RNA-seq Analysis 1

Introduction to Bioconductor

Pietà Schofield

Contents

Rmarkdown

Donald Knuth coined the phrase literate programming (Knuth, 1984; Knuth, 1992) and he made a distinction

- **Literate proramming** embeds the code within the natural language description of the logic behind the code

versus

- **Documentation generation** structured comments embedded in the code are extracted to produce documentation

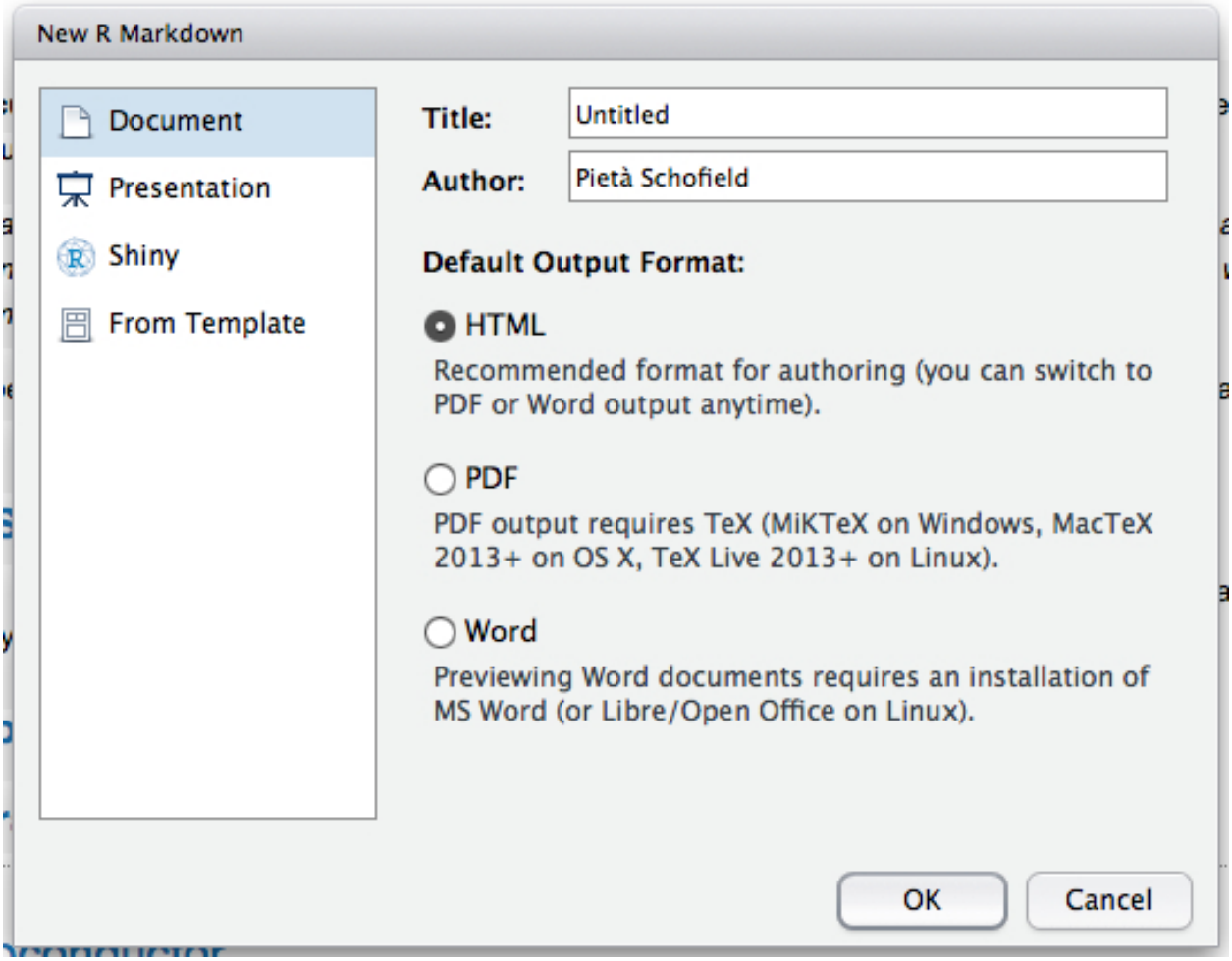
"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do." Knuth (1984)

This can be particularly useful way of looking at things when you are using a computer to perform data analysis.

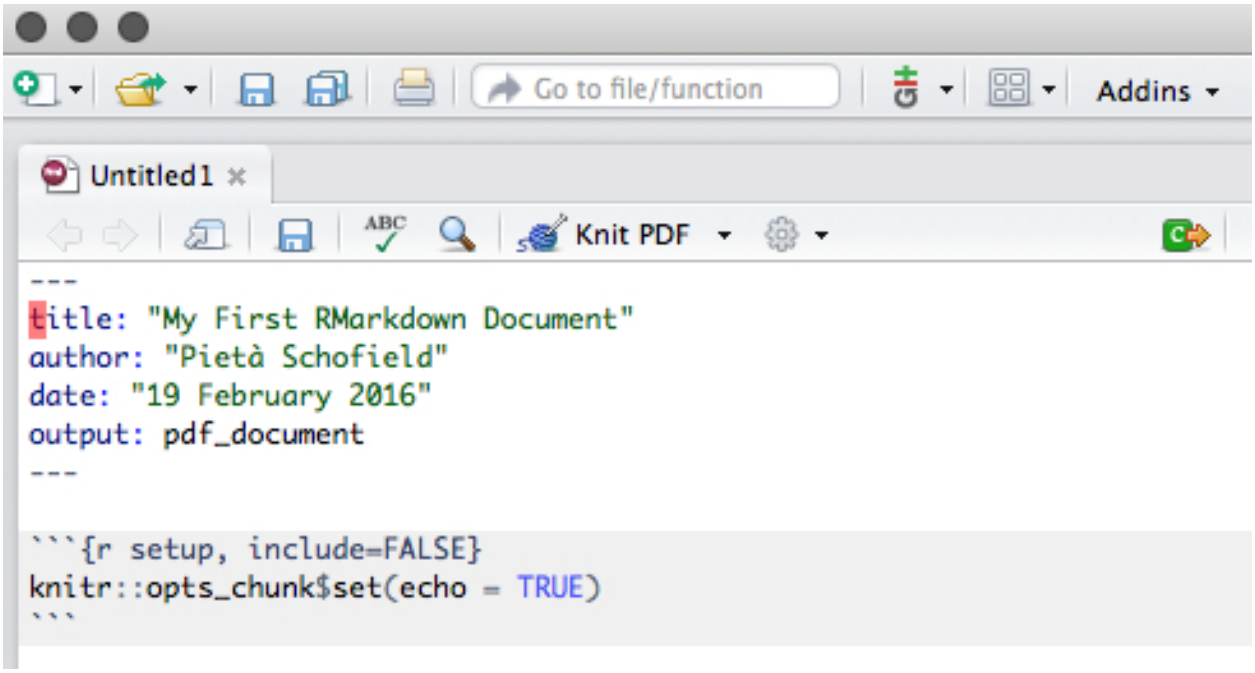
RStudio

RStudio is designed to make literate programming simple. It has a few options but the main two are provided by the packages `sweave` and `knitr`, which enable the writing of documents in a markup language such as HTML , LaTeX or markdown, with embedded code chunks, that can then be compiled into a final document in a range of formats such as portable document format (PDF), Microsoft Word (docx) or HTML. During the compilation process code chunks are executed and output such as figures are generated and incorporated into the final document.

In this course we will focus of Rmarkdown there is extensive documentation available as http://rmarkdown.rstudio.com. You can start a new `rmarkdown` document from with-in RStudio from the **File - New File - R Markdown...** menu option that opens the dialogue box below



Select Document option on the left hand side and the PDF option on the right hand side, fill in a title of your choice and your name and a to create a demo rmarkdown document, that will open in the RStudio editor panel. Without making any changes to this file you can compile it into a PDF by clicking the **Knit PDF** button on the editor panel button bar. You will be prompted to save the file, so you should chose and name and location for the file.



If you examine the file in the editor you will see it is structured in a particular ways.

YAML header

In typical recursive geek speak YAML is an acronym that stands for **Y**AML **A**in't **M**arkup **L**anguage. It is a header block of parameters used through out the document compilation process. It uses a standard format for many programming and markup languages, though the permissable option vary. It is separated from the rest of the document being surrounded by three dashes `---`

For example the YAML block for this document is

```
---
title: 'RNA-seq Analysis 1: Introduction to Bioconductor'
author: "Pietà Schofield"
output:
  html_document:
    fig_caption: yes
    toc: yes
    toc_depth: 2
    toc_float:
      collapsed: no
      smooth_scroll: no
    code_folding: show
    css: workshop.css
---
```

Code Chunks

Sections of R (and other language) code can be embedded in the document surrounded by the three back tick delimites as in the example below.

```
```{r chuckName, fig.caption="Some Random Stuff", eval=FALSE}
#
# Plot anything R can plot
#
plot(1:10+rnorm(10),1:10+rnorm(10),pch="x",
     xlab="expected", ylab="measured",main="Demo Plot")
#
# One option is to just send it to the default device and knitr
# captures it and put it in a temporary place
#
abline(0,1,col="red")
#
# alternatively write it to a file and link to the file in the markdown
#
```
```

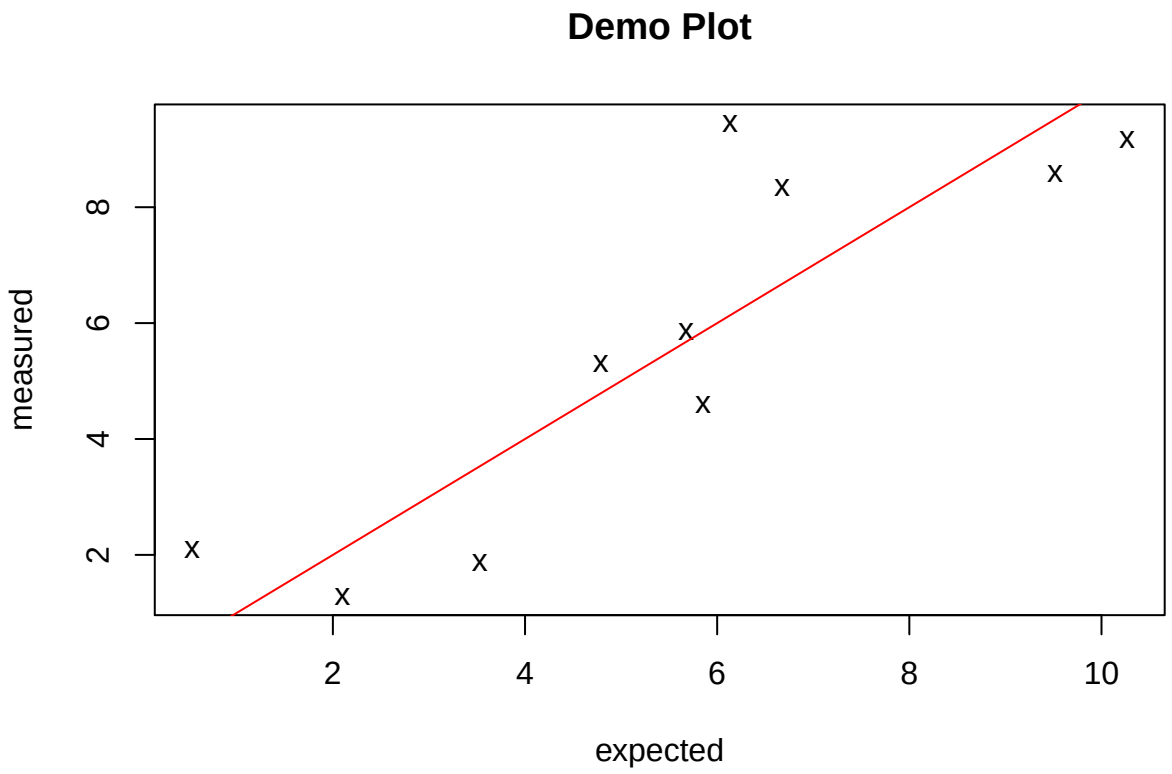
The first line holds paramters or **chunk options** between the parentheses that control how the chunk will be processed during compilation and how the output from the chuck if it is executed will be incorporated and formated in the document. For example whether the chunk is executed is controlled by the **eval** option and whether the chunk is displayed in the output document is controlled by the **echo** option

Markdown Text

The rest of the document is in markdown and certain codes will produce various formatting styles in the output document. For example various numbers of # characters at the start of a line control heading levels * or _ will control italic and bold text.

Graphics

The code chunk above produces a graph in the output. It is possible to change the code in the chunk to change the figure. It is also possible to set chunk options so that only the graph is displayed and not the code, or only the code and not the graph.



Exercise 1

Create an rmarkdown document that will compile to PDF.

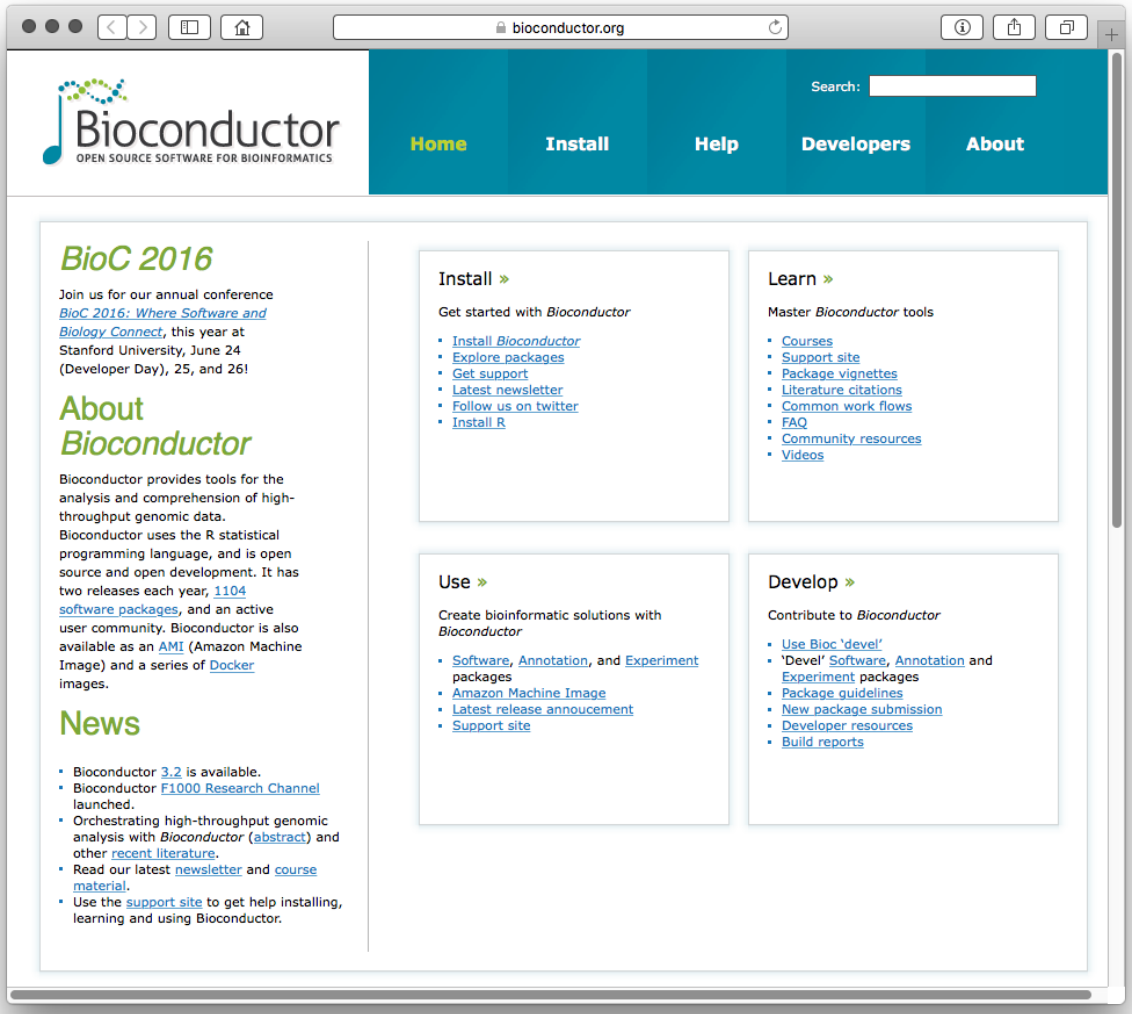
- Alter the document to hide the code that displays the summary of the car dataset.
- Alter the document to add a table of contents.
- Add a code chunk to the document to use the `head()` function to display the first 10 items in the pressure data frame.
- Add a final section heading and code chunk using the `sessionInfo()` function to include the R configuration information in the document.

Bioconductor

Bioconductor (Huber, W., Carey, et al., 2015) is a curated repository of open source packages for the R statistical environment specifically aimed at biological information processing. There are three main strands of packages

- Software packages that contain libraries of functions and data object classes for the processing and statistical analysis of biological data
- Annotation packages that contain data files that hold curated biological information databases for example lists of genes in a species, or the DNA sequence of an organism.
- Experimental Data packages that contain the results files in various states of processing and analysis form biological experiment.

The project has a website which is well worth exploring to discover the range of available packages.



Installation

The initial installation of bioconductor

```
source("http://bioconductor.org/biocLite.R")
```

```
biocLite("BiocInstaller")
```

Subsequently you can install packages with the BiocInstaller package

```
require(BiocInstaller)
biocLite("NameOfPackage")
```

Core Data Classes

One of the most useful things Bioconductor provides is a collection of common data classes that have storage structures and processing methods that as used extensively throughout the library.

Biostrings

The Biostrings package (Pages, Aboyoun, Gentleman, et al., 2016) provides several classes for the storage of sequences of single characters. It provides a general virtual class XString for storing big string sequences of any character. It provides derived classes [BString], [DNAString], [RNAString] and [AAString] for general strings, DNA sequences, RNA sequences and Amino Acid sequences such as peptides and proteins. I also provides a class for collections or sets of such objects derived from the [XStringSet] class. It also provides classes for holding multiple sequence alignments and masked and quality scaled sequences and sets of aligned sequences.

Also provided are several standard constants such as `DNA_ALPHABET` with the full set of IUPAC letters for DNA, and `DNA_BASES` that contains just the unambiguous base alphebet.

```
require(Biostrings)
DNA_ALPHABET
```

```
 [1] "A" "C" "G" "T" "M" "R" "W" "S" "Y" "K" "V" "H" "D" "B" "N" "-" "+"
[18] "."
```

```
DNA_BASES
```

```
[1] "A" "C" "G" "T"
```

So for example we can generate a vector of 10 bases chosen at random from the `DNA_BASES` alphabet using the `sample()` function

```
# sample the alphabet
vcDNA <- sample(DNA_BASES,10,replace=T)
str(vcDNA)
```

```
 chr [1:10] "C" "A" "A" "T" "C" "A" "T" "A" "A" ...
```

The constructor for an object of the DNAString class wants a single string of chararters not a vector of single characters so we can use the `paste()` function to turn the vector of bases to a single string

```
stDNA <- paste(vcDNA,collapse="")
str(stDNA)
```

```
 chr "CAATCATAAA"
```

We can then use the `DNAString()` constructor function to make a DNAString object.

```
DNA <- DNAString(stDNA)
str(DNA)
```

```
Formal class 'DNAString' [package "Biostrings"] with 5 slots
  ..@ shared         :Formal class 'SharedRaw' [package "XVector"] with 2 slots
  .. .. ..@ xp                    :<externalptr>
  .. .. ..@ .link_to_cached_object:<environment: 0x7fce15ed23b8>
  ..@ offset         : int 0
  ..@ length         : int 10
  ..@ elementMetadata: NULL
  ..@ metadata       : list()
```

The `str()` function here reveals the structure of the DNA objects you can see it is of class `DNAString` and you can see the name of the package that defined this class. It also has 5 `slots`, there are the objcts that appear in the indented list starting with an `@` sign.

You can retrieve the content of these slots with the syntax *objectname* @ *slotname* .

```
DNA@shared
```

```
SharedRaw of length 10 (data starting at address 0x7fce1b18aad8)
```

In the case of XString objects you can also retrieve the sequence as a character string using the casting function `as.character()`

```
seq <- as.character(DNA)
seq
```

```
[1] "CAATCATAAA"
```

As well as the `str()` function used above we can use the `is()` function to see what type of thing R knows an object to be.

```
is(DNA)
```

```
[1] "DNAString" "XString"   "XRaw"      "XVector"   "Vector"    "Annotated"
```

Biostrings provides methods for reading and writing common file formats containing sequences searching such as FASTA and FASTQ, for searching with pattern and position weight matrices, and for pairwise sequence alignments.

```
bigSeq <- DNAString("AGCTGGTCGATGACGTTAGGATCAGTACGATTGAGGCAGGGTCAGTAATTGGATTGAGGATTGACCCCGATG")
smallSeq <- DNAString("TCAGTA")
pos <- matchPattern(smallSeq,bigSeq)
pos
```

```
  Views on a 72-letter DNAString subject
subject: AGCTGGTCGATGACGTTAGGATCAGTACGATT...TCAGTAATTGGATTGAGGATTGACCCCGATG
views:
    start end width
[1]    22  27     6 [TCAGTA]
[2]    42  47     6 [TCAGTA]
```

Execise 2

- What kind of object is the pos object above (the result of the `patternMatch()` call)
  - use the `str()` function to view its structure.
  - recover content of the ranges slot
  - use the `is()` function discover the full list of object types and classes the ranges slot belongs to

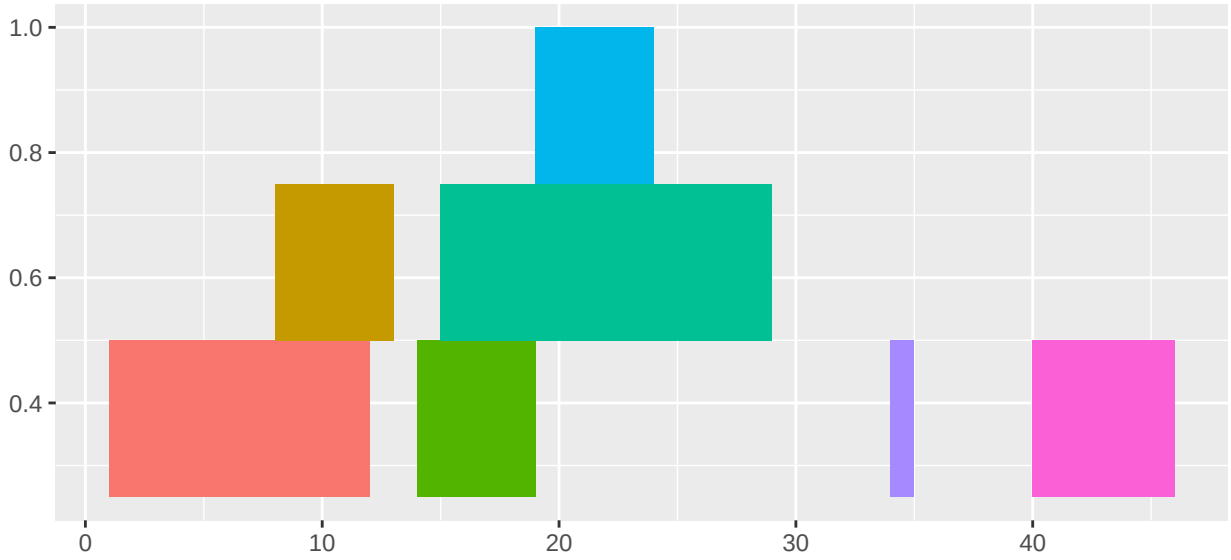
IRanges

The IRanges package (Lawrence, Huber, Pagès, et al., 2013) provides a way of storing, manipulating and calculating with interval ranges. Interval ranges represent regions of a sequence.

You can construct an *IRange* object with the `IRanges()` constructor, by specifying either of two lists,

- start and end positions
- start positions and widths

```
library(IRanges)
ir <- IRanges(start = c(1, 8, 14, 15, 19, 34, 40),
              width = c(12, 6, 6, 15, 6, 2, 7))
plotRanges(ir)
```



The IRanges package provides functions for manipulating and analysing IRanges objects, for example there is a function `overLapsAny()` that will tell you which ranges in an IRanges object overlap

```
overLaps <- overlapsAny(ir)
overLaps
```

```
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

and exactly which the overlap with

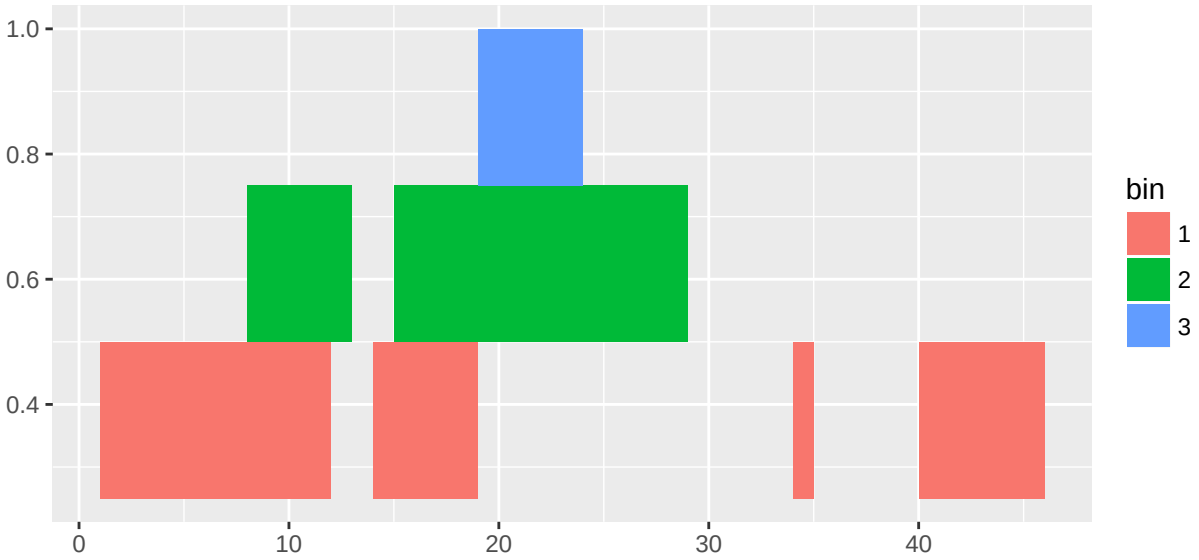
```
overLaps <- findOverlaps(ir)
overLaps
```

```
Hits object with 15 hits and 0 metadata columns:
       queryHits subjectHits
       <integer>   <integer>
   [1]         1           1
   [2]         1           2
   [3]         2           1
   [4]         2           2
   [5]         3           3
   ...       ...         ...
  [11]         5           3
  [12]         5           4
  [13]         5           5
  [14]         6           6
  [15]         7           7
  -------
  queryLength: 7
  subjectLength: 7
```

We can split the ranges in to groups (or bins) of non-overlapping (or disjoint) ranges

```
bins <- disjointBins(ir)
bins
```

```
[1] 1 2 1 2 3 1 1
```



GenomicRanges

The GenomicRanges package (Lawrence, Huber, Pagès, et al., 2013) makes use of a compression technique called Run Length Encoding(RLE) to store extra information with interval ranges to hold data such as genomic locations

Run Length Encoding

```
# generate a repetative sequence
mySeq <- unlist(sapply(rnorm(10,5,1),
                  function(n){
                    rep(sample(DNA_BASES,1),n)
                  }))
# what does it look like
mySeq
```

```
 [1] "A" "A" "A" "A" "A" "A" "T" "T" "T" "T" "C" "C" "C" "C" "C" "C" "C"
[18] "C" "C" "C" "A" "A" "A" "A" "T" "T" "T" "T" "T" "T" "A" "A" "A" "C"
[35] "C" "C" "C" "C" "A" "A" "A" "T" "T" "T" "T" "T"
```

```
# how big is it
paste0("Size of mySeq ",format(object.size(mySeq),units = "Kb"))
```

```
[1] "Size of mySeq 0.5 Kb"
```

```
# turn it into an RLE object
myRLE <- Rle(mySeq)
# what does that look like
myRLE
```

```
character-Rle of length 46 with 9 runs
  Lengths:   6   4  10   4   6   3   5   3   5
  Values : "A" "T" "C" "A" "T" "A" "C" "A" "T"
```

```
# how big is that
paste0("Size of myRLE ",format(object.size(myRLE),units = "Kb"))
```

```
[1] "Size of myRLE 1.4 Kb"
```

Hmmmm! not much advantage "I thought you said it was a compression technique", well if you a looking at longer sequences with more repeats

```
# generate a much bigger repetative sequence
mySeq <- unlist(sapply(rnorm(100,10,2),
                  function(n){
                    rep(sample(DNA_BASES,1),n)
                  }))
paste0("Size of mySeq ",format(object.size(mySeq),units = "Kb"))
```

```
[1] "Size of mySeq 7.7 Kb"
```

```
# and now the RLE
myRLE <- Rle(mySeq)
paste0("Size of myRLE ",format(object.size(myRLE),units = "Kb"))
```

```
[1] "Size of myRLE 2.1 Kb"
```

GenomicRanges (also called GRanges) are excelent for genomic annotations. The minimal requirements for a GRanges object are an RLE object with sequences names and a set of ranges associated with the seqences in the form of an IRanges object. The GRanges object also contains a slot for strand (sense on the DNA), however this defaults to * which represents unspecified if not specifically given.

```
grObj <- GRanges(seqnames=Rle(c(rep("seq1",4),rep("seq2",2),"seq3")),
                 ranges = IRanges(start = c(110, 200 , 250, 350, 100, 300, 40),
                                  width = c(50, 40, 60, 150, 60, 200, 70)))
grObj
```

```
GRanges object with 7 ranges and 0 metadata columns:
      seqnames     ranges strand
         <Rle>  <IRanges>  <Rle>
  [1]     seq1 [110, 159]      *
  [2]     seq1 [200, 239]      *
  [3]     seq1 [250, 309]      *
  [4]     seq1 [350, 499]      *
  [5]     seq2 [100, 159]      *
  [6]     seq2 [300, 499]      *
  [7]     seq3 [ 40, 109]      *
  -------
  seqinfo: 3 sequences from an unspecified genome; no seqlengths
```

GRanges objects can store other information along with this basic data, typically they are used for information in GTF and BED annotation files. The package `rtracklayer` has many functions for importing and exporting data from these file formats

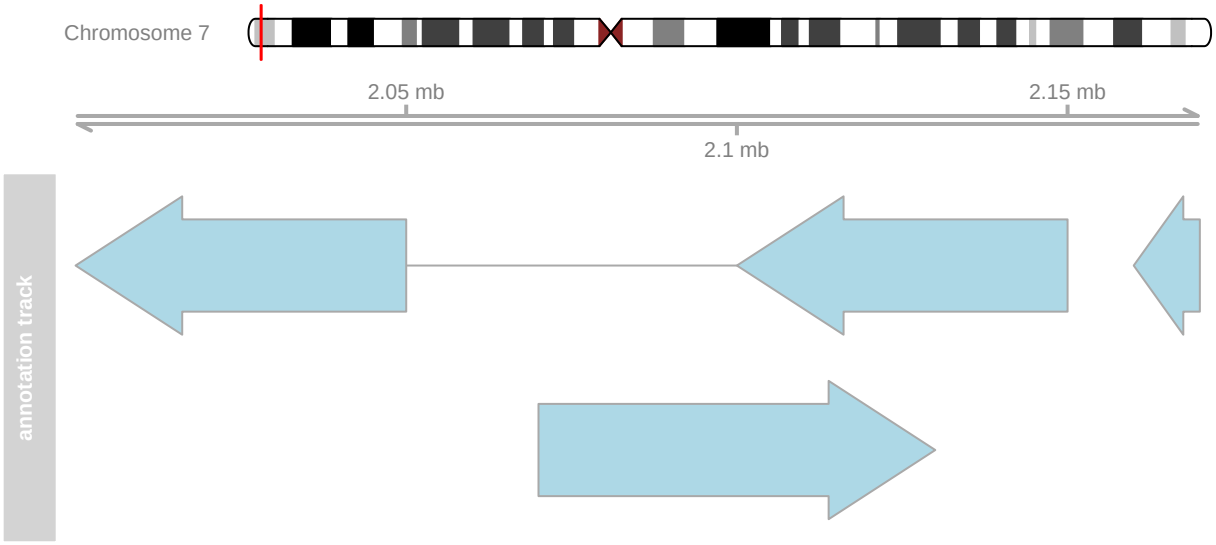
```
require(rtracklayer)
track <- import(system.file("tests", "v1.gff", package = "rtracklayer"))
export(track, "my.gff3", "gff3")
track
```

```
GRanges object with 3 ranges and 5 metadata columns:
      seqnames             ranges strand |   source     type     score
         <Rle>          <IRanges>  <Rle> | <factor> <factor> <numeric>
  [1]    chr21 [1000000, 1011000]      + | TeleGene enhancer       500
  [2]    chr22 [1010000, 1010100]      + | TeleGene promoter       900
  [3]    chr22 [1020000, 1020000]      - | TeleGene promoter       800
          phase    group
      <integer> <factor>
  [1]      <NA>   touch1
  [2]      <NA>   touch1
  [3]      <NA>   touch2
  -------
  seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

R has a few packages that are designed for visualiation of genomic range data. Two of the more flexible are the package `Gviz` and the package `ggbio`.

For example we can plot use Gviz to plot some mocked up annotation

```
require(Gviz)
# if we mock up some data
grObj <- GRanges(seqnames=Rle(rep("7",4)),
                 ranges=IRanges(start=c(2000000, 2070000, 2100000, 2160000),
                                end=c(2050000, 2130000, 2150000, 2170000)),
                 strand=c("-", "+", "-", "-"),
                 group=c("Group1","Group2","Group1", "Group3"))
# make and annotation track
annTrack <- AnnotationTrack(grObj, genome="hg19", feature="test",
                            id=paste("annTrack item", 1:4),
                            name="annotation track",
                            stacking="squish")
# make and axis track
axis <- GenomeAxisTrack()
# and an ideogram (chromosome map track)
ideo <- IdeogramTrack(genome="hg19",chromosome=7)
## Now plot the tracks
res <- plotTracks(list(ideo, axis, annTrack))
```



GenomicAlignments

There are several packages in bioconductor that are specifically designed for the handling of short read data, in the form of FASTQ raw files and alignment files such as BAM and SAM files. There are differences in the underlying data structures they use and the functions the provide.

The `ShortRead` package (Morgan, Anders, Lawrence, et al., 2009) is particularly suited for processing reads in FASTQ files and provides functions for sampling and QC tasks, it imports the reads into `Biostring` objects. It also contains some functions for single ended alignments. However, the packages `GenomicAlignments` (Lawrence, Huber, Pagès, et al., 2013) provides much more extensive support for paired end and gapped alignments in BAM files. It uses the package `Rsamtools` to import and export alignment data files.

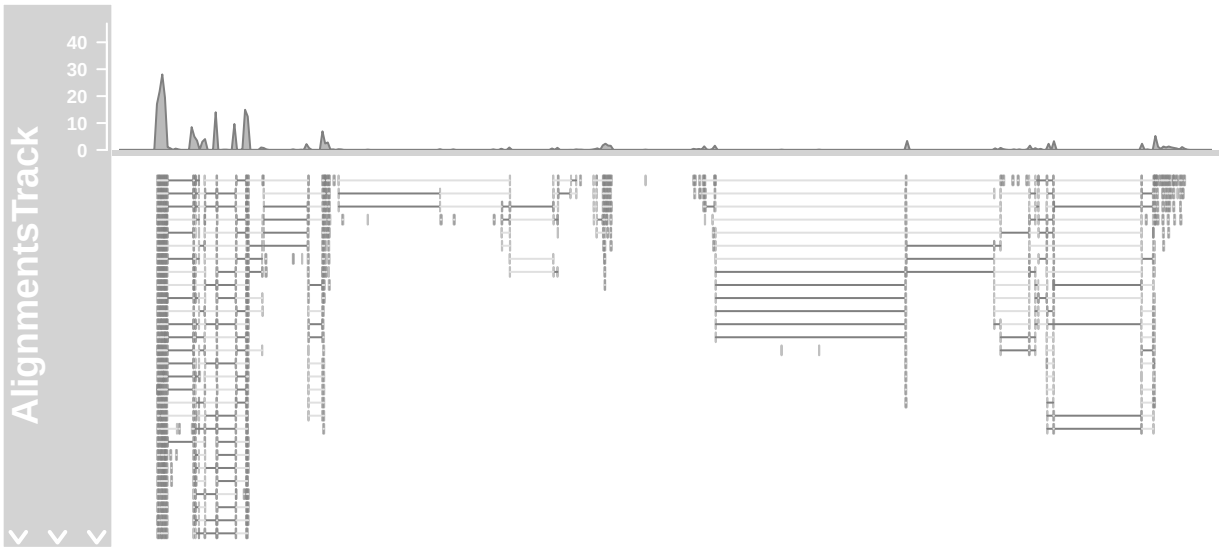
When alignments are read in the reads are held in GRanges objects with all the functions of the GenomicRanges and IRanges packages available for processing.

```
alignFile <- system.file(package="Gviz", "extdata", "gapped.bam")
gaData <- readGAlignmentPairs(alignFile)
gaData
```

```
GAlignmentPairs object with 1074 pairs, strandMode=1, and 0 metadata columns:
         seqnames strand   :             ranges  --             ranges
            <Rle>  <Rle>   :          <IRanges>  --          <IRanges>
     [1]    chr12      +   : [2966852, 2966901]  -- [2966993, 2967042]
     [2]    chr12      +   : [2966856, 2966905]  -- [2966963, 2967012]
     [3]    chr12      +   : [2966856, 2966905]  -- [2967024, 2967073]
     [4]    chr12      +   : [2966859, 2966908]  -- [2967003, 2967052]
     [5]    chr12      +   : [2966859, 2966908]  -- [2967010, 2967059]
     ...      ...    ... ...                ... ...                ...
  [1070]    chr12      +   : [3154300, 3154349]  -- [3154446, 3154495]
  [1071]    chr12      -   : [3154359, 3154408]  -- [3154244, 3154293]
  [1072]    chr12      -   : [3154599, 3154648]  -- [3154459, 3154508]
  [1073]    chr12      -   : [3154743, 3154792]  -- [3154597, 3154646]
  [1074]    chr12      -   : [3155131, 3155180]  -- [3154990, 3155039]
  -------
  seqinfo: 1 sequence from an unspecified genome
```

`Gvis` also has functions to make plotting of alignment data relatively painless.

```
atrack <- AlignmentsTrack(alignFile,isPaired=T)
plotTracks(atrack,from=2960000,to=3160000,chromosome="chr12")
```



Data Import Export

Bioinformatics is settling on a fairly steady set of file formats, FASTA and FASTQ for sequence data for example and SAM/BAM for NGS short read alignment data. While many of packages mentioned in this workshop possess functions for importing and exporting data that the package data structures are designed to handle they often don't possess functions for other files. There are also some standard formats for processed data for example read depths, peak locations and genetic annotations. This type of data often appears as track in a genome browser.

rtracklayer

The package rtracklayer (Lawrence, Gentleman, and Carey, 2009) has two useful generic functions `import()` and `export()` that will handle several track formats including FASTA, GTF, BED, WIG.

Learning Bioconductor

Bioconductor provides many resources for assisting in learning how to do various analyses. It has a very active support forum where many bioconductor uses and many of the package authors will answer questions regarding the packages.

Package Vignettes

All Bioconductor packages are supposed to come with a user manual which is a concise list of the functions and data structures available in the package listed alphabetically and with all the function parameters documented. They are also expected to have examples for all the functions and at least one vignette document that is meant to be a more recipe, How-to, document. You can list the names of the available vignettes with the command `vignette()`

```
vignette(package="Biostrings")
```

You can then view the vignette with the same command only this time passing the name of the vignette you want.

```
vignette("BiostringsQuickOverview")
```

Workflows

One of the ways this is done is via workflows

Tutorials

The support forum has a specific section of links to tutorial

Bibliography

[1] Huber, W., Carey, et al. “Orchestrating high-throughput genomic analysis with Bioconductor”. In: Nature Methods 12.2 (2015), pp. 115–121.

[2] D. E. Knuth. “Literate Programming”. In: Comput. J. 27.2 (1984), pp. 97–111.

[3] D. E. Knuth. Literate Programming. Cambridge: Cambridge University Press, 1992.

[4] M. Lawrence, R. Gentleman and V. Carey. “rtracklayer: an R package for interfacing with genome browsers”. In: Bioinformatics 25 (2009), pp. 1841–1842.

[5] M. Lawrence, W. Huber, H. Pagès, et al. “Software for Computing and Annotating Genomic Ranges”. In: PLoS Computational Biology 9 (2013).

[6] M. Morgan, S. Anders, M. Lawrence, et al. "ShortRead: a bioconductor package for input, quality assessment and exploration of high-throughput sequence data." In: Bioinformatics (Oxford, England) 25.19 (Oct. 01, 2009), pp. 2607–2608. ISSN: 1367-4811 1367-4803. DOI: 10.1093/bioinformatics/btp450. pmid: pmid.

[7] H. Pages, P. Aboyoun, R. Gentleman, et al. Biostrings: String objects representing biological sequences, and matching. R package. version 2.38.4, 2016.

Session Info

```
R version 3.2.3 (2015-12-10)
Platform: x86_64-apple-darwin13.4.0 (64-bit)
Running under: OS X 10.11.3 (El Capitan)

locale:
[1] en_GB.UTF-8/en_GB.UTF-8/en_GB.UTF-8/C/en_GB.UTF-8/en_GB.UTF-8

attached base packages:
 [1] grid      stats4    parallel  stats     graphics  grDevices utils
 [8] datasets  methods   base

other attached packages:
 [1] Gviz_1.14.4                rtracklayer_1.30.2
 [3] GenomicAlignments_1.6.3    Rsamtools_1.22.0
 [5] SummarizedExperiment_1.0.2 Biobase_2.30.0
 [7] GenomicRanges_1.22.4       GenomeInfoDb_1.6.3
 [9] Biostrings_2.38.4          XVector_0.10.0
[11] IRanges_2.4.8              S4Vectors_0.8.11
[13] BiocGenerics_0.16.1        BiocInstaller_1.20.1
[15] knitr_1.12.3               RefManageR_0.10.6
[17] pietalib_0.1

loaded via a namespace (and not attached):
 [1] VariantAnnotation_1.16.4 splines_3.2.3
 [3] lattice_0.20-33          colorspace_1.2-6
 [5] htmltools_0.3            GenomicFeatures_1.22.13
 [7] yaml_2.1.13              XML_3.98-1.3
 [9] survival_2.38-3          foreign_0.8-66
[11] DBI_0.3.1                BiocParallel_1.4.3
[13] RColorBrewer_1.1-2       lambda.r_1.1.7
[15] matrixStats_0.50.1       plyr_1.8.3
[17] stringr_1.0.0            zlibbioc_1.16.0
[19] munsell_0.4.3            gtable_0.2.0
[21] futile.logger_1.4.1      evaluate_0.8
[23] labeling_0.3             latticeExtra_0.6-28
[25] biomaRt_2.26.1           AnnotationDbi_1.32.3
[27] Rcpp_0.12.3              acepack_1.3-3.3
[29] BSgenome_1.38.0          scales_0.4.0
[31] formatR_1.2.1            Hmisc_3.17-2
[33] gridExtra_2.2.0          ggplot2_2.0.0
[35] digest_0.6.9             stringi_1.0-1
[37] biovizBase_1.18.0        RJSONIO_1.3-0
[39] bibtex_0.4.0             tools_3.2.3
```

```
[41] bitops_1.0-6            magrittr_1.5
[43] RCurl_1.95-4.7          RSQLite_1.0.0
[45] dichromat_2.0-0         Formula_1.2-1
[47] cluster_2.0.3           futile.options_1.0.0
[49] lubridate_1.5.0         rmarkdown_0.9.5
[51] rpart_4.1-10            nnet_7.3-12
```