Introduction to Bioconductor

David Wheeler, CCR, NCI

1/29/2015

Contents

"Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data. Bioconductor uses the R statistical programming language, and is open source and open development. It has two releases each year, 936 software packages, and an active user community."

These 936 packages are searchable at:

http://bioconductor.org/packages/release/BiocViews.html

Bioconductor Packages by Category

Software (936)	AnnotationData (895)	ExperimentData (223)
AssayDomain (299)	ChipManufacturer (374)	Cancer (35)
BiologicalQuestion (261)	ChipName (195)	CGH
Infrastructure (186)	CustomArray (2)	ChIPchipData (1)
ResearchField (193)	CustomCDF (16)	ChIPseqData (4)
StatisticalMethod (261)	CustomDBSchema (11)	EColiData (1)
Technology (591)	FunctionalAnnotation (13)	ExpressionData (4)
WorkflowStep (477)	Organism (532)	FlowCytData
	PackageType (524)	HapMap (7)
	SequenceAnnotation (3)	HighThroughputSequencingData (9)

Software (936)	AnnotationData (895)	ExperimentData (223)
		HIV (1)
		MassSpectrometryData (7)
		MethylseqData
		NormalTissue (3)
		Proteome
		QualityAndTesting
		RNAExpressionData (16)
		RNAseqData (19)
		SpikeIns
		StemCells (1)
		Yeast (10)

Installing Bioconductor

```
source("http://www.bioconductor.org/biocLite.R") # source the R coded installer

biocLite() # run the installer to get the basic package
```

Biobase–the Foundation Library

Functions Used

- `data()` # see the available sample data and load some
- `library()` # load an installed package into the current session
- `ls()` # list the objects available in the current session–analogous to Unix *ls*
- `table()` # tabulate counts for the values in an object

Upon loading *Biobase*, a summary of the package is displayed:

```
library(Biobase)  #load your installed copy of the Biobase library
```

```
## Loading required package: BiocGenerics
## Loading required package: parallel
##
## Attaching package: 'BiocGenerics'
##
## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB
##
## The following object is masked from 'package:stats':
##
```

```
##     xtabs
##
## The following objects are masked from 'package:base':
##
##     anyDuplicated, append, as.data.frame, as.vector, cbind,
##     colnames, do.call, duplicated, eval, evalq, Filter, Find, get,
##     intersect, is.unsorted, lapply, Map, mapply, match, mget,
##     order, paste, pmax, pmax.int, pmin, pmin.int, Position, rank,
##     rbind, Reduce, rep.int, rownames, sapply, setdiff, sort,
##     table, tapply, union, unique, unlist, unsplit
##
## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with
##     'browseVignettes()'. To cite Bioconductor, see
##     'citation("Biobase")', and for packages 'citation("pkgname")'.
```

What do we learn from the summary?

- In addition to *Biobase*, two other packages upon which Biobase depends are automatically loaded. These are *BiocGenerics* and *parallel*.
- A number of functions from other packages are "masked". This means that a recently loaded library defines a function of the same name as that of a previously loaded library and the new function now supersedes the old. This is not usually a problem because the new function is generally very similar to the old.
- Vignettes for a package take you through a typical analysis and are a very good way to get acquainted with the package.

***Biobase* comes with Sample Data** Another way to get acquainted with a new package is to load any sample data it may provide. To see the sample data use the `data()` function.

```
data(package = "Biobase")  # see example data sets from package Biobase
```

```
Data sets in package 'Biobase':

SW                      Class to Contain High-Throughput Assays and
                        Experimental Metadata
aaMap                   Dataset: Names and Characteristics of Amino
                        Acids
geneCov                 Sample expression matrix and phenotype
                        data.frames.
geneCovariate           Sample expression matrix and phenotype
                        data.frames.
geneData                Sample expression matrix and phenotype
                        data.frames.
reporter                Example data.frame representing reporter
                        information
sample.ExpressionSet    Dataset of class 'ExpressionSet'
sample.MultiSet         Data set of class 'MultiSet'
seD                     Sample expression matrix and phenotype
                        data.frames.
```

Let's load a small one...

```
data("aaMap")  # load the example data set 'geneData'

aaMap  # take a look
```

```
##             name let.1 let.3   scProp hyPhilic acidic
## 1        alanine     A   ala nonpolar    FALSE     NA
## 2       cysteine     C   cys    polar       NA     NA
## 3  aspartic.acid     D   asp    polar     TRUE   TRUE
## 4  glutamic.acid     E   glu    polar     TRUE   TRUE
## 5  phenylalanine     F   phe nonpolar    FALSE     NA
## 6        glycine     G   gly nonpolar       NA     NA
## 7      histidine     H   his    polar     TRUE  FALSE
## 8     isoleucine     I   ile nonpolar    FALSE     NA
## 9         lysine     K   lys    polar     TRUE  FALSE
## 10       leucine     L   leu nonpolar    FALSE     NA
## 11    methionine     M   met nonpolar    FALSE     NA
## 12    asparagine     N   asn    polar     TRUE  FALSE
## 13       proline     P   pro nonpolar       NA     NA
## 14     glutamine     Q   gln    polar     TRUE  FALSE
## 15      arginine     R   arg    polar     TRUE  FALSE
## 16        serine     S   ser    polar       NA     NA
## 17     threonine     T   thr    polar       NA     NA
## 18        valine     V   val nonpolar    FALSE     NA
## 19    tryptophan     W   trp nonpolar       NA     NA
## 20      tyrosine     Y   tyr    polar       NA     NA
```

```
table(aaMap$scProp)  # make a quick table of the side chain properties
```

```
##
## nonpolar    polar
##        9       11
```

```
ls("package:Biobase")  # get a listing of the methods of Biobase
```

What new functions have you gained by loading *Biobase*? To find out use: There are over 120 function listed and many of them have to do with microarray analysis which tells you something about the history of *Bioconductor*. Let's begin our introduction to *Biobase* by focusing on functions used to construct and extract data from an `ExpressionSet`, the data structure upon which *Biobase* revolves.

Expression Sets

The `ExpressionSet` is used to contain microarray data. It is comprised of 7 components, called *slots*, of the types and content listed in the table. The table headings give the *accessor* functions used to access the data in the *slots*.

assayData	phenoData	experimentData	annotation	featureData
environment	AnnotatedDataFrame	*MIAME* object	character	AnnotatedDataFra
expression data	sample information	description of experiment	name of an annotation database	feature informatio

We'll draw from the sample data available within `Biobase` to build an `ExpressionSet` from scratch using 4 of the 7 slots–we will omit the optional `featureData` and `protolData` for the sake of brevity. The `classVersion` slot is filled in automatically and need not concern us.

Constructing an ExpressionSet

Functions used

- `ExpressionSet()` to create an ExpressionSet object
- `c()` to combine elements into a vector
- `colnames()` to get the column names of a matrix
- `data()` to import example data from a package
- `data.frame()` to create a data.frame
- `head()` to see the first lines of a variable
- `names()` to get or set the column names of a data.frame
- `new()` generic function to create a new object

Assembling the 4 Components First, let's get the data for the core of the `ExpressionSet`, the expression matrix itself.

```
data(geneData)  # a sample expression matrix
colnames(geneData)  # take a look at the sample names
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
geneData[1:4, 1:5]  #take a look at the matrix itself
```

```
##                         A         B        C        D        E
## AFFX-MurIL2_at  192.7420  85.75330 176.7570 135.5750 64.49390
## AFFX-MurIL10_at  97.1370 126.19600  77.9216  93.3713 24.39860
## AFFX-MurIL4_at   45.8192   8.83135  33.0632  28.7072  5.94492
## AFFX-MurFAS_at   22.5445   3.60093  14.6883  12.3397 36.86630
```

Now, let's get some phenotype data that we can use to classify the samples. This will be the first of 2 *data.frame*s to build the `AnnotatedDataFrame` of the `phenoData` slot.

```
data(geneCov)  # the sample information is here--this will be the first data.frame in phenoData

head(geneCov)  # take a look at what we have
```

```
##   cov1 cov2 cov3
## A    1    1    1
## B    1    1    1
## C    1    1    1
## D    1    1    1
## E    1    2    1
## F    1    2    1
```

The column names are rather uninformative. How can we change cov1 to 'gender', cov2 to "case", and cov3 to "type"?

```
names(geneCov)
```

```
## [1] "cov1" "cov2" "cov3"
```

```
names(geneCov) = c("gender", "case", "type")
names(geneCov)
```

```
## [1] "gender" "case"   "type"
```

We can also describe the columns more fully by adding a second *data.frame* to the phenotype data to give a label description to each covariate column. The first argument to the *data.frame* constructor is adding 3 columns to the frame by name while the second is labeling the rows.

```
# Describe the columns with metadata--this is the second data.frame of
# phenData

metadata = data.frame(labelDescription = c("patient gender", "Case/control",
    "Tumor type"), row.names = c("cov1", "cov2", "cov3"))

metadata
```

```
##      labelDescription
## cov1   patient gender
## cov2     Case/control
## cov3       Tumor type
```

Now we can construct the `phenoData` component which is an *AnnotatedDataFrame*, so called because it consists of one *data.frame* to hold the covariate data and a second that annotates the covariates. The `new()` function is also a constructor, taking as its first argument the type of object to construct. The remaining arguments are the names of the two *data.frams* to put into the *AnnotatedDataFrams*.

```
# construct the `phenoData` component.

phenoData = new("AnnotatedDataFrame", data = geneCov, varMetadata = metadata)
phenoData
```

```
## An object of class 'AnnotatedDataFrame'
##   rowNames: A B ... Z (26 total)
##   varLabels: gender case type
##   varMetadata: labelDescription
```

The third component of our `ExpressionSet` is only a reference to the `hgu95av2` annotation database. The database used the *Affymetrix Human Genome U95 Set* annotation data assembled from public repositories and provides mappings between probe identifiers and gene symbols, chromosome locations, EC numbers, and other data. The table below lists all the mappings:

hgu95av2ACCNUM	hgu95av2CHR	hgu95av2CHRLOC	hgu95av2ENZYME	hgu95
hgu95av2GENENAME	hgu95av2GO	hgu95av2MAP	hgu95av2OMIM	hgu95
hgu95av2PFAM	hgu95av2PMID	hgu95av2PROSITE	hgu95av2REFSEQ	hgu95
hgu95av2UNIGENE	hgu95av2CHRLENGTHS	hgu95av2ENZYME2PROBE	hgu95av2GO2ALLPROBES	hgu95
hgu95av2PATH2PROBE	hgu95av2PMID2PROBE			

```
annotation = "hgu95av2"  # add a reference to our annotation database
```

The final component is a description of the experiment, following the MIAMI (Minimum Information About a Microarray Experiment) specification.

```
experimentData = new("MIAME", name = "C. Darwin", lab = "CCR", contact = "email.gov",
    title = "Small Experiment", abstract = "An example ExpressionSet", url = "www.lab.gov")
```

Now, we can build the `ExpressionSet` from the 4 components using the `ExpressionSet()` constructor.

```
eset = ExpressionSet(assayData = geneData, phenoData = phenoData, experimentData = experimentData,
    annotation = annotation)

str(eset)
```

```
## Formal class 'ExpressionSet' [package "Biobase"] with 7 slots
##   ..@ experimentData   :Formal class 'MIAME' [package "Biobase"] with 13 slots
##   .. .. ..@ name             : chr "C. Darwin"
##   .. .. ..@ lab              : chr "CCR"
##   .. .. ..@ contact          : chr "email.gov"
##   .. .. ..@ title            : chr "Small Experiment"
##   .. .. ..@ abstract         : chr "An example ExpressionSet"
##   .. .. ..@ url              : chr "www.lab.gov"
##   .. .. ..@ pubMedIds        : chr ""
##   .. .. ..@ samples          : list()
##   .. .. ..@ hybridizations   : list()
##   .. .. ..@ normControls     : list()
##   .. .. ..@ preprocessing    : list()
##   .. .. ..@ other            : list()
##   .. .. ..@ .__classVersion__:Formal class 'Versions' [package "Biobase"] with 1 slot
##   .. .. .. .. ..@ .Data:List of 2
##   .. .. .. .. .. ..$ : int [1:3] 1 0 0
##   .. .. .. .. .. ..$ : int [1:3] 1 1 0
##   ..@ assayData        :<environment: 0x3014668>
##   ..@ phenoData        :Formal class 'AnnotatedDataFrame' [package "Biobase"] with 4 slots
##   .. .. ..@ varMetadata      :'data.frame':  3 obs. of  1 variable:
##   .. .. .. ..$ labelDescription: chr [1:3] "patient gender" "Case/control" "Tumor type"
##   .. .. ..@ data             :'data.frame':  26 obs. of  3 variables:
##   .. .. .. ..$ gender: num [1:26] 1 1 1 1 1 1 1 1 1 1 ...
```

```
##   .. .. .. ..$ case  : num [1:26] 1 1 1 1 2 2 2 2 2 2 ...
##   .. .. .. ..$ type  : num [1:26] 1 1 1 1 1 1 1 1 2 2 ...
##   .. .. ..@ dimLabels        : chr [1:2] "sampleNames" "sampleColumns"
##   .. .. ..@ .__classVersion__:Formal class 'Versions' [package "Biobase"] with 1 slot
##   .. .. .. .. ..@ .Data:List of 1
##   .. .. .. .. .. ..$ : int [1:3] 1 1 0
##   ..@ featureData      :Formal class 'AnnotatedDataFrame' [package "Biobase"] with 4 slots
##   .. .. ..@ varMetadata      :'data.frame':  0 obs. of  1 variable:
##   .. .. .. ..$ labelDescription: chr(0)
##   .. .. ..@ data             :'data.frame':  500 obs. of  0 variables
##   .. .. ..@ dimLabels        : chr [1:2] "featureNames" "featureColumns"
##   .. .. ..@ .__classVersion__:Formal class 'Versions' [package "Biobase"] with 1 slot
##   .. .. .. .. ..@ .Data:List of 1
##   .. .. .. .. .. ..$ : int [1:3] 1 1 0
##   ..@ annotation       : chr "hgu95av2"
##   ..@ protocolData     :Formal class 'AnnotatedDataFrame' [package "Biobase"] with 4 slots
##   .. .. ..@ varMetadata      :'data.frame':  0 obs. of  1 variable:
##   .. .. .. ..$ labelDescription: chr(0)
##   .. .. ..@ data             :'data.frame':  26 obs. of  0 variables
##   .. .. ..@ dimLabels        : chr [1:2] "sampleNames" "sampleColumns"
##   .. .. ..@ .__classVersion__:Formal class 'Versions' [package "Biobase"] with 1 slot
##   .. .. .. .. ..@ .Data:List of 1
##   .. .. .. .. .. ..$ : int [1:3] 1 1 0
##   ..@ .__classVersion__:Formal class 'Versions' [package "Biobase"] with 1 slot
##   .. .. ..@ .Data:List of 4
##   .. .. .. ..$ : int [1:3] 3 1 1
##   .. .. .. ..$ : int [1:3] 2 26 0
##   .. .. .. ..$ : int [1:3] 1 3 0
##   .. .. .. ..$ : int [1:3] 1 0 0
```

The `ExpressionSet` is an object of a new type called 'S4' and its structure is more arcane than that of the objects we have been using. Such objects provide numerous *accessor functions* to allow us to get at their data. The help page for the object will list the functions available.

Taking a look at its structure, we see 7 *slots* corresponding to those listed earlier in the table. The data in these slots is available through the *accessor functions* listed in the header of the table. In addition, note that the covariates are listed in the `@data` data.frame within the `@phenoData` slot and that they are prefixed with a `$` sign. These elements can be reached directly as though they were elements of a list, e.g. `eset$gender`.

Were're finished. Now let's do a bit of analysis!

Analyzing the Data in Our ExpressionSet

Functions used

- `abline()` # to draw a horizontal line
- `array()` to create an empty array
- `boxplot()` to draw a boxplot
- `cutree()` to cut an hclust()-generated dendrogram into groups
- `dist()` to generate a distance matrix for hclust()
- `exprs()` to extract an expression matrix from an ExpressionSet
- `factor()` to create a factor object

- `featureNames()` to extract the name of probes (row name) from an ExpressionSet
- `format()` to format a number for display
- `hclust()` to perform hierarchical clustering
- `head()` to see the first portion of large variable
- `layout()` to change the layout of our graphics
- `length()` to get the length of an object
- `paste()` to concatenate character strings and numbers
- `plot()` to plot a graph
- `t.test()` to run a T-test on two groups
- `varMetadata()` to get variable labels from an ExpressionSet
- `which()` to see which elements of an object satisfy a condition

Analysis of the First Covariate, Gender There are three covariates in the ExpressionSet `eset`. We'll use the `varMetadata` function to see them.

```
varMetadata(eset)  # show the metadata about the phenotypes
```

```
##        labelDescription
## gender   patient gender
## case       Case/control
## type         Tumor type
```

Let's examine the effect of *gender* on our set of 500 probes. First, we will explore the data interactively using box plots and running a T-test on a single row of data (one probe). When we have that working, we will scale up to analyze all 500 probes.

The `exprs()` function extracts the expression matrix of 500 probes (rows) and 26 samples (columns) from `eset` so we'll be using this to get at the data. The *gender* covarate column in the first `data.frame` in the `phenoData` slot of `eset` contains 26 entries, one per sample, that can be used to partition the samples into two groups.

```
E = exprs(eset)  # extract the expression data as a matrix

E[1:4, 1:5]  # take a look
```

```
##                         A         B        C        D        E
## AFFX-MurIL2_at  192.7420  85.75330 176.7570 135.5750 64.49390
## AFFX-MurIL10_at  97.1370 126.19600  77.9216  93.3713 24.39860
## AFFX-MurIL4_at   45.8192   8.83135  33.0632  28.7072  5.94492
## AFFX-MurFAS_at   22.5445   3.60093  14.6883  12.3397 36.86630
```

```
gender = eset$gender  # we can get the gender array this way

gender  # take a look
```

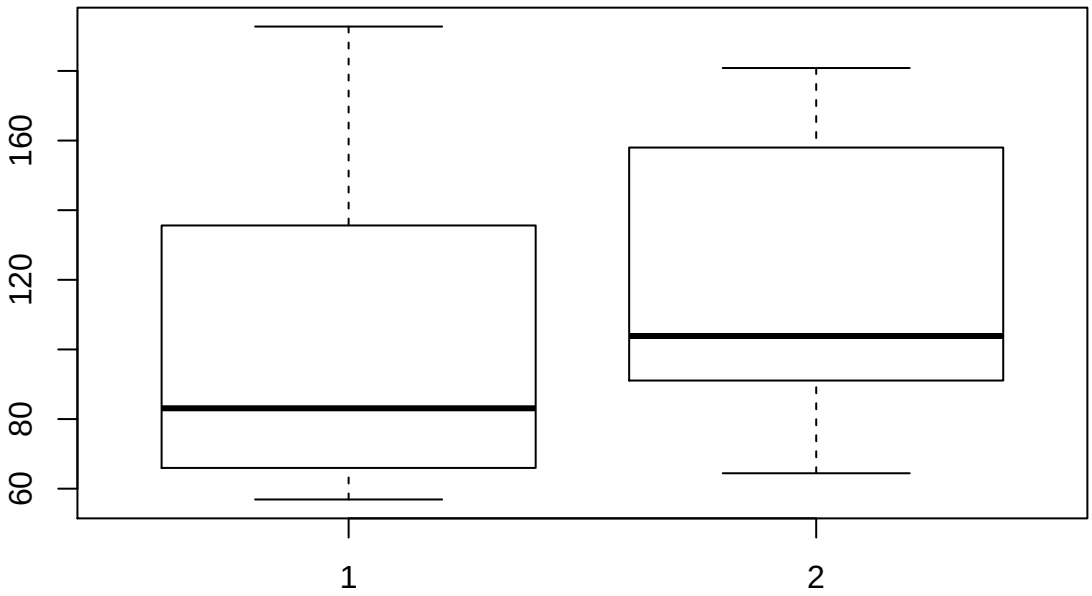
```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2
```

We will now use `gender` as a grouping vector for our 26 samples and feed `boxplot` a *model* for the expression data in the first row that says:

`E[1,] is.a.function.of gender`

The *is.a.function.of* operator is the tilde `~`

```r
# group the samples by gender and boxplot the probe intensities for the
# groups use the ~ 'is.a.function.of' symbol to define the model for boxplot

boxplot(E[1, ] ~ gender)
```




```r
# use the same model for a `t.test` of the probe intensities for the 2
# groups

t.test(E[1, ] ~ gender)
```

```
##
##  Welch Two Sample t-test
##
## data:  E[1, ] by gender
## t = -1.0582, df = 23.578, p-value = 0.3007
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -54.97275  17.73203
## sample estimates:
## mean in group 1 mean in group 2
##        105.8006        124.4210
```

Scale up for all 500 Probes Now that we have an analysis that works for one probe, we can apply it to all 500 probes using a `for` loop. This, by the way, is not the preferred method of making such comparisons but it is simple and illustrates the basic principles.

```r
T = vector()  # initialize an array for P-Values

for (n in 1:500) {
    T[n] = t.test(E[n, ] ~ gender)$p.value
}  # do just what we did above for each probe and extract the P-value from the t.test object--this is t
```

```
length(T)  # just check to make sure we got 500 P-vals
```

```
## [1] 500
```

```
head(T)  # take a look
```

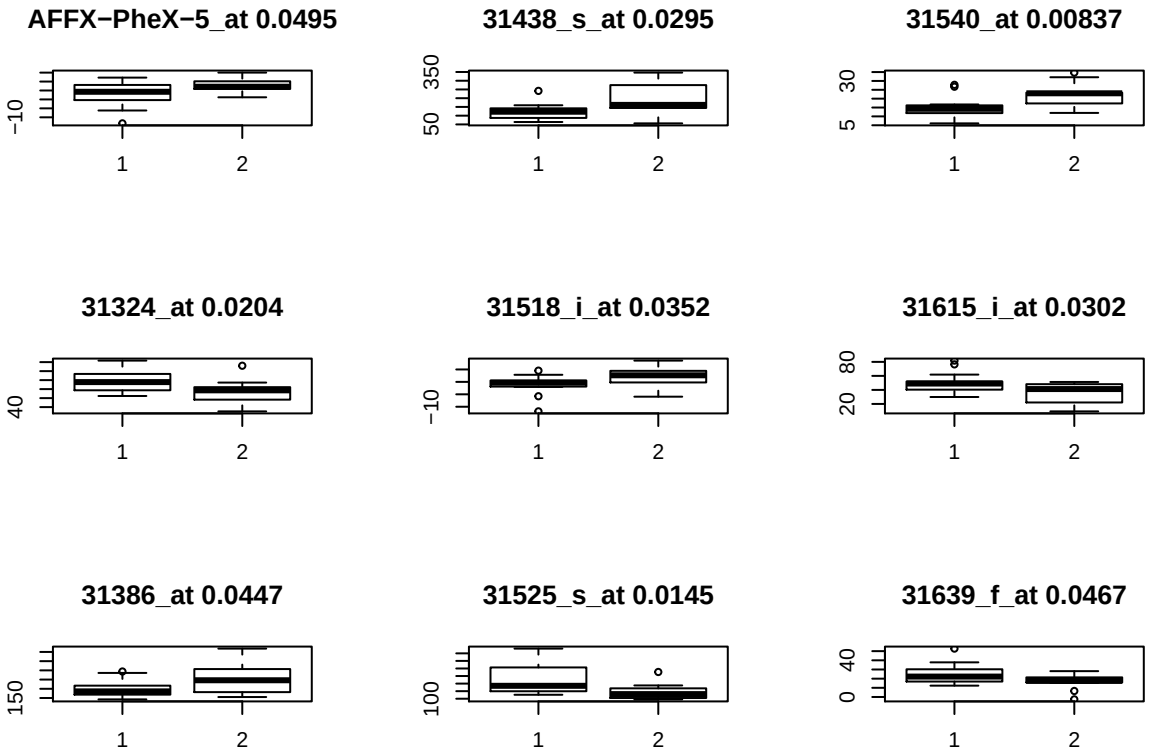
```
## [1] 0.3006994 0.5717812 0.9852404 0.3062817 0.6786697 0.6793440
```

```
which(T <= 0.05)  #use 'which' to see the indices of the significant cases
```

```
## [1]  30  85 147 199 279 286 301 376 400
```

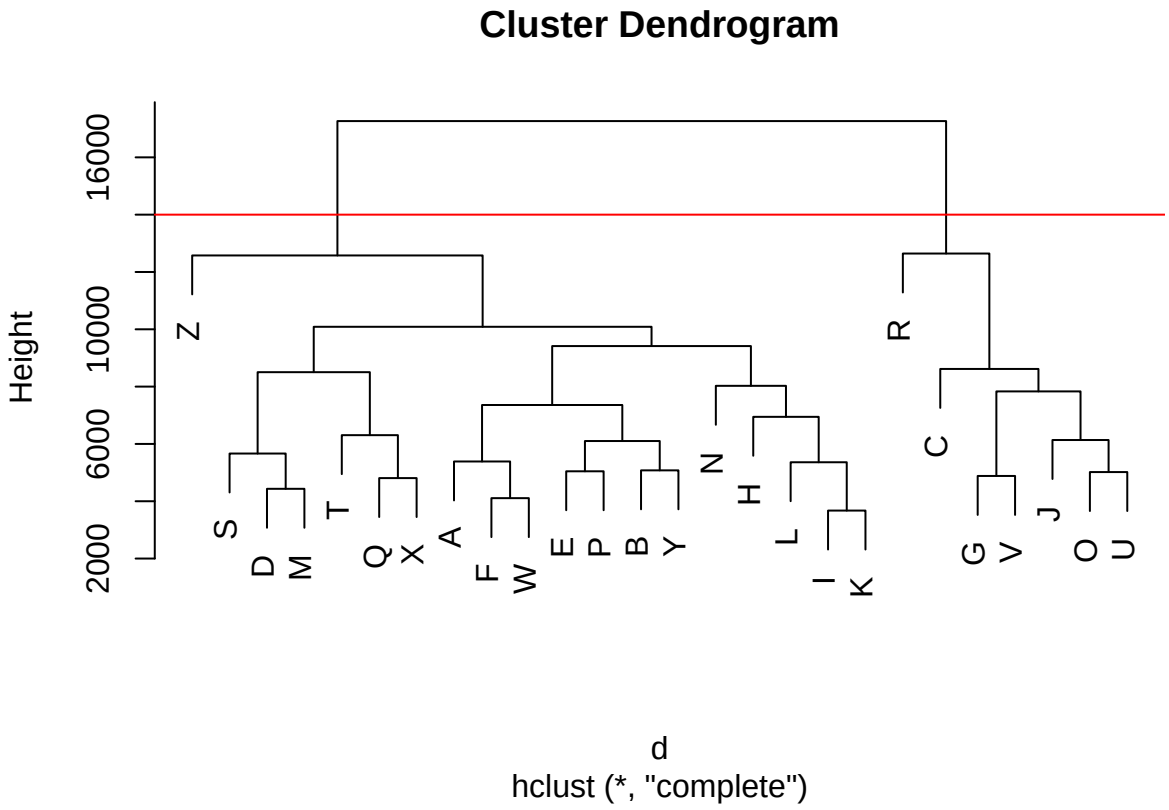
We see only a handful of probes with significant differences between the two groups of gender. Let's make boxplots for these 9 interesting probes.

```
# we can divide the graphical display into 12 areas using

layout(matrix(1:9, 3, 3))

# next, we plot 9 boxplots and label them with probe id and P-value


F = featureNames(eset)  # if we want to add the feature names to the graph title

for (n in which(T <= 0.05)) {
    boxplot(E[n, ] ~ gender, main = paste(F[n], format(T[n], digits = 3)))
}  # loop through all cases where the P-val <= 0.05
```



Adding Our Own Covariate During the introductory session we analyzed `geneData` (which is the same expression matrix we have incorporated into `eset`) using hierarchical clustering and found that it broke down into two distinct groups. Let's add those groups to the `phenoData` slot of `eset` and reanalyze.

```
# First, we'll create the distance matrix needed by `hclust()`


d = dist(t(E))  # transpose the expression matrix to cluster samples (columns) rather than probes (rows
h = hclust(d)  # perform the clustering
plot(h)  # plot the cluster dendrogram
abline(h = 14000, col = "red")
```



Here we have our dendrogram and see that we can cut it at a height of 14000 to generate the two groups.

```
groups = factor(cutree(h, h = 14000))  # cut the tree at 1400
groups  # take a look at which samples fall into which groups
```

```
## A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
## 1 1 2 1 1 1 2 1 1 2 1 1 1 1 2 1 1 2 1 1 2 2 1 1 1 1
## Levels: 1 2
```

Now we are ready to add a new covariate to eset–we'll call it `hc.group`.

```
eset$hc.group = groups  # add the new covariate

varMetadata(eset)  # verify that we have done what we wanted to do
```

```
##          labelDescription
## gender     patient gender
## case         Case/control
## type           Tumor type
## hc.group             <NA>
```

We can now run our T-tests.

```
T = vector()
for (n in 1:500) {
    T[n] = t.test(E[n, ] ~ eset$hc.group)$p.value
}  # perform the 500 T-tests
```

Is the following code equivalent the above?

```
T=array();for(n in 1:500){T[n]=t.test(E[n,]~groups)$p.value}
```

How many probes pass the test?

```
which(T <= 0.05)  # see which pass the test
```

```
##   [1]   1   3   6   8  12  20  22  23  24  25  27  28  31  35  36  38  46
##  [18]  47  48  51  58  65  67  73  74  82  86  89  91  96 106 110 117 118
##  [35] 121 122 123 126 128 129 131 134 139 140 146 153 157 158 163 169 174
##  [52] 177 178 180 181 184 185 186 189 192 193 196 197 201 203 204 205 207
##  [69] 208 210 214 218 220 222 224 226 232 233 235 236 237 239 242 251 253
##  [86] 256 262 263 264 265 266 269 270 272 275 276 281 282 288 290 291 295
## [103] 296 297 298 299 302 304 306 307 309 312 313 315 316 317 320 326 327
## [120] 328 329 333 334 336 341 343 344 346 351 353 355 362 363 369 373 375
## [137] 379 381 383 384 387 389 390 392 393 396 397 398 403 404 405 409 410
## [154] 423 426 431 432 433 436 437 441 445 446 452 455 458 460 465 467 469
## [171] 470 471 474 476 482 483 485 487 494 495 499
```

This is a much greater effect than we observed with *gender*!

What is our best P-value?

```
min(T)
```

```
## [1] 3.268072e-10
```

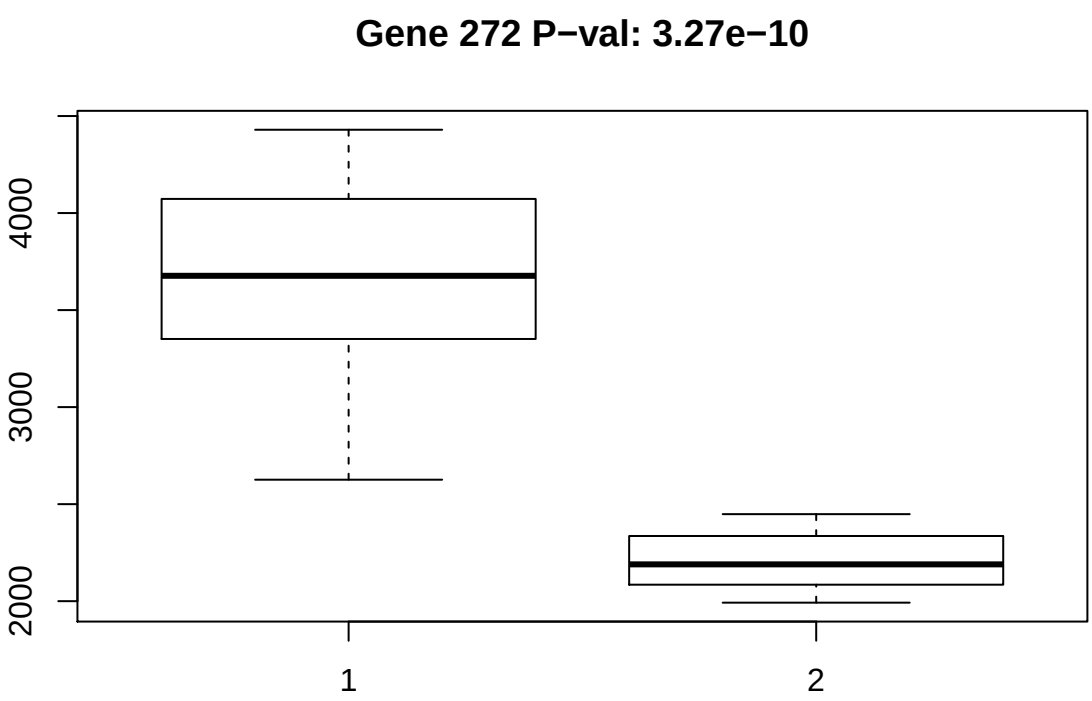
If we want to know which probe has the lowest P-value, we can use `which.min()`:

```
which.min(T)  # Let's see the index of the probe with the best P-value
```

```
## [1] 272
```

Now, let's `boxplot()` it.

```
# let's see the boxplot

boxplot(E[which.min(T), ] ~ eset$hc.group, main = paste("Gene", which.min(T),
    "P-val:", format(T[which.min(T)], digits = 3)))
```



What is the the probe id corresponding to the minimum P-value?

```
featureNames(eset[which.min(T)])
```

```
## [1] "31511_at"
```

Biostrings

The *Biostrings* library provides containers for DNA, RNA, or protein sequences; *BString* and *BStringSet*. These come in three varieties:

DNA	RNA	Proteins, Peptides
DNAString()	RNAString()	AAString()
DNAStringSet()	RNAStringSet()	AAStringSet()

Let's load the *Biostrings* library and take a look at the sample data provided:

```
library(Biostrings)  # load the library 'Biostrings'
```

```
## Loading required package: S4Vectors
## Loading required package: stats4
```

```
## Loading required package: IRanges
## Loading required package: XVector
```

Next, let's see if there is any sample data to explore:

```
data(package = "Biostrings")  # list the example data sets of 'Biostrings'

data(phiX174Phage)  # load a sample DNAStringSet

phiX174Phage  # take a look
```

```
##   A DNAStringSet instance of length 6
##     width seq                                               names
## [1]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA Genbank
## [2]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA RF70s
## [3]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA SS78
## [4]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA Bull
## [5]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA G97
## [6]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA NEB03
```

```
ls("package:Biostrings")  # list the methods of 'Biostrings'
```

Functions provided by *biostrings*

```
##   [1] "AA_ALPHABET"                      "AAMultipleAlignment"
##   [3] "AA_PROTEINOGENIC"                 "AA_STANDARD"
##   [5] "AAString"                         "AAStringSet"
##   [7] "AAStringSetList"                  "aligned"
##   [9] "alphabet"                         "alphabetFrequency"
##  [11] "AMINO_ACID_CODE"                  "append"
##  [13] "as.data.frame"                    "as.list"
##  [15] "as.matrix"                        "BString"
##  [17] "BStringSet"                       "BStringSetList"
##  [19] "cDNA"                             "chartr"
##  [21] "codons"                           "coerce"
##  [23] "collapse"                         "colmask"
##  [25] "colmask<-"                        "compare"
##  [27] "compareStrings"                   "complement"
##  [29] "complementedPalindromeArmLength" "complementedPalindromeLeftArm"
##  [31] "complementedPalindromeRightArm"  "computeAllFlinks"
##  [33] "consensusMatrix"                  "consensusString"
##  [35] "consensusViews"                   "countIndex"
##  [37] "countPattern"                     "countPDict"
##  [39] "countPWM"                         "coverage"
##  [41] "deletion"                         "detail"
##  [43] "dinucleotideFrequency"            "dinucleotideFrequencyTest"
##  [45] "dna2rna"                          "DNA_ALPHABET"
##  [47] "DNA_BASES"                        "DNAMultipleAlignment"
##  [49] "DNAString"                        "DNAStringSet"
##  [51] "DNAStringSetList"                 "duplicated"
```

```
##  [53] "encoding"                          "end"
##  [55] "endIndex"                          "errorSubstitutionMatrices"
##  [57] "extractAllMatches"                 "extractAt"
##  [59] "fasta.info"                        "fastq.geometry"
##  [61] "findComplementedPalindromes"       "findPalindromes"
##  [63] "gaps"                              "GENETIC_CODE"
##  [65] "GENETIC_CODE_TABLE"                "getGeneticCode"
##  [67] "getSeq"                            "get_seqtype_conversion_lookup"
##  [69] "gregexpr2"                         "hasAllFlinks"
##  [71] "hasLetterAt"                       "hasOnlyBaseLetters"
##  [73] "head"                              "IlluminaQuality"
##  [75] "%in%"                              "indel"
##  [77] "initialize"                        "injectHardMask"
##  [79] "insertion"                         "intersect"
##  [81] "isMatchingAt"                      "isMatchingEndingAt"
##  [83] "isMatchingStartingAt"              "is.unsorted"
##  [85] "IUPAC_CODE_MAP"                    "lcprefix"
##  [87] "lcsubstr"                          "lcsuffix"
##  [89] "letter"                            "letterFrequency"
##  [91] "letterFrequencyInSlidingView"      "longestConsecutive"
##  [93] "mask"                              "maskeddim"
##  [95] "maskedncol"                        "maskednrow"
##  [97] "maskedratio"                       "maskedwidth"
##  [99] "maskGaps"                          "maskMotif"
## [101] "masks"                             "masks<-"
## [103] "match"                             "matchLRPatterns"
## [105] "matchPattern"                      "matchPDict"
## [107] "matchProbePair"                    "matchprobes"
## [109] "matchPWM"                          "maxScore"
## [111] "maxWeights"                        "mergeIUPACLetters"
## [113] "minScore"                          "minWeights"
## [115] "mismatch"                          "mismatchSummary"
## [117] "mismatchTable"                     "mkAllStrings"
## [119] "N50"                               "narrow"
## [121] "nchar"                             "nedit"
## [123] "neditAt"                           "neditEndingAt"
## [125] "neditStartingAt"                   "needwunsQS"
## [127] "nindel"                            "nmatch"
## [129] "nmismatch"                         "nnodes"
## [131] "nucleotideFrequencyAt"             "nucleotideSubstitutionMatrix"
## [133] "oligonucleotideFrequency"          "oligonucleotideTransitions"
## [135] "order"                             "padAndClip"
## [137] "pairwiseAlignment"                 "PairwiseAlignments"
## [139] "PairwiseAlignmentsSingleSubject"   "palindromeArmLength"
## [141] "palindromeLeftArm"                 "palindromeRightArm"
## [143] "partitioning"                      "pattern"
## [145] "patternFrequency"                  "PDict"
## [147] "PhredQuality"                      "pid"
## [149] "pmatchPattern"                     "PWM"
## [151] "PWMscoreStartingAt"                "quality"
## [153] "QualityScaledAAStringSet"          "QualityScaledBStringSet"
## [155] "QualityScaledDNAStringSet"         "QualityScaledRNAStringSet"
## [157] "qualitySubstitutionMatrices"       "rank"
## [159] "readAAMultipleAlignment"           "readAAStringSet"
```

```
## [161] "readBStringSet"                "readDNAMultipleAlignment"
## [163] "readDNAStringSet"              "readRNAMultipleAlignment"
## [165] "readRNAStringSet"              "relistToClass"
## [167] "replaceAt"                     "replaceLetterAt"
## [169] "reverse"                       "reverseComplement"
## [171] "rna2dna"                       "RNA_ALPHABET"
## [173] "RNA_BASES"                     "RNA_GENETIC_CODE"
## [175] "RNAMultipleAlignment"          "RNAString"
## [177] "RNAStringSet"                  "RNAStringSetList"
## [179] "rowmask"                       "rowmask<-"
## [181] "saveXStringSet"                "score"
## [183] "seqtype"                       "seqtype<-"
## [185] "setdiff"                       "setequal"
## [187] "show"                          "SolexaQuality"
## [189] "sort"                          "splitAsListReturnedClass"
## [191] "stackStrings"                  "start"
## [193] "startIndex"                    "stringDist"
## [195] "subpatterns"                   "subseq"
## [197] "subseq<-"                      "substr"
## [199] "substring"                     "summary"
## [201] "tail"                          "tb"
## [203] "tb.width"                      "threebands"
## [205] "toComplex"                     "toString"
## [207] "transcribe"                    "translate"
## [209] "trimLRPatterns"                "trinucleotideFrequency"
## [211] "twoWayAlphabetFrequency"       "type"
## [213] "unaligned"                     "union"
## [215] "uniqueLetters"                 "unitScale"
## [217] "unlist"                        "unmasked"
## [219] "unstrsplit"                    "updateObject"
## [221] "vcountPattern"                 "vcountPDict"
## [223] "Views"                         "vmatchPattern"
## [225] "vmatchPDict"                   "vwhichPDict"
## [227] "which.isMatchingAt"            "which.isMatchingEndingAt"
## [229] "which.isMatchingStartingAt"    "whichPDict"
## [231] "width"                         "width0"
## [233] "writePairwiseAlignments"       "write.phylip"
## [235] "writeXStringSet"               "xscat"
```

Exploring Biostrings

Functions Used

- data()
- class()
- str()
- names()
- DNAString()
- RNAString()
- AAString()
- reverse()
- complement()
- reverseComplement()

- translate()
- alphabetFrequency()
- trinucleotideFrequency()
- dinucleotideFrequency()
- oligonucleotideFrequency()
- tail()
- consensusMatrix()
- which()
- colSums()

Creating a DNAString object Here we make a DNAString to contain a portion of the sequence of exon 4 of the human oncogene homologue (c-abl) of the abelson murine leukemia virus. Let's create a DNAString from the sequence and translate it.

```
abl1 = DNAString("ATCTTCGCGAATGGTATATAGTAGCGCTTCGCGCATAAT")  # use a 'constructor' to make a DNAString
abl1  # take a look at the new object
```

```
##   39-letter "DNAString" instance
## seq: ATCTTCGCGAATGGTATATAGTAGCGCTTCGCGCATAAT
```

```
translate(abl1)
```

```
##   13-letter "AAString" instance
## seq: IFANGI**RFAHN
```

A more advanced function: alphabetFrequency() The function `alphabetFrequency` looks interesting, so let's try it.

```
data(phiX174Phage)  # get a DNAStringset

phiX174Phage  # take a look
```

```
##   A DNAStringSet instance of length 6
##     width seq                                               names
## [1]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA Genbank
## [2]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA RF70s
## [3]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA SS78
## [4]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA Bull
## [5]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA G97
## [6]  5386 GAGTTTTATCGCTTCCATGAC...ATTGGCGTATCCAACCTGCA NEB03
```

```
alphabetFrequency(phiX174Phage)  # generate a table of base counts
```

```
##         A    C    G    T M R W S Y K V H D B N - + .
## [1,] 1291 1157 1254 1684 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [2,] 1292 1156 1253 1685 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [3,] 1292 1156 1253 1685 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [4,] 1292 1155 1253 1686 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [5,] 1292 1156 1253 1685 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [6,] 1292 1155 1253 1686 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Nice, but how do we use this? We can compute %GC for these 6 sequences as follows:

```
a = alphabetFrequency(phiX174Phage)  # assign the base counts to a variable

(a[, "G"] + a[, "C"])/(a[, "G"] + a[, "C"] + a[, "A"] + a[, "T"]) * 100  # use the variable to comput %
```

```
## [1] 44.76420 44.72707 44.72707 44.70850 44.72707 44.70850
```

The sequences seem to be very similar in composition. How similar? We can easily take a couple more steps:

```
dinucleotideFrequency(phiX174Phage)  # get a table of dinucleotide counts
```

```
##       AA  AC  AG  AT  CA  CC  CG  CT  GA  GC  GG  GT  TA  TC  TG  TT
## [1,] 395 261 251 383 257 229 267 404 327 347 255 325 312 320 480 572
## [2,] 394 260 253 384 258 229 265 404 327 347 254 325 313 320 480 572
## [3,] 394 260 253 384 258 229 265 404 327 347 254 325 313 320 480 572
## [4,] 395 260 252 384 257 229 266 403 327 346 254 326 313 320 480 573
## [5,] 395 261 253 382 257 229 266 404 326 346 254 327 314 320 479 572
## [6,] 394 260 253 384 258 227 265 405 327 347 254 325 313 321 480 572
```

```
trinucleotideFrequency(phiX174Phage)  # get a table of trinucleotide counts
```

```
##      AAA AAC AAG AAT ACA ACC ACG ACT AGA AGC AGG AGT ATA ATC ATG ATT CAA
## [1,] 133  67  93 102  47  62  63  89  72  53  74  52  60  56 140 127  91
## [2,] 133  66  93 102  47  62  62  89  73  53  75  52  60  56 141 127  91
## [3,] 133  66  93 102  47  62  62  89  73  53  75  52  60  56 141 127  91
## [4,] 133  66  93 103  47  62  62  89  73  53  74  52  60  56 140 128  91
## [5,] 133  66  93 103  47  62  62  90  73  53  75  52  60  56 139 127  91
## [6,] 133  66  93 102  47  62  62  89  73  53  75  52  60  56 141 127  91
##      CAC CAG CAT CCA CCC CCG CCT CGA CGC CGG CGT CTA CTC CTG CTT GAA GAC
## [1,]  40  61  64  59  38  70  62  52  92  31  92  78  67 132 127  70  82
## [2,]  40  62  64  59  38  70  62  52  92  30  91  78  67 132 127  69  82
## [3,]  40  62  64  59  38  70  62  52  92  30  91  78  67 132 127  69  82
## [4,]  39  62  64  58  38  71  62  52  92  30  92  78  67 132 126  70  82
## [5,]  40  62  63  58  38  71  62  52  92  30  92  78  67 132 127  70  82
## [6,]  40  62  64  59  36  69  63  52  92  30  91  78  68 132 127  69  82
##      GAG GAT GCA GCC GCG GCT GGA GGC GGG GGT GTA GTC GTG GTT TAA TAC TAG
## [1,]  74 101  60  71  67 149  63  72  20 100  54  65  66 140 101  72  23
## [2,]  74 102  61  71  66 149  63  72  19 100  55  65  65 140 101  72  24
## [3,]  74 102  61  71  66 149  63  72  19 100  55  65  65 140 101  72  24
## [4,]  74 101  61  71  66 148  62  72  20 100  54  65  67 140 101  73  23
## [5,]  74 100  61  71  66 148  62  72  19 101  55  65  67 140 101  73  24
## [6,]  74 102  61  71  66 149  63  72  19 100  55  65  65 140 101  72  24
##      TAT TCA TCC TCG TCT TGA TGC TGG TGT TTA TTC TTG TTT
## [1,] 116  91  58  67 104 139 130 130  81 120 132 142 178
## [2,] 116  91  58  67 104 138 130 130  82 120 132 142 178
## [3,] 116  91  58  67 104 138 130 130  82 120 132 142 178
## [4,] 116  91  58  67 104 139 129 130  82 121 132 141 179
## [5,] 116  91  58  67 104 138 129 130  82 121 132 141 178
## [6,] 116  91  58  68 104 138 130 130  82 120 132 142 178
```

```
O = oligonucleotideFrequency(phiX174Phage[1], 5)  # tabulate 5-mers for the first sequence

colnames(O)[which.max(O)]  # identify the 5-mer with highest frequency
```

```
## [1] "TGCTG"
```

The BiostringSet in More Detail

A *BiostringSet* is a collection of *Biostrings* that is stored as one long string with ranges that are used to construct the individual members.

Let's make a small one Here we make a *BiostringSet* of two sequences by feeding the constructor function a single DNA sequence, two starting positions, and two widths.

```
library(Biostrings)

sset = DNAStringSet(DNAString("gcgcgctcgctcg"), start = c(1, 5), width = c(5,
    8))

sset
```

```
##   A DNAStringSet instance of length 2
##     width seq
## [1]     5 GCGCG
## [2]     8 GCTCGCTC
```

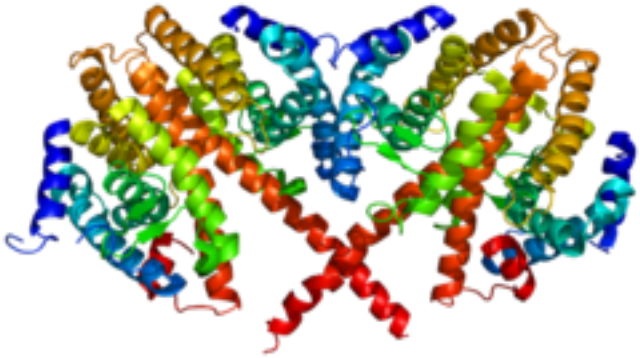
What can we do with a *Biostring*?

A Look at the Binding Site of HNF4-alpha

Functions Used

- `barplot()` to plot a stacked bar graph of rows 1:4 (ATCG) of a consensus matrix
- `consensusMatrix()` to compute a consensus matrix from a `DNAStringSet`
- `rainbow()` to create a vector of colors

HNF4 alpha is a nuclear transcription factor which binds DNA as a homodimer. Let's work with a *BiostringSet* consisting of 71 aligned HNF4-alpha binding sites. To do this, we will load the sample *BiostringSet*, `HNF4alpha`.



```
data(HNF4alpha)  # load the HNF4aplha sample data set

HNF4alpha  # see how it looks
```

```
##   A DNAStringSet instance of length 71
##      width seq
##  [1]    13 AGTTCAAGGATCA
##  [2]    13 GGGGTCAAGGGTT
##  [3]    13 AGGGTAAAGGTTG
##  [4]    13 GTCACAAAAGTCC
##  [5]    13 GGTCCAAAGGGCG
##  ...   ... ...
## [67]    13 CTTGGAACCGGGG
## [68]    13 AGGTCAGGGTCCC
## [69]    13 TGTCCAAAGTCCA
## [70]    13 TGATCAGACAAAG
## [71]    13 AAACCAAAGTTCA
```

To see the internal structure.....once again use `str()`

Where is the DNA sequence?

```
str(HNF4alpha)  # see the structure of `phiX174Phage`
```

```
## Formal class 'DNAStringSet' [package "Biostrings"] with 5 slots
##   ..@ pool           :Formal class 'SharedRaw_Pool' [package "IRanges"] with 2 slots
##   .. .. ..@ xp_list                    :List of 1
##   .. .. .. ..$ :<externalptr>
##   .. .. ..@ .link_to_cached_object_list:List of 1
##   .. .. .. ..$ :<environment: 0x76a18a0>
##   ..@ ranges         :Formal class 'GroupedIRanges' [package "IRanges"] with 7 slots
##   .. .. ..@ group          : int [1:71] 1 1 1 1 1 1 1 1 1 1 ...
##   .. .. ..@ start          : int [1:71] 1 14 27 40 53 66 79 92 105 118 ...
##   .. .. ..@ width          : int [1:71] 13 13 13 13 13 13 13 13 13 13 ...
##   .. .. ..@ NAMES          : NULL
##   .. .. ..@ elementType    : chr "integer"
##   .. .. ..@ elementMetadata: NULL
##   .. .. ..@ metadata       : list()
##   ..@ elementType    : chr "DNAString"
##   ..@ elementMetadata: NULL
##   ..@ metadata       : list()
```

In the structure output we see '@ranges'. Let's try to access the ranges using the `@ranges` syntax for S4 objects. This is roughly analogous to the `$` method for lists.

```
head(HNF4alpha@ranges)  # get head of the table of ranges for the 6 sequences
```

```
##   group start end width
## 1     1     1  13    13
## 2     1    14  26    13
## 3     1    27  39    13
```

```
## 4     1    40  52    13
## 5     1    53  65    13
## 6     1    66  78    13
```

```
tail(HNF4alpha@ranges)  # get the tail of the table of ranges for the 6 sequences
```

```
##   group start end width
## 1     1   846 858    13
## 2     1   859 871    13
## 3     1   872 884    13
## 4     1   885 897    13
## 5     1   898 910    13
## 6     1   911 923    13
```

To examine the binding site, let's generate a consensus matrix.

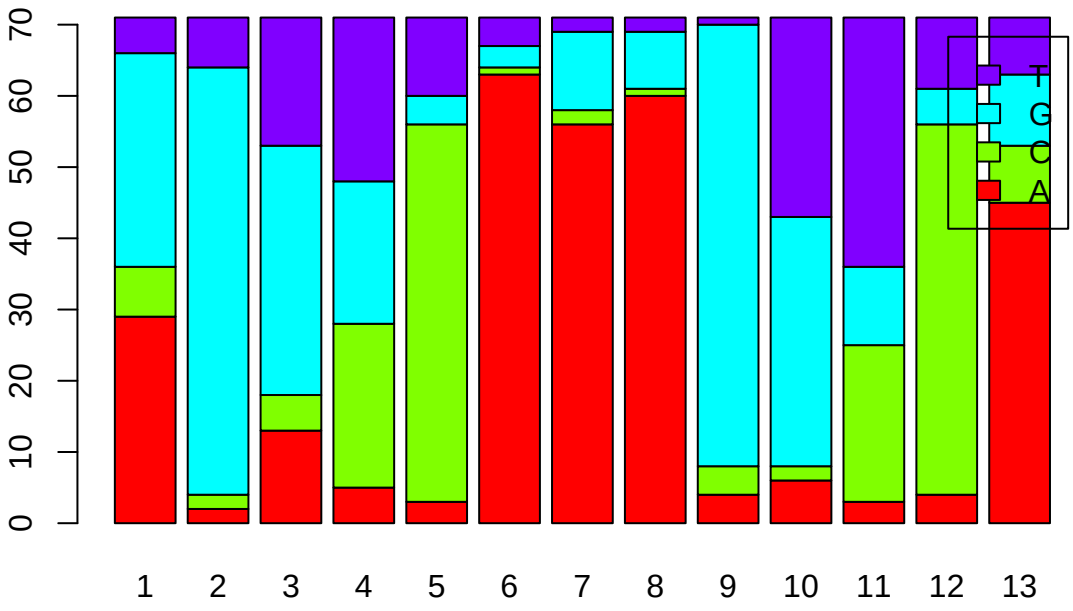
```
m = consensusMatrix(HNF4alpha)  # counts for each nucleotide at each position

m[1:4, ]  # the matrix is large so just look at 4 rows
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## A   29    2   13    5    3   63   56   60    4     6     3     4    45
## C    7    2    5   23   53    1    2    1    4     2    22    52     8
## G   30   60   35   20    4    3   11    8   62    35    11     5    10
## T    5    7   18   23   11    4    2    2    1    28    35    10     8
```

Can we display this result graphically? How about a stacked bar plot?

Is this similar to a "Sequence Logo"?

```
barplot(m[1:4, ], col = rainbow(4), legend = TRUE, names.arg = paste(1:13))
```



BSgenome

The *BSgenome* library provides a container for genomes, including large multi-chromosome genomes. A number of genomes have been packaged for Bioconductor. The `available()` function lists these genomes.

```
library(BSgenome)  # load the library 'BSgenome'
```

```
## Loading required package: GenomeInfoDb
## Loading required package: GenomicRanges
## Loading required package: rtracklayer
```

```
available.genomes()  # get a list of genomes you can install
```

```
##  [1] "BSgenome.Alyrata.JGI.v1"
##  [2] "BSgenome.Amellifera.BeeBase.assembly4"
##  [3] "BSgenome.Amellifera.UCSC.apiMel2"
##  [4] "BSgenome.Amellifera.UCSC.apiMel2.masked"
##  [5] "BSgenome.Athaliana.TAIR.04232008"
##  [6] "BSgenome.Athaliana.TAIR.TAIR9"
##  [7] "BSgenome.Btaurus.UCSC.bosTau3"
##  [8] "BSgenome.Btaurus.UCSC.bosTau3.masked"
##  [9] "BSgenome.Btaurus.UCSC.bosTau4"
## [10] "BSgenome.Btaurus.UCSC.bosTau4.masked"
## [11] "BSgenome.Btaurus.UCSC.bosTau6"
## [12] "BSgenome.Btaurus.UCSC.bosTau6.masked"
## [13] "BSgenome.Celegans.UCSC.ce10"
## [14] "BSgenome.Celegans.UCSC.ce2"
## [15] "BSgenome.Celegans.UCSC.ce6"
## [16] "BSgenome.Cfamiliaris.UCSC.canFam2"
## [17] "BSgenome.Cfamiliaris.UCSC.canFam2.masked"
## [18] "BSgenome.Cfamiliaris.UCSC.canFam3"
## [19] "BSgenome.Cfamiliaris.UCSC.canFam3.masked"
## [20] "BSgenome.Dmelanogaster.UCSC.dm2"
## [21] "BSgenome.Dmelanogaster.UCSC.dm2.masked"
## [22] "BSgenome.Dmelanogaster.UCSC.dm3"
## [23] "BSgenome.Dmelanogaster.UCSC.dm3.masked"
## [24] "BSgenome.Drerio.UCSC.danRer5"
## [25] "BSgenome.Drerio.UCSC.danRer5.masked"
## [26] "BSgenome.Drerio.UCSC.danRer6"
## [27] "BSgenome.Drerio.UCSC.danRer6.masked"
## [28] "BSgenome.Drerio.UCSC.danRer7"
## [29] "BSgenome.Drerio.UCSC.danRer7.masked"
## [30] "BSgenome.Ecoli.NCBI.20080805"
## [31] "BSgenome.Gaculeatus.UCSC.gasAcu1"
## [32] "BSgenome.Gaculeatus.UCSC.gasAcu1.masked"
## [33] "BSgenome.Ggallus.UCSC.galGal3"
## [34] "BSgenome.Ggallus.UCSC.galGal3.masked"
## [35] "BSgenome.Ggallus.UCSC.galGal4"
## [36] "BSgenome.Ggallus.UCSC.galGal4.masked"
## [37] "BSgenome.Hsapiens.NCBI.GRCh38"
## [38] "BSgenome.Hsapiens.UCSC.hg17"
## [39] "BSgenome.Hsapiens.UCSC.hg17.masked"
## [40] "BSgenome.Hsapiens.UCSC.hg18"
```

```
## [41] "BSgenome.Hsapiens.UCSC.hg18.masked"
## [42] "BSgenome.Hsapiens.UCSC.hg19"
## [43] "BSgenome.Hsapiens.UCSC.hg19.masked"
## [44] "BSgenome.Mfuro.UCSC.musFur1"
## [45] "BSgenome.Mmulatta.UCSC.rheMac2"
## [46] "BSgenome.Mmulatta.UCSC.rheMac2.masked"
## [47] "BSgenome.Mmulatta.UCSC.rheMac3"
## [48] "BSgenome.Mmulatta.UCSC.rheMac3.masked"
## [49] "BSgenome.Mmusculus.UCSC.mm10"
## [50] "BSgenome.Mmusculus.UCSC.mm10.masked"
## [51] "BSgenome.Mmusculus.UCSC.mm8"
## [52] "BSgenome.Mmusculus.UCSC.mm8.masked"
## [53] "BSgenome.Mmusculus.UCSC.mm9"
## [54] "BSgenome.Mmusculus.UCSC.mm9.masked"
## [55] "BSgenome.Osativa.MSU.MSU7"
## [56] "BSgenome.Ptroglodytes.UCSC.panTro2"
## [57] "BSgenome.Ptroglodytes.UCSC.panTro2.masked"
## [58] "BSgenome.Ptroglodytes.UCSC.panTro3"
## [59] "BSgenome.Ptroglodytes.UCSC.panTro3.masked"
## [60] "BSgenome.Rnorvegicus.UCSC.rn4"
## [61] "BSgenome.Rnorvegicus.UCSC.rn4.masked"
## [62] "BSgenome.Rnorvegicus.UCSC.rn5"
## [63] "BSgenome.Rnorvegicus.UCSC.rn5.masked"
## [64] "BSgenome.Scerevisiae.UCSC.sacCer1"
## [65] "BSgenome.Scerevisiae.UCSC.sacCer2"
## [66] "BSgenome.Scerevisiae.UCSC.sacCer3"
## [67] "BSgenome.Sscrofa.UCSC.susScr3"
## [68] "BSgenome.Sscrofa.UCSC.susScr3.masked"
## [69] "BSgenome.Tgondii.ToxoDB.7.0"
## [70] "BSgenome.Tguttata.UCSC.taeGut1"
## [71] "BSgenome.Tguttata.UCSC.taeGut1.masked"
```

A Brief Exploration of the C. Elegans Genome

Let's load the *C. elegans* genome to experiment.

```
library(BSgenome.Celegans.UCSC.ce6)  # load your local copy of the C. elegans genome
```

The name of a *BSgenome* object consists of four parts; the type of object (*BSgenome*), a abbreviated organism name, the source of the assembly, and the version. After loading the genome, however, you can refer to the genome using on the second part of the name–in this case "Celegans".

What can we do with BSgenomes? Use 'ls("package:BSgenome") to see:

```
ls("package:BSgenome")  # list the methods of 'BSgenome'
```

```
##  [1] "as.list"                     "available.genomes"
##  [3] "available.SNPs"              "bsapply"
##  [5] "BSgenome"                    "compatibleGenomes"
##  [7] "countPWM"                    "forgeBSgenomeDataPkg"
##  [9] "forgeMaskedBSgenomeDataPkg" "forgeMasksFiles"
## [11] "forgeSeqFiles"               "forgeSeqlengthsFile"
## [13] "gdapply"                     "gdReduce"
```

```
## [15] "GenomeData"                "GenomeDataList"
## [17] "getBSgenome"               "getSeq"
## [19] "injectSNPs"                "installed.genomes"
## [21] "installed.SNPs"            "MaskedBSgenome"
## [23] "masknames"                 "matchPWM"
## [25] "mseqnames"                 "newSNPlocs"
## [27] "organism"                  "provider"
## [29] "providerVersion"           "referenceGenome"
## [31] "releaseDate"               "releaseName"
## [33] "score"                     "seqinfo"
## [35] "seqinfo<-"                 "seqnames"
## [37] "seqnames<-"                "show"
## [39] "snpcount"                  "SNPcount"
## [41] "snpid2alleles"             "snpid2grange"
## [43] "snpid2loc"                 "snplocs"
## [45] "SNPlocs"                   "SNPlocs_pkgname"
## [47] "sourceUrl"                 "species"
## [49] "vcountPattern"             "vmatchPattern"
```

```
Celegans  # see some header information
```

```
## Worm genome
## |
## | organism: Caenorhabditis elegans (Worm)
## | provider: UCSC
## | provider version: ce6
## | release date: May 2008
## | release name: WormBase v. WS190
## | 7 sequences:
## |   chrI   chrII  chrIII chrIV  chrV   chrX   chrM
## | (use 'seqnames()' to see all the sequence names, use the '$' or '[['
## | operator to access a given sequence)
```

```
Celegans[["chrI"]]  # disply the first portion of chromosome I
```

```
##   15072421-letter "DNAString" instance
## seq: GCCTAAGCCTAAGCCTAAGCCTAAGCCTAAGCCT...GGCTTAGGCTTAGGCTTAGGTTTAGGCTTAGGC
```

```
class(Celegans[["chrI"]])  # this is a DNAString
```

```
## [1] "DNAString"
## attr(,"package")
## [1] "Biostrings"
```

```
str(Celegans)  # see the components--guess the names of some accessor functions
```

```
## Formal class 'BSgenome' [package "BSgenome"] with 17 slots
##   ..@ pkgname           : chr "BSgenome.Celegans.UCSC.ce6"
##   ..@ single_sequences  :Formal class 'TwobitNamedSequences' [package "BSgenome"] with 1 slot
##   .. .. ..@ twobitfile:Formal class 'TwoBitFile' [package "rtracklayer"] with 1 slot
##   .. .. .. .. ..@ resource: chr "/home/david/R/x86_64-pc-linux-gnu-library/3.1/BSgenome.Celegans.UCSC
```

```
##   ..@ multiple_sequences:Formal class 'RdaCollection' [package "XVector"] with 2 slots
##   .. .. ..@ dirpath : chr "/home/david/R/x86_64-pc-linux-gnu-library/3.1/BSgenome.Celegans.UCSC.ce6/
##   .. .. ..@ objnames: chr(0)
##   ..@ source_url        : chr "http://hgdownload.cse.ucsc.edu/goldenPath/ce6/bigZips/"
##   ..@ user_seqnames     : Named chr [1:7] "chrI" "chrII" "chrIII" "chrIV" ...
##   .. ..- attr(*, "names")= chr [1:7] "chrI" "chrII" "chrIII" "chrIV" ...
##   ..@ injectSNPs_handler:Formal class 'InjectSNPsHandler' [package "BSgenome"] with 4 slots
##   .. .. ..@ SNPlocs_pkgname          : chr(0)
##   .. .. ..@ getSNPcount              :function ()
##   .. .. ..@ getSNPlocs               :function ()
##   .. .. ..@ seqname_translation_table: chr(0)
##   ..@ .seqs_cache       :<environment: 0xbb91890>
##   ..@ .link_counts      :<environment: 0xbb91ba0>
##   ..@ nmask_per_seq     : int 0
##   ..@ masks             :Formal class 'RdaCollection' [package "XVector"] with 2 slots
##   .. .. ..@ dirpath : chr NA
##   .. .. ..@ objnames: chr(0)
##   ..@ organism          : chr "Caenorhabditis elegans"
##   ..@ species           : chr "Worm"
##   ..@ provider          : chr "UCSC"
##   ..@ provider_version  : chr "ce6"
##   ..@ release_date      : chr "May 2008"
##   ..@ release_name      : chr "WormBase v. WS190"
##   ..@ seqinfo           :Formal class 'Seqinfo' [package "GenomeInfoDb"] with 4 slots
##   .. .. ..@ seqnames   : chr [1:7] "chrI" "chrII" "chrIII" "chrIV" ...
##   .. .. ..@ seqlengths : int [1:7] 15072421 15279323 13783681 17493785 20919568 17718854 13794
##   .. .. ..@ is_circular: logi [1:7] FALSE FALSE FALSE FALSE FALSE FALSE ...
##   .. .. ..@ genome     : chr [1:7] "ce6" "ce6" "ce6" "ce6" ...
```

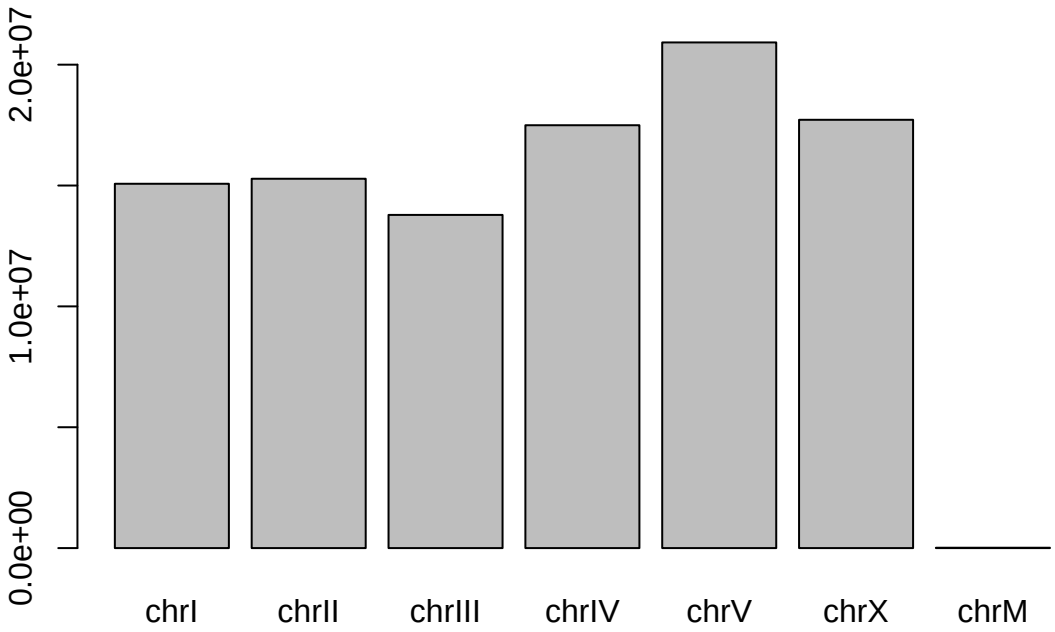
```
seqinfo(Celegans)  # get a summary of the sequences included
```

```
## Seqinfo object with 7 sequences (1 circular) from ce6 genome:
##   seqnames seqlengths isCircular genome
##   chrI       15072421      FALSE    ce6
##   chrII      15279323      FALSE    ce6
##   chrIII     13783681      FALSE    ce6
##   chrIV      17493785      FALSE    ce6
##   chrV       20919568      FALSE    ce6
##   chrX       17718854      FALSE    ce6
##   chrM          13794       TRUE    ce6
```

```
seqlengths(Celegans)  # get an array of the sequence lengths
```

```
##     chrI    chrII   chrIII    chrIV     chrV     chrX     chrM
## 15072421 15279323 13783681 17493785 20919568 17718854    13794
```

```
barplot(seqlengths(Celegans))  # generate a quick barplot
```



Computing the GC Content of *C. elegans* Chromosome I?

Functions Used

- `c()` to combine objects into a vector
- `alphabetFrequency()` to generate a table of nucleotide frequencies for a `BSgenome` DNA sequence
- `sum()` to sum positions in a vector

```
F = alphabetFrequency(Celegans[["chrI"]])  # get the frequency table

F  # take look at the table
```

```
##       A       C       G       T       M       R       W       S       Y
## 4835939 2695879 2692150 4848453       0       0       0       0       0
##       K       V       H       D       B       N       -       +       .
##       0       0       0       0       0       0       0       0       0
```

```
sum(F[c("G", "C")])/sum(F[c("A", "T", "C", "G")]) * 100  # compute % GC
```

```
## [1] 35.7476
```

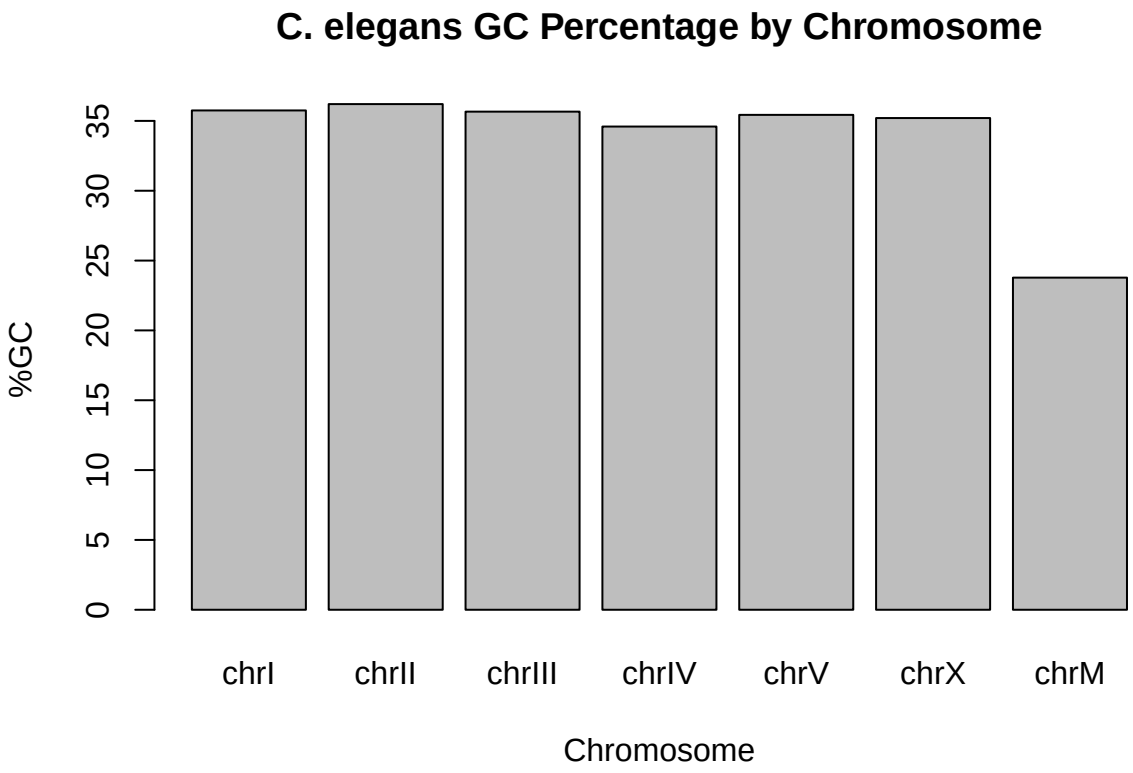
How can we compute %GC for all chromosomes?

```
S = seqnames(Celegans)  # put the 'seqnames' into S

Fr = vector()  # initialize a vector to hold the %GC values

for (s in S) {
    # loop through each of the 'seqnames'--these are the chromosomes
```

```
    F = alphabetFrequency(Celegans[[s]])  # get a table for one chromosome, 's'

    # compute %GC and store this in 'Fr[s]'

    Fr[s] = sum(F[c("G", "C")])/sum(F[c("A", "T", "C", "G")]) * 100

}


Fr  # display Fr to see if we got it right
```

```
##     chrI    chrII   chrIII    chrIV     chrV     chrX     chrM
## 35.74760 36.20196 35.66144 34.59386 35.42942 35.20301 23.77845
```

```
barplot(Fr, ylab = "%GC", xlab = "Chromosome", main = "C. elegans GC Percentage by Chromosome")  # barp
```



GenomicFeatures

The *GenomicFeatures* library provides the functions for the extraction of the sequences of features such as genes, exons, introns, and promoters from BSgenomes.

Let's take another look at the *C. elegans* genome. We already have the genome itself but let's add a transcript database to our environment.

```
library(TxDb.Celegans.UCSC.ce6.ensGene)  # load the C. elegans transcript database
```

```
## Loading required package: GenomicFeatures
```

```
## Loading required package: AnnotationDbi
##
## Attaching package: 'AnnotationDbi'
##
## The following object is masked from 'package:BSgenome':
##
##     species
##
## The following object is masked from 'package:GenomeInfoDb':
##
##     species
```

By loading this transcript database, we have also loaded a new library called `GenomicFeatures` that is required to use the database. What new functions have we acquired?

```
ls("package:GenomicFeatures")  # list the methods of 'GenomicFeatures'
```

```
##  [1] "asBED"                          "asGFF"
##  [3] "as.list"                        "cds"
##  [5] "cdsBy"                          "cdsByOverlaps"
##  [7] "DEFAULT_CIRC_SEQS"              "determineDefaultSeqnameStyle"
##  [9] "disjointExons"                  "distance"
## [11] "exons"                          "exonsBy"
## [13] "exonsByOverlaps"                "extractTranscripts"
## [15] "extractTranscriptSeqs"          "extractTranscriptsFromGenome"
## [17] "extractUpstreamSeqs"            "features"
## [19] "fiveUTRsByTranscript"           "genes"
## [21] "getChromInfoFromBiomart"        "getChromInfoFromUCSC"
## [23] "getPromoterSeq"                 "id2name"
## [25] "intronsByTranscript"            "isActiveSeq"
## [27] "isActiveSeq<-"                  "makeFDbPackageFromUCSC"
## [29] "makeFeatureDbFromUCSC"          "makeTranscriptDb"
## [31] "makeTranscriptDbFromBiomart"    "makeTranscriptDbFromGFF"
## [33] "makeTranscriptDbFromUCSC"       "makeTxDbPackage"
## [35] "makeTxDbPackageFromBiomart"     "makeTxDbPackageFromUCSC"
## [37] "metadata"                       "microRNAs"
## [39] "promoters"                      "seqinfo"
## [41] "seqinfo<-"                      "show"
## [43] "sortExonsByRank"                "supportedMiRBaseBuildValues"
## [45] "supportedUCSCFeatureDbTables"   "supportedUCSCFeatureDbTracks"
## [47] "supportedUCSCtables"            "threeUTRsByTranscript"
## [49] "transcriptLocs2refLocs"         "transcripts"
## [51] "transcriptsBy"                  "transcriptsByOverlaps"
## [53] "transcriptWidths"               "tRNAs"
## [55] "UCSCFeatureDbTableSchema"
```

Extracting and Analyzing Genomic Features

Functions Used

- `exons()` to extract exon features from a transcript database
- `exonsBy()` to extract exons, grouped by gene

- `genes()` to extract gene features
- `head()` to see the first portion a large variable
- `seqnames()` to get the chromosome assignments of features
- `strand()` to get the strand assignment of features
- `table()` to produce a table of counts for features by category
- `tail()` to see the last portion a large variable

```
ce.exons = exons(TxDb.Celegans.UCSC.ce6.ensGene)  # get the exons coordinates
head(ce.exons)  # take a look
```

```
## GRanges object with 6 ranges and 1 metadata column:
##       seqnames         ranges strand |   exon_id
##          <Rle>      <IRanges>  <Rle> | <integer>
##   [1]     chrI [11495, 11561]      + |         1
##   [2]     chrI [11499, 11557]      + |         2
##   [3]     chrI [11499, 11561]      + |         3
##   [4]     chrI [11505, 11561]      + |         4
##   [5]     chrI [11618, 11689]      + |         5
##   [6]     chrI [11623, 11689]      + |         6
##   -------
##   seqinfo: 7 sequences (1 circular) from ce6 genome
```

```
ce.genes = genes(TxDb.Celegans.UCSC.ce6.ensGene)  # get the gene coordinates
head(ce.genes)  #take a look
```

```
## GRanges object with 6 ranges and 1 metadata column:
##           seqnames              ranges strand |     gene_id
##              <Rle>           <IRanges>  <Rle> | <character>
##    2L52.1    chrII [    1867,     4663]      + |      2L52.1
##   2RSSE.1    chrII [15268114, 15273216]      + |     2RSSE.1
##   2RSSE.2    chrII [15274293, 15275591]      + |     2RSSE.2
##   2RSSE.3    chrII [15277792, 15278477]      - |     2RSSE.3
##     3R5.1   chrIII [13780108, 13781013]      + |       3R5.1
##     3R5.2   chrIII [13782460, 13783332]      + |       3R5.2
##   -------
##   seqinfo: 7 sequences (1 circular) from ce6 genome
```

```
tail(ce.genes)  #take a look at the end
```

```
## GRanges object with 6 ranges and 1 metadata column:
##            seqnames             ranges strand |     gene_id
##               <Rle>          <IRanges>  <Rle> | <character>
##    ZK994.3     chrV [8494873, 8502138]      + |     ZK994.3
##    ZK994.6     chrV [8502569, 8504559]      - |     ZK994.6
##    ZK994.7     chrV [8500557, 8500623]      + |     ZK994.7
##   ZK994.t1     chrV [8495280, 8495352]      + |    ZK994.t1
##   ZK994.t2     chrV [8499542, 8499614]      - |    ZK994.t2
##   ZK994.t3     chrV [8496082, 8496153]      - |    ZK994.t3
##   -------
##   seqinfo: 7 sequences (1 circular) from ce6 genome
```

```
table(strand(ce.genes))  # see gene counts by strand
```

```
##
##     +     -     *
## 14077 13851     0
```

```
table(seqnames(ce.genes))  # see gene counts by chromosome
```

```
##
##   chrI  chrII chrIII  chrIV   chrV   chrX   chrM
##   3139   3784   2875   9052   5890   3164     24
```

The gene_ids in the single metadata column are ENSEMBL ids and can be used to search in the UCSC Genome Browser.

```
# get the exons with their gene assignments
ce.exons.bygene = exonsBy(TxDb.Celegans.UCSC.ce6.ensGene, by = c("gene"))
head(ce.exons.bygene)  # take a look
```

```
## GRangesList object of length 6:
## $2L52.1
## GRanges object with 7 ranges and 2 metadata columns:
##       seqnames       ranges strand |   exon_id   exon_name
##          <Rle>    <IRanges>  <Rle> | <integer> <character>
##   [1]     chrII [1867, 1911]      + |     21558        <NA>
##   [2]     chrII [2506, 2694]      + |     21559        <NA>
##   [3]     chrII [2738, 2888]      + |     21560        <NA>
##   [4]     chrII [2931, 3036]      + |     21561        <NA>
##   [5]     chrII [3406, 3552]      + |     21562        <NA>
##   [6]     chrII [3802, 3984]      + |     21563        <NA>
##   [7]     chrII [4201, 4663]      + |     21564        <NA>
##
## $2RSSE.1
## GRanges object with 5 ranges and 2 metadata columns:
##       seqnames               ranges strand | exon_id exon_name
##   [1]     chrII [15268114, 15268456]      + |   32895      <NA>
##   [2]     chrII [15269361, 15269696]      + |   32896      <NA>
##   [3]     chrII [15269762, 15269933]      + |   32897      <NA>
##   [4]     chrII [15270698, 15270875]      + |   32898      <NA>
##   [5]     chrII [15272945, 15273216]      + |   32899      <NA>
##
## $2RSSE.2
## GRanges object with 4 ranges and 2 metadata columns:
##       seqnames               ranges strand | exon_id exon_name
##   [1]     chrII [15274293, 15274418]      + |   32900      <NA>
##   [2]     chrII [15274931, 15275206]      + |   32901      <NA>
##   [3]     chrII [15275254, 15275361]      + |   32902      <NA>
##   [4]     chrII [15275436, 15275591]      + |   32903      <NA>
##
## ...
## <3 more elements>
## -------
## seqinfo: 7 sequences (1 circular) from ce6 genome
```

GenomicRanges

The *GenomicRanges* library provides the framework for feature extraction and analysis. Among its most powerful functions are those used to determine feature overlap.

We load the library as usual:

```
library("GenomicRanges")  # load the library 'GenomicRanges'
```

And we list the newly acquired functions in the usual manner:

```
ls("package:GenomicRanges")  # list the methods of 'GenomicRanges'
```

```
##   [1] "as.data.frame"
##   [2] "assay"
##   [3] "assay<-"
##   [4] "assays"
##   [5] "assays<-"
##   [6] "cbind"
##   [7] "checkConstraint"
##   [8] "coerce"
##   [9] "colData"
##  [10] "colData<-"
##  [11] "compare"
##  [12] "countOverlaps"
##  [13] "coverage"
##  [14] "disjoin"
##  [15] "disjointBins"
##  [16] "distance"
##  [17] "distanceToNearest"
##  [18] "duplicated"
##  [19] "duplicated.GenomicRanges"
##  [20] "elementMetadata"
##  [21] "elementMetadata<-"
##  [22] "end"
##  [23] "end<-"
##  [24] "exptData"
##  [25] "exptData<-"
##  [26] "findOverlaps"
##  [27] "flank"
##  [28] "follow"
##  [29] "gaps"
##  [30] "GenomicRangesList"
##  [31] "GIntervalTree"
##  [32] "granges"
##  [33] "GRanges"
##  [34] "GRangesList"
##  [35] "grglist"
##  [36] "intersect"
##  [37] "isDisjoint"
##  [38] "makeGRangesFromDataFrame"
##  [39] "makeGRangesListFromFeatureFragments"
##  [40] "makeSeqnameIds"
##  [41] "map"
```

```
##  [42] "mapCoords"
##  [43] "match"
##  [44] "mcols"
##  [45] "mcols<-"
##  [46] "merge"
##  [47] "narrow"
##  [48] "nearest"
##  [49] "ngap"
##  [50] "Ops"
##  [51] "order"
##  [52] "overlapsAny"
##  [53] "pgap"
##  [54] "phicoef"
##  [55] "pintersect"
##  [56] "precede"
##  [57] "promoters"
##  [58] "psetdiff"
##  [59] "punion"
##  [60] "queryHits"
##  [61] "ranges"
##  [62] "ranges<-"
##  [63] "rank"
##  [64] "rbind"
##  [65] "reduce"
##  [66] "relistToClass"
##  [67] "resize"
##  [68] "restrict"
##  [69] "rglist"
##  [70] "rowData"
##  [71] "rowData<-"
##  [72] "score"
##  [73] "score<-"
##  [74] "seqinfo"
##  [75] "seqinfo<-"
##  [76] "seqnames"
##  [77] "seqnames<-"
##  [78] "setdiff"
##  [79] "shift"
##  [80] "show"
##  [81] "sort"
##  [82] "sort.GenomicRanges"
##  [83] "split"
##  [84] "splitAsListReturnedClass"
##  [85] "start"
##  [86] "start<-"
##  [87] "strand"
##  [88] "strand<-"
##  [89] "subjectHits"
##  [90] "subset"
##  [91] "subsetByOverlaps"
##  [92] "SummarizedExperiment"
##  [93] "tile"
##  [94] "tileGenome"
##  [95] "trim"
```

```
##  [96] "union"
##  [97] "unlist"
##  [98] "updateObject"
##  [99] "values"
## [100] "values<-"
## [101] "width"
## [102] "width<-"
```

Constructing a Histogram of Exon Lengths

Functions Used

- `length()` to get the number of exons contained in the `ce.exons` object
- `head()` to see part of the output of `width()`
- `hist()` to create and plot a histogram of the exon widths
- `width()` to get a vector of the exon widths

The `width()` function can be used to make an array of exon widths, the distribution of which we can then plot as histogram.

```
str(ce.exons)  # see the structure of the ce.exons GRanges object
```

```
## Formal class 'GRanges' [package "GenomicRanges"] with 6 slots
##   ..@ seqnames       :Formal class 'Rle' [package "S4Vectors"] with 4 slots
##   .. .. ..@ values         : Factor w/ 7 levels "chrI","chrII",..: 1 2 3 4 7 5 6
##   .. .. ..@ lengths        : int [1:7] 21557 22940 19513 28144 24 31825 22830
##   .. .. ..@ elementMetadata: NULL
##   .. .. ..@ metadata       : list()
##   ..@ ranges         :Formal class 'IRanges' [package "IRanges"] with 6 slots
##   .. .. ..@ start          : int [1:146833] 11495 11499 11499 11505 11618 11623 14951 16473 16473 164
##   .. .. ..@ width          : int [1:146833] 67 59 63 57 72 67 210 113 318 332 ...
##   .. .. ..@ NAMES          : NULL
##   .. .. ..@ elementType    : chr "integer"
##   .. .. ..@ elementMetadata: NULL
##   .. .. ..@ metadata       : list()
##   ..@ strand         :Formal class 'Rle' [package "S4Vectors"] with 4 slots
##   .. .. ..@ values         : Factor w/ 3 levels "+","-","*": 1 2 1 2 1 2 1 2 1 2 ...
##   .. .. ..@ lengths        : int [1:12] 10972 10585 11346 11594 9873 9640 14639 13505 15833 16016 ..
##   .. .. ..@ elementMetadata: NULL
##   .. .. ..@ metadata       : list()
##   ..@ elementMetadata:Formal class 'DataFrame' [package "S4Vectors"] with 6 slots
##   .. .. ..@ rownames       : NULL
##   .. .. ..@ nrows          : int 146833
##   .. .. ..@ listData       :List of 1
##   .. .. .. ..$ exon_id: int [1:146833] 1 2 3 4 5 6 7 8 9 10 ...
##   .. .. ..@ elementType    : chr "ANY"
##   .. .. ..@ elementMetadata: NULL
##   .. .. ..@ metadata       : list()
##   ..@ seqinfo        :Formal class 'Seqinfo' [package "GenomeInfoDb"] with 4 slots
##   .. .. ..@ seqnames   : chr [1:7] "chrI" "chrII" "chrIII" "chrIV" ...
##   .. .. ..@ seqlengths : int [1:7] 15072421 15279323 13783681 17493785 20919568 17718854 13794
##   .. .. ..@ is_circular: logi [1:7] FALSE FALSE FALSE FALSE FALSE FALSE ...
```

```
##   .. .. ..@ genome     : chr [1:7] "ce6" "ce6" "ce6" "ce6" ...
##   ..@ metadata       :List of 1
##   .. ..$ genomeInfo:List of 18
##   .. .. ..$ Db type                                  : chr "TxDb"
##   .. .. ..$ Supporting package                       : chr "GenomicFeatures"
##   .. .. ..$ Data source                              : chr "UCSC"
##   .. .. ..$ Genome                                   : chr "ce6"
##   .. .. ..$ Organism                                 : chr "Caenorhabditis elegans"
##   .. .. ..$ UCSC Table                               : chr "ensGene"
##   .. .. ..$ Resource URL                             : chr "http://genome.ucsc.edu/"
##   .. .. ..$ Type of Gene ID                          : chr "Ensembl gene ID"
##   .. .. ..$ Full dataset                             : chr "yes"
##   .. .. ..$ miRBase build ID                         : chr NA
##   .. .. ..$ transcript_nrow                          : chr "35019"
##   .. .. ..$ exon_nrow                                : chr "146833"
##   .. .. ..$ cds_nrow                                 : chr "127881"
##   .. .. ..$ Db created by                            : chr "GenomicFeatures package from Bioconductor
##   .. .. ..$ Creation time                            : chr "2014-09-26 11:21:48 -0700 (Fri, 26 Sep 20
##   .. .. ..$ GenomicFeatures version at creation time: chr "1.17.17"
##   .. .. ..$ RSQLite version at creation time         : chr "0.11.4"
##   .. .. ..$ DBSCHEMAVERSION                          : chr "1.0"
```

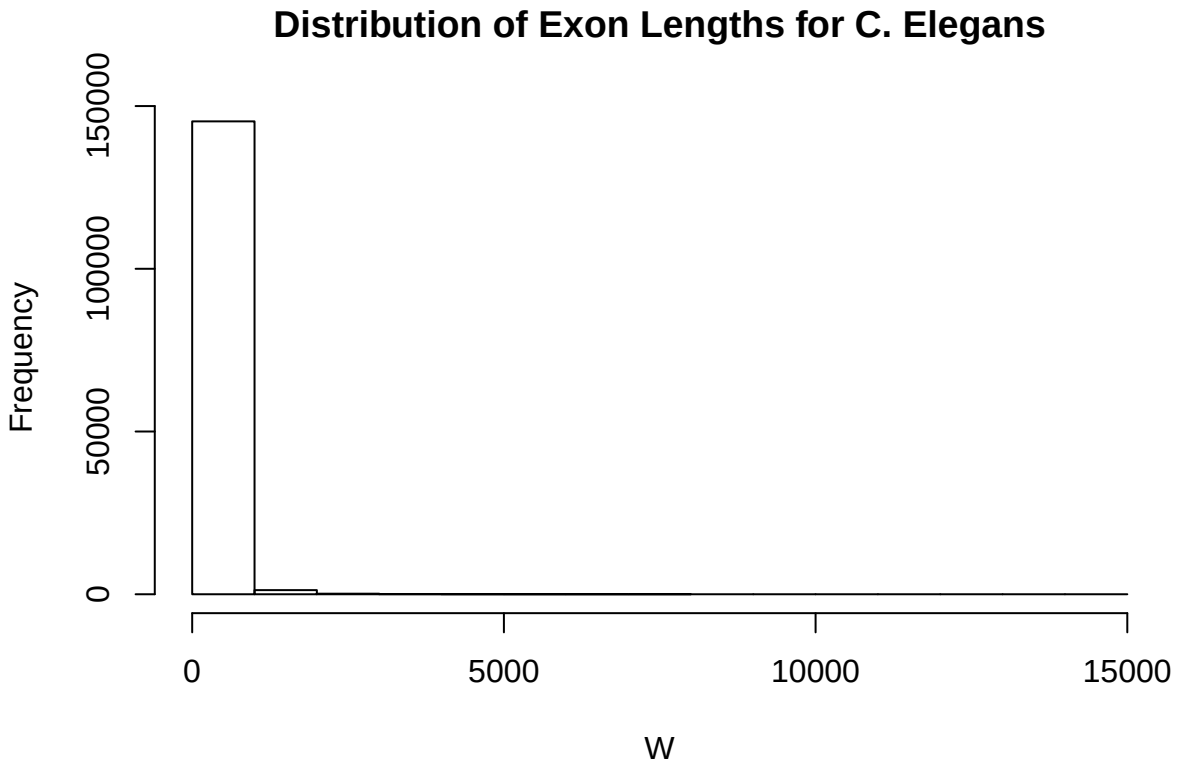
```
length(ce.exons)  # how many exons are there in C. elegans?
```

```
## [1] 146833
```

```
W = width(ce.exons)  # get the lengths for all exons

head(W)  # take a look
```

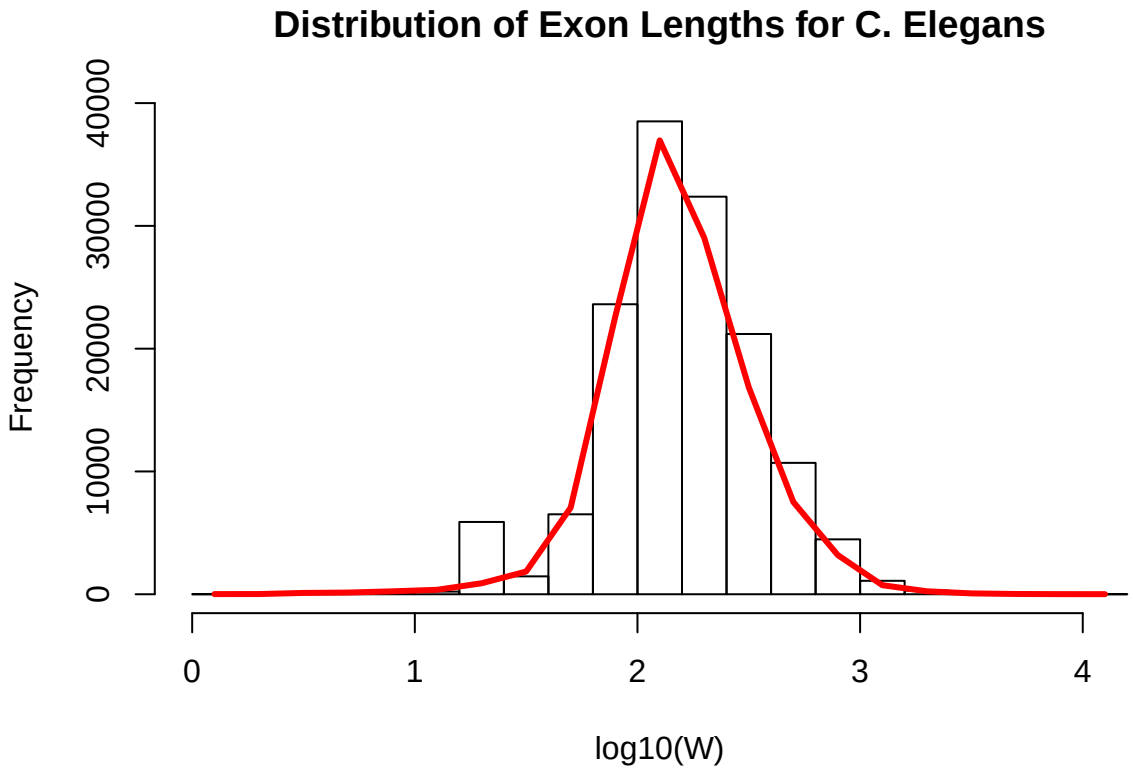
```
## [1] 67 59 63 57 72 67
```

```
hist(W, main = "Distribution of Exon Lengths for C. Elegans")  # compute and plot the histogram
```



This histogram looks a bit cramped. How can we uncramp it?

```
hist(log10(W), main = "Distribution of Exon Lengths for C. Elegans")


ce.cds = cds(TxDb.Celegans.UCSC.ce6.ensGene)  # get the C. elegans CDS coordinates

W2 = width(ce.cds)  # get the CDS widths

H = hist(log10(W2), plot = FALSE)  # comput the second histogram but don't plot it--save it in 'H'

lines(H$mids, H$counts, col = "red", lwd = 3)  # plot the second histogram using coordinates taken from
```



For the most part, the two histograms track one another, however at the lower end of the distribution there appears to be a class of short exons with even shorter CDS's.

GenomicAlignments

The *GenomicAlignments* library provides containers for storing and manipulating short genomic alignments (typically obtained by aligning short reads to a reference genome). This includes read counting, computing the coverage, junction detection, and working with the nucleotide sequences of the alignments.

```
library(GenomicAlignments)
```

```
## Loading required package: Rsamtools
```

What new functions have we acquired?

The *GenomicAlignments* library pulls in *Rsamtools* along with it. The latter allows us to read in *BAM* (Binary Alignment Map) files. Let's read one in from the sample data. Here we will read it directly from its location on disk just as one would a locally generated *BAM* file.

```
bam <- readGAlignments(system.file("extdata", "ex1.bam", package = "Rsamtools"))  # read in the file fr

bam  # take a look
```

```
## GAlignments object with 3271 alignments and 0 metadata columns:
##          seqnames strand       cigar    qwidth     start       end
##             <Rle>  <Rle> <character> <integer> <integer> <integer>
##      [1]     seq1      +         36M        36         1        36
##      [2]     seq1      +         35M        35         3        37
```

```
##      [3]     seq1      +         35M        35         5        39
##      [4]     seq1      +         36M        36         6        41
##      [5]     seq1      +         35M        35         9        43
##      ...      ...    ...         ...       ...       ...       ...
##   [3267]     seq2      +         35M        35      1524      1558
##   [3268]     seq2      +         35M        35      1524      1558
##   [3269]     seq2      -         35M        35      1528      1562
##   [3270]     seq2      -         35M        35      1532      1566
##   [3271]     seq2      -         35M        35      1533      1567
##              width     njunc
##          <integer> <integer>
##      [1]        36         0
##      [2]        35         0
##      [3]        35         0
##      [4]        36         0
##      [5]        35         0
##      ...       ...       ...
##   [3267]        35         0
##   [3268]        35         0
##   [3269]        35         0
##   [3270]        35         0
##   [3271]        35         0
##   -------
##   seqinfo: 2 sequences from an unspecified genome
```

One way to get a quick overview of the nature of the alignments in the *BAM* file is to tabulate the CIGAR strings.

```
sort(table(cigar(bam)), decreasing = TRUE)  # make a table of the CIGAR strings for the alignments in t
```

```
##
##      35M      36M      40M      34M      33M 14M4I17M 15M4I16M 16M4I15M
##     2804      283      112       37        6        4        4        3
##  7M4I24M 17M4I14M  9M2I24M 10M4I21M 11M5I19M 12M4I19M 13M1D22M 13M4I18M
##        3        2        2        1        1        1        1        1
## 14M5I17M 18M4I13M 18M5I12M  7M2I31M  8M4I24M  9M1D26M
##        1        1        1        1        1        1
```

Counting Overlaps Between Features

Functions Used

- `c()` to combine objects into a vector
- `encodeOverlaps()` to categorize the overlaps between a reference and query features
- `isCompatibleWithSplicing()` to flag overlaps as compatible with splicing
- `GRanges()` to create a `GRanges` object
- `GRangesList()` to create a `GRangesList` object from a set of `GRanges` objects
- `length()` to get the number of ranges in a `GRanges` or `GRangesList` object
- `plot()` to set up a plot window with proper coordinates
- `segment()` to draw some lines based on feature starts and widths
- `start()` to get feature starts
- `width()` to get feature widths

Overlaps between aligned reads and the known transcripts of a gene can be computed using `encodeOverlaps`. To illustrate the principle, we will actually create alignment ranges for four reads and compute the overlaps with a single gene model. Let's take a quick look at a couple of functions for detecting and assessing feature overlaps: `encodeOverlaps` and `isCompatibleWithSplicing`. We'll use a simplified system to make the process clearer.

First, we'll create a GRanges object from scratch with the start and end points for an alignment.

```
# make a GRanges object: a read with three segments of alignment on 'Seq1',
# strand '+'

sr1 = GRanges(ranges = IRanges(c(7, 15, 22), c(9, 19, 23)), seqnames = c("Seq1"),
    strand = c("+"))
sr1
```

```
## GRanges object with 3 ranges and 0 metadata columns:
##       seqnames    ranges strand
##          <Rle> <IRanges>  <Rle>
##   [1]     Seq1  [ 7,  9]      +
##   [2]     Seq1  [15, 19]      +
##   [3]     Seq1  [22, 23]      +
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

The, we'll create one more to round out our very small library of very short reads.

```
# make one more--this one has only two segments of alignment
sr2 = GRanges(ranges = IRanges(c(16, 23), c(19, 24)), seqnames = c("Seq1"),
    strand = c("+"))
```

Using the same method, we can create a transcript model with 5 exons:

```
# make a GRanges objects: a transcript model with 5 segments on 'Seq1',
# strand '+'

tx <- GRanges(ranges = IRanges(c(1, 4, 15, 22, 38), c(2, 9, 19, 25, 47)), seqnames = c("Seq1"),
    strand = c("+"))
```

Finally, we can package the 4 reads into one GRangesList and the transcript model into a second GRanges list.

```
srlib = GRangesList(sr1, sr2)  # create our short read library as a GRangesList

ref = GRangesList(tx)  # create our transcript model reference as a second GRangesList
```

We are now ready to analyze the overlaps between the reads in the first list and the transcript model in the second:

```
overlaps = encodeOverlaps(srlib, ref)  # find and classify overlaps bewtween short reads and model

overlaps  # take a look at the result
```

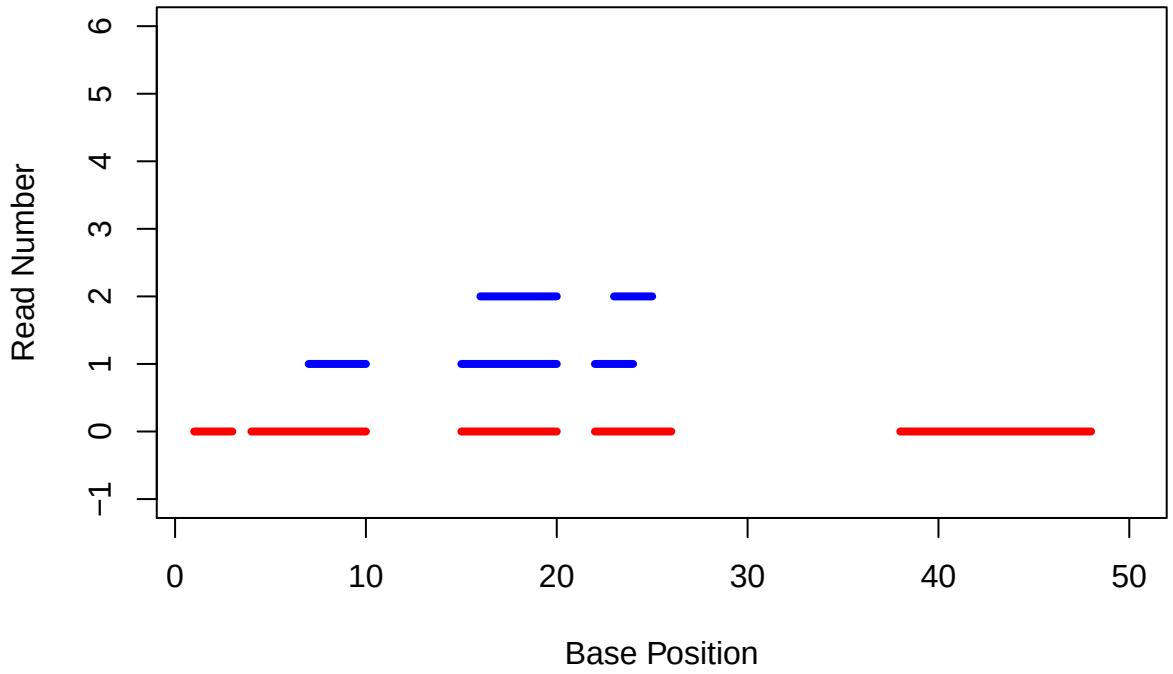
```
## OverlapEncodings object of length 2
##     Loffset Roffset       encoding flippedQuery
## [1]       1       1 3:jmm:agm:aaf:        FALSE
## [2]       2       1       2:jm:ai:        FALSE
```

Let's now use `isCompatibleWithSplicing` to see which read alignments are compatible with the model. We'll also plot the transcript model with the reads to better understand what is happening.

```
isCompatibleWithSplicing(overlaps)  # flag the short read overlaps by compatibility with the transcript
```

```
## [1]  TRUE FALSE
```

```
# Plot the transcript model in red

plot(c(1, 50), c(-1, 6), type = "n", xlab = "Base Position", ylab = "Read Number")
for (j in 1:length(ref)) {
    for (i in 1:length(ref[[j]])) {
        segments(start(ref[[j]][i]), 1 - j, start(ref[[j]][i]) + width(ref[[j]][i]),
            1 - j, col = "red", lwd = 4)
    }
}

# plot the 4 short reads in blue with y-coordinate=read number

for (j in 1:length(srlib)) {
    for (i in 1:length(srlib[[j]])) {
        segments(start(srlib[[j]][i]), j, start(srlib[[j]][i]) + width(srlib[[j]][i]),
            j, col = "blue", lwd = 4)
    }
}
```



Why is the alignment for short read number 2 incompatible with the transcript model?

The SummarizedExperiment

The `SummarizedExperiment` is designed to contain multiple matrices of counts generated by NGS analysis where the columns represent samples and the rows represent the genomic features to which the counts apply. The samples are annotated in a manner similar to that used in the `ExpressionSet`–that is with a *data.frame* that puts the sample names in the rows and the covariate data in the columns. Using this method, the samples can be easily grouped for analysis.

Constructing an SummarizedExperiment

Functions Used

- `Data.Frame()` to construct our `colData` of covariate data
- `GRanges()` to construct the features of our *SummarizedExperiment*
- `IRanges()` to construct the feature ranges within our *GRanges* object
- `SimpleList()` to construct a list of assay matrices
- `SummarizedExperiment()` to assemble our *SummarizedExperiment* form its components
- `assays()` to extract a single counts matrix from our *SummarizedExperiment*
- `c()` to combine coordinates and various character strings into vectors
- `colData()` to extract the covariate data from our *SummarizedExperiment*
- `floor()` to discard the fractional part of the numbers generated by `runif()`
- `head()` to see the first part of our assay matrices
- `matrix()` to create a matrix as an argument to `layout()`
- `rep()` to repeat 'chr1' 20 times and 'chr2' 30 times
- `rnorm()` to generate random counts for the assay matrices
- `rowData()` to extract the feature data from our *SummarizedExperiment*
- `runif()` to generate random coordinates for the features
- `sample()` to generate random strand assignments by sampling from the set ("+","-")

A `SummarizedExperiment` is comprised of 4 slots:

assays()	rowData()	colData()	exptData
matrix or list of matrices	GRanges, GrangesList	data.frame	list
counts	feature ranges	sample information	description of experiment

We'll build a `SummarizedExperiment` from scratch and then analyze it as we did in the case of the `ExpressionSet`. This time, however, we will fabricate the data in such a way that each `SummarizedExperiment` we build will be a little different. We'll build a small set of two count matrices of 50 features and 4 samples.

```
nrows = 500
ncols = 4  # set the dimensions for the count matrices

# fill the first matrix with the absolute values of three superimposed
# normal random variates with increasing SD

# a skewed distribution

counts = c(rnorm(1800, 0, 1000), runif(200, 4000, 10000))  # create 2000 counts using two random distri
```

```
counts = floor(abs(counts))  # convert negative counts to positive and make all integers
counts = matrix(counts, nrows)  # package the 2000 counts into a matrix of 'nrows'


# this one is not quite as skewed

counts2 = c(rnorm(1800, 0, 1000), runif(200, 3000, 8000))  # create 2000 counts using two random distri
counts2 = floor(abs(counts2))  # convert negative counts to positive and make all integers
counts2 = matrix(counts2, nrows)  # package the 2000 counts into a matrix of 'nrows'


# set up our features, GRanges objects, as the rows: 200 on chr1, 300 on
# chr2; starting coordinates will range from 100,000 to 1,000,000, all
# widths will be 100, strands will be chosen randomly

rowData = GRanges(seqnames = rep(c("chr1", "chr2"), c(200, 300)), ranges = IRanges(floor(runif(500,
    1e+05, 1e+06)), width = 100), strand = sample(c("+", "-"), 500, TRUE))

# call the samples A-D, label them as two pairs of (Tumor,Normal)

colData = DataFrame(type = rep(c("Tum", "Norm"), 2), row.names = LETTERS[1:4])

# Create the `SummarizedExperiment`

se = SummarizedExperiment(assays = SimpleList(run1 = counts, run2 = counts2),
    rowData = rowData, colData = colData)
se
```

```
## class: SummarizedExperiment
## dim: 500 4
## exptData(0):
## assays(2): run1 run2
## rownames: NULL
## rowData metadata column names(0):
## colnames(4): A B C D
## colData names(1): type
```

Let's see what we've accomplished by using the accessor functions (table above) to check the contents of `se`.

```
assays(se)  # see that we have 2 assays in the `SummarizedExperiment`
```

```
## List of length 2
## names(2): run1 run2
```

```
head(assays(se)$run1)  # take a look at assay 'run1'
```

```
##         A    B    C    D
## [1,]  215 1050  298  226
## [2,] 1485   32 1331  289
## [3,]  635 1142  692 1360
## [4,]  610  499  297  645
## [5,]  386  796 1000  600
## [6,]  182  460 1040  156
```

```
colData(se)  # take a look at the samples and their covariates
```

```
## DataFrame with 4 rows and 1 column
##          type
##   <character>
## A         Tum
## B        Norm
## C         Tum
## D        Norm
```

```
rowData(se)  # take a look at the features
```

```
## GRanges object with 500 ranges and 0 metadata columns:
##         seqnames           ranges strand
##            <Rle>        <IRanges>  <Rle>
##     [1]     chr1 [335862, 335961]      +
##     [2]     chr1 [644291, 644390]      +
##     [3]     chr1 [519050, 519149]      +
##     [4]     chr1 [967387, 967486]      +
##     [5]     chr1 [558668, 558767]      +
##     ...      ...               ...    ...
##   [496]     chr2 [639519, 639618]      -
##   [497]     chr2 [302517, 302616]      +
##   [498]     chr2 [874144, 874243]      +
##   [499]     chr2 [720672, 720771]      +
##   [500]     chr2 [259951, 260050]      +
##   -------
##   seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

Analyzing the Data in Our SummarizedExperiment

Functions Used

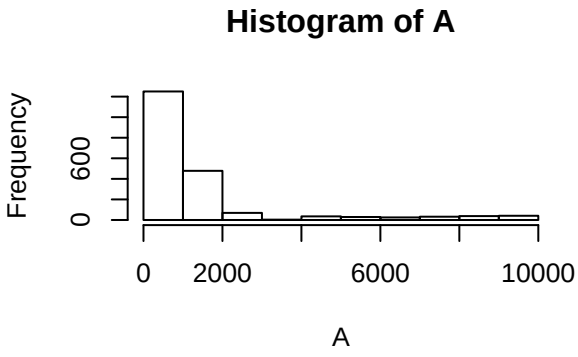
- `asinh()` to reduce the range of our assay data while allowing negative and "0" counts
- `assays()` to extract a single matrix of counts from our *SummarizedExperiment*
- `boxplot()` to boxplot and compare sample counts by covariate (tumor,normal)
- `colData()` to extract sample covariate vectors from our *SummarizedExperiment*
- `layout()` to partition the graphics display, allowing us to display 12 graphs at once
- `t.test()` to perform a T-test on the sample means in the two covariate groups

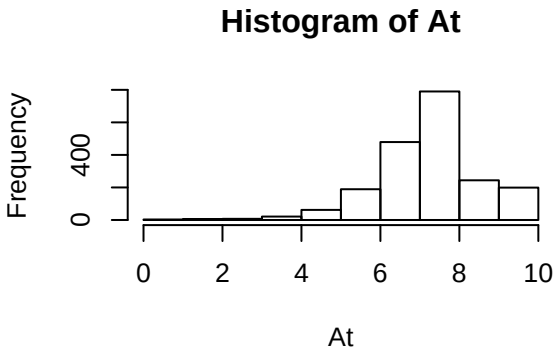
Now we're ready for a bit of analysis. By now I'm sure you know what's coming.... Yes, let's make a boxplot and run a T-test. We'll first transform the counts using `asinh()` (inverse hyperbolic sine). This is similar to a `log()` transform and is equivalent to:

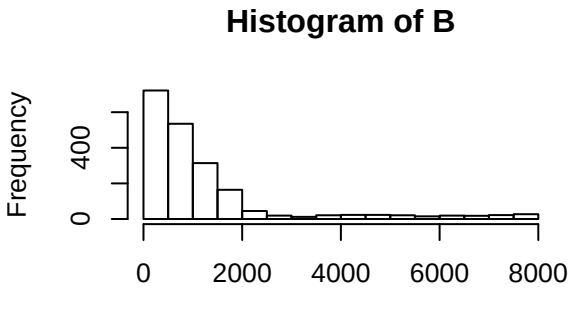
$$log(y + sqrt(y^2 + 1))$$

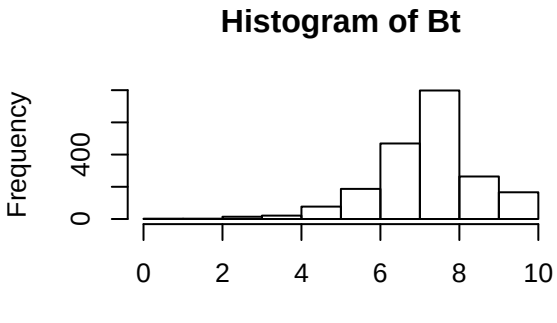
The `asinh()` transform is also performed for the same reason as the `log()` transform–the range of the data is quite large and outliers can easily skew the analysis. The advantage of the `asinh()` transform is that, unlike a `log()` transform, it is defined for a count of zero. Let's see how it works on our data. We'll plot the histograms of each assay matrix both with and without the `asinh()` transform.

```
# give the numbers of the graphs in the first argument, then the number rows
# and columns for the grid as the next two

layout(matrix(c(1, 2, 3, 4), 2, 2))

A = assays(se)$run1  # get the run1 data
B = assays(se)$run2  # get the run2 data
At = asinh(assays(se)$run1)  # transform the run1 data
Bt = asinh(assays(se)$run2)  # transform the run2 data

hist(A)
hist(B)
hist(At)
hist(Bt)
```



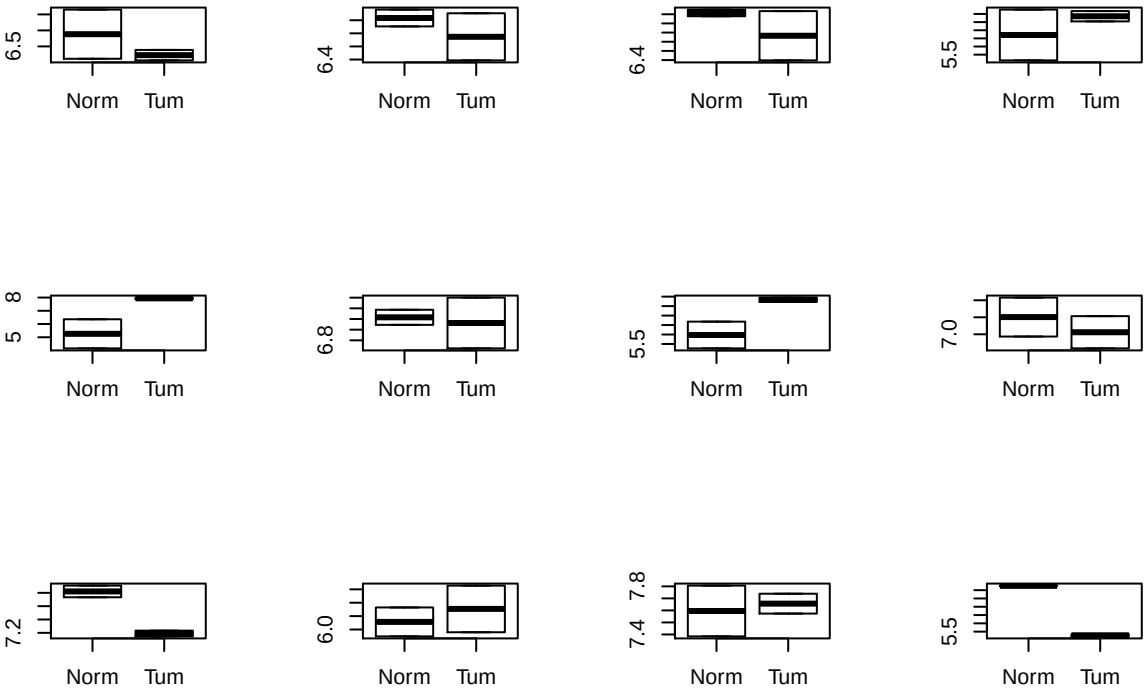






Now let's generate boxplots by (tumor/normal) status and look perform a T-test on the sample means in the case of the counts for feature number 1.

```
groups = colData(se)$type

layout(matrix(1:12, 3, 4))

for (i in 1:6) {
    boxplot(At[i, ] ~ groups)
}
for (i in 1:6) {
    boxplot(Bt[i, ] ~ groups)
}
```




```
t.test(A[1, ] ~ groups)
```

```
##
##  Welch Two Sample t-test
##
## data:  A[1, ] by groups
## t = 0.9213, df = 1.02, p-value = 0.5236
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -4639.016  5402.016
## sample estimates:
## mean in group Norm  mean in group Tum
##              638.0              256.5
```

```
t.test(B[1, ] ~ groups)
```

```
##
##  Welch Two Sample t-test
##
## data:  B[1, ] by groups
## t = 0.8699, df = 1.089, p-value = 0.5344
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2837.382  3350.382
## sample estimates:
## mean in group Norm  mean in group Tum
##              844.0              587.5
```

```
t.test(At[1, ] ~ groups)
```

```
##
##  Welch Two Sample t-test
##
## data:  At[1, ] by groups
## t = 0.8338, df = 1.09, p-value = 0.548
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -7.568464  8.877810
## sample estimates:
## mean in group Norm  mean in group Tum
##           6.881690           6.227017
```

```
t.test(Bt[1, ] ~ groups)
```

```
##
##  Welch Two Sample t-test
##
## data:  Bt[1, ] by groups
## t = 0.9176, df = 1.036, p-value = 0.523
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.848374  6.843629
## sample estimates:
## mean in group Norm  mean in group Tum
##           7.428681           6.931054
```

Appendix: Function Index by Frequency

Here is the tally of the functions appearing in the "Functions Used" sections. The tallies heavily reflect the content of the course but also partially reflect the essential nature of some functions. In any case, the list and the tallies may be of interest.

```
# read in the table of counts and set colnames

F = read.table("~/Rcourse/R-Bioc.counts", col.names = c("fun", "counts"))

F[order(-F$count, F$fun), ]  # order the entries in F
```

```
##                          fun counts
## 18                    head()      5
## 47                       c()      4
## 54                    data()      3
## 15                  length()      3
## 3         alphabetFrequency()      2
## 62                  assays()      2
## 10                 boxplot()      2
## 36                 colData()      2
## 5            consensusMatrix()      2
## 58                 GRanges()      2
```

```
## 21                   layout()      2
## 9                     names()      2
## 63                     plot()      2
## 44                    table()      2
## 65                     tail()      2
## 56                   t.test()      2
## 55                    which()      2
## 27                    width()      2
## 41                  AAString()     1
## 11                    abline()     1
## 38                    array()      1
## 50                    asinh()      1
## 72                  barplot()      1
## 14                    class()      1
## 13                 colnames()      1
## 64                 colSums()      1
## 39               complement()      1
## 20                   cutree()      1
## 4                 data.frame()     1
## 8                 Data.Frame()     1
## 37    dinucleotideFrequency()     1
## 40                      dist()     1
## 46                DNAString()     1
## 66           encodeOverlaps()     1
## 23                    exons()      1
## 61                  exonsBy()      1
## 12            ExpressionSet()     1
## 30                    exprs()      1
## 34                   factor()      1
## 26              featureNames()     1
## 2                     floor()      1
## 71                   format()      1
## 43                    genes()      1
## 51              GRangesList()     1
## 57                   hclust()      1
## 19                     hist()      1
## 29                  IRanges()      1
## 60 isCompatibleWithSplicing()     1
## 1                   library()      1
## 74                       ls()      1
## 70                   matrix()      1
## 49                      new()      1
## 42 oligonucleotideFrequency()     1
## 73                    paste()      1
## 53                  rainbow()      1
## 48                      rep()      1
## 52                  reverse()      1
## 45        reverseComplement()     1
## 69                RNAString()     1
## 32                    rnorm()      1
## 7                   rowData()      1
## 59                    runif()      1
## 31                   sample()      1
## 68                  segment()      1
```

```
## 17                 seqnames()      1
## 22               SimpleList()      1
## 67                    start()      1
## 25                      str()      1
## 35                   strand()      1
## 33                      sum()      1
## 28     SummarizedExperiment()      1
## 24                 translate()      1
## 6    trinucleotideFrequency()      1
## 16               varMetadata()      1
```