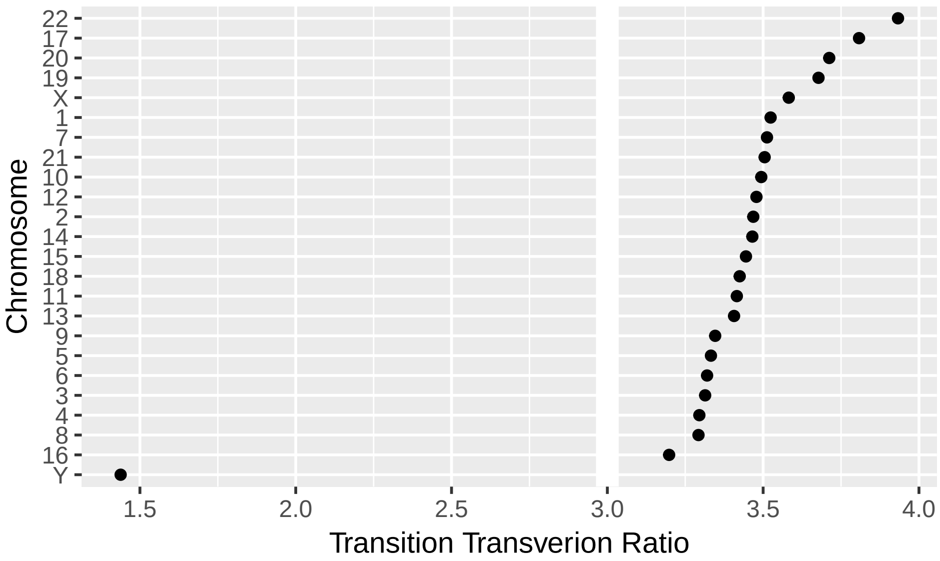


Fig 3. Dot plot of chromosomes ordered by estimated transition-transversion ratio. A white reference line is drawn at the expected ratio for a human exome.

```
alignments <- read_bam(h1_bam) %>%
  filter(mapq > 20)
```

Note that the BAM file is only read into memory once we perform an operation on it. Next we generate our genomic bins using the `tile` function from `GenomicRanges`:

```
bins <- tile(h1_bam, width = 10000)
```

Finally, we can use `group_by_overlaps` with `summarise` to compute the total number of reads within each window.

```
h1_bam_read_summary <- h1_bam %>%
  group_by_overlaps(bins) %>%
  summarise(n_reads = n())
```

https://support.bioconductor.org/p/71601/ also useful example?

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 to prepare data for the visualisation of multimodal biological datasets.

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` [17] and the figures were made with `ggbio` [18]. All code required to reproduce this report is available at https://github.com/sa-lee/plyranges-paper.

References

1. 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

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

3. 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

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

5. R Core Team. R: A language and environment for statistical computing [Internet]. Vienna, Austria: R Foundation for Statistical Computing; 2018. Available: `https://www.R-project.org/`

6. 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

7. 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

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. Riemondy KA, Sheridan RM, Gillen A, Yu Y, Bennett CG, Hesselberth JR. Valr: Reproducible genome interval arithmetic in r. F1000Research. 2017; doi:10.12688/f1000research.11997.1

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

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

12. 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

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

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

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

17. Xie Y. Dynamic documents with R and knitr [Internet]. 2nd ed. Boca Raton,

Florida: Chapman; Hall/CRC; 2015. Available: https://yihui.name/knitr/

18. Yin T, Cook D, Lawrence M. Ggbio: An r package for extending the grammar of graphics for genomic data. Genome Biology. BioMed Central Ltd; 2012;13: R77.