Lab: An introduction to Bioconductor

Martin Morgan

14 May, 2007

Take one of these paths through the lab:

1. Tour 1 visits a few of the major packages in Bioconductor, much like in the accompanying lecture. There is a lot of material here; you are unlikely to traverse it all.

2. Tour 2 provides an introduction to packages and their structure, and is most useful for those wanting, in the long term, to effectively use R's package mechanism to organize their projects.

1 The package tour

1.1 Data input

The Affymetrix expression platform produces CEL files. Each CEL file corresponds to a single chip, and several chips typically constitute an experiment. The objective of this section is to read several CEL files into data structures that Bioconductor uses.

Exercise 1
Load packages required for the task.

We will read cell files using the R package affy. A second R package, affydata, contains sample data files.

```
> library(affy)
> library(affydata)
```

An error such as

```
Error in library(affydata) : there is no package called 'affydata'
```

means that the package is not yet installed. To install a package, the best route is

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

Packages often have *dependencies*, i.e., relying on other packages to perform their work. The `biocLite` function is smart enough to identify all dependencies, so that a command like that above may result in several packages being installed.

Exercise 2
Tell R where our CEL files are located.

The `system.file` command is a convenient way to locate the path to installed packages; normally you would use `file.path` to construct the appropriate path.

```
> celPath <- system.file("celfiles", package = "affydata")
```

Windows users will likely want to use a `/` rather than `\` as separators in the file path (`file.path` follows this convention). This is because in R the `\` character has special meaning, and typically replaces the character following the `\` with something unexpected.

Exercise 3
With the file path in hand, read the data into R.

```
> affyBatch <- ReadAffy(celfile.path = celPath)
```

Find out about `ReadAffy` from the help page

```
> help("ReadAffy")
```

This represents one way to read CEL data into Bioconductor; different packages may expect different data, and expect different data input methods. The recommended way for accessing data is often contained in a package *vignette*, accessible with the command

```
> openVignette()
```

(evaluated after the library is loaded) or, on Windows, through the 'Vignettes' menu available after Biobase is loaded.

Exercise 4
Make sure the data look reasonable.

At a minimum, take a look at the data object you have just created:

```
> affyBatch
```

```
AffyBatch object
size of arrays=712x712 features (13 kb)
cdf=HG-U133A (22283 affyids)
number of samples=2
number of genes=506944
annotation=hgu133a
notes=
```

AffyBatch is actually kind of smart. To be informative, it requires information about the chip used in the experiment. The CEL files contain an annotation indicating the chip, but the description of the chip is available in a separate package. AffyBatch checks to see whether the package with the annotation data is installed, and if not visits the Bioconductor site to retrieve and install it. Cool.

1.2 Preprocessing

Expression data typically requires pre-processing steps such as background correction and normalization. In general preprocessing is a data reduction, from the probes on Affymetrix chip to summaries of probe sets.

Exercise 5
Use the R help and vignette system to investigate how to perform background correction, normalization, PM correction, and probe summarization.

The key function in the affy package is expresso, described most clearly in the affy vignette. A more direct though less flexible route from CEL file to preprocessed data is through justRMA. Here we change the 'working directory' to the location of the CEL files, then evaluate justRMA:

```
> currentDirectory <- getwd()
> setwd(celPath)
> celFiles <- list.files(celPath)
> expressionSet <- justRMA(filenames = celFiles)
```

```
Background correcting
Normalizing
Calculating Expression
```

```
> setwd(currentDirectory)
```

Pre-processing produces a qualitatively different type of data, and this is represented in many Bioconductor packages as an *ExpressionSet*:

```
> expressionSet
```

```
ExpressionSet (storageMode: lockedEnvironment)
assayData: 22283 features, 2 samples
  element names: exprs, se.exprs
phenoData
  rowNames: binary.cel, text.cel
  varLabels and varMetadata:
    sample: arbitrary numbering
featureData
  featureNames: 1007_s_at, 1053_at, ..., AFFX-TrpnX-M_at (22283 total)
  varLabels and varMetadata: none
experimentData: use 'experimentData(object)'
Annotation [1] "hgu133a"
```

Exercise 6
Packages often provide alternative methods for analysis, including analysis of two-color experiments. Use the vsn *package to further process an existing* ExpressionSet *named* lymphoma *and found in the* vsn *package.*

lymphoma is derived from a two-color experiment, with even-numbered samples representing the 'Red' channel, and odd numbered samples the corresponding 'Green' channel.

```
> library(vsn)
> data(lymphoma)
> vsnData <- justvsn(lymphoma)
```

Use the R help system to determine what justvsn (hint: vsn is an acronym for *variance stabilized normalization*) does. What does the help system tell you about the lymphoma data?

1.3 Visualization

R has a lot of visualization techniques, both for traditional graphs (e.g., the lattice package) and especially for visualization of high-throughput data.

So-called M-vs-A plots are a useful way to visually assess the effectiveness of normalization methods like that in the package vsn. M is the difference between the sample and reference expression level; A the average of the sample and reference expression levels.

Exercise 7
Remember that the lymphoma *data is arranged such that pairs of columns represent the 'Red' and 'Green' channels. Extract the underlying data from the* ExpressionSet*, for a particular pair of channels*

```
> exprVals <- exprs(vsnData)
> green <- exprVals[, 7]
> red <- exprVals[, 8]
```

exprs extracts a matrix representing the expression values, with 'features' as rows and samples as columns. The subsetting operations (e.g., exprVals[,7] extracts a numeric vector of values corresponding to a single column.

Exercise 8
Now calculate the M and A values:

```
> M <- green - red
> A <- (green + red)/2
```

Notice that these operations are *vector* operations – green and red are actually vectors of 9216 elements, and M is now a vector of pair-wise differences. Always a good idea to make sure things are happening as expected:

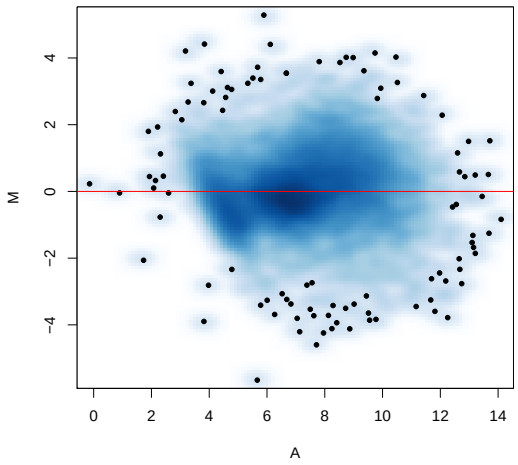

Figure 1: M versus A plot for lymphoma data samples 7 (red) and 8 (green).

```
> head(green, n = 5)

        1         2         3         4         5
 8.241099  8.647391  9.201032 10.136215 11.274535

> head(red, n = 5)

        1         2         3         4         5
 8.248713  9.305025 12.344540  9.036242 12.813635

> head(M, n = 5)

           1           2           3           4           5
-0.007614116 -0.657634966 -3.143507963  1.099973569 -1.539100668
```

Exercise 9
It is not informative to do a scatterplot of a large number of points. Instead, use the geneplotter package to generate a plot where density reflects local abundance.

```
> library(geneplotter)
> smoothScatter(A, M, pch = 20, xlab = "A", ylab = "M")
> abline(h = 0, col = "red")
```

The cryptic arguments to `smoothScatter` are directives about the appearance of the figure; a description can be found on the help page for `plot.default`. `abline` adds a straight line to the plot. The output is shown in Figure 1. The idea is that the normalization has made it so that the variance is independent of the expression level. Is this figure consistent with the expectation? What about other red and green pairs? What about the data before `justvsn`?

1.4 Interrogation

Bioconductor tries to organize data so that information remains connected across different domains.

Exercise 10
Load the `sample.ExpressionSet` *data from the* Biobase *package, give it a shorter name for convenience, and look at the annotation information, and the feature name of the second element in this data object.*

```
> library(Biobase)
> data(sample.ExpressionSet)
> obj <- sample.ExpressionSet
```

This sample data is derived from an Affymetrix `hgu95av2` chip. Each feature has a name associated with it, with the name referring to information in the chip description.

```
> annotation(obj)
```

```
[1] "hgu95av2"
```

```
> feature <- featureNames(obj)[100]
> feature
```

```
[1] "31339_at"
```

Each chip has a collection of annotations, derived from publicly accessible sources and collated into an *annotation* package. Some of the annotation information relies on additional sources of data, such as that collated into the GO (gene ontology) package.

Exercise 11
Load libraries required to access annotation information and construct the map from our `feature` *to its gene ontologies:*

```
> library(annotate)
> library(hgu95av2)
> library(GO)
> ontologies <- hgu95av2GO[[feature]]
> length(ontologies)
```

```
[1] 4
```

Exercise 12
Look at a selection of the ontologies identified, perhaps using the R *help system and* GO *documentation to understand the results.*

```
> ontologies[[1]]
```

```
$GOID
[1] "GO:0008150"

$Evidence
[1] "ND"

$Ontology
[1] "BP"
```

The AnnBuilder package allows construction of custom annotations. A flexible alternative is to use a package such as biomaRt, which integrates with web-based resources.

Exercise 13
Load biomaRt and select a 'mart' for retrieving information about features in our expression set.

```
> library(biomaRt)
> ensembl <- useMart("ensembl", dataset = "hsapiens_gene_ensembl")
```

This is fun – biomaRt is retrieving the information from a remote site (the one that hosts the ensembl mart), and confirming that the data set we selected is available. Check out the biomaRt vignettes and help pages to see how to discover available marts and resources.

Exercise 14
Let's look at one of our features, and use `getGene` from biomaRt to retrieve information about it (unfortunately, the chip naming convention followed by the mart people is different from that used in Bioconductor, so the `array` argument is constructed by hand).

```
> feature <- featureNames(obj)[100]
> gene <- getGene(id = feature, type = "affy_hg_u95av2", mart = ensembl)
> names(gene)

 [1] "affy_hg_u95av2"        "hgnc_symbol"           "description"
 [4] "chromosome_name"       "band"                  "strand"
 [7] "start_position"        "end_position"          "ensembl_gene_id"
[10] "ensembl_transcript_id"

> gene$description

[1] "protease inhibitor 15 preproprotein [Source:RefSeq_peptide;Acc:NP_056970]"
```

The command `names(gene)` provides the names of the types of information retrieved by the query; `genes$description` accesses some of that information.

1.5 Linear models

Designed experiments can frequently be cast as linear models. The limma package provides an interface to R extensive linear modeling capability, with features exploiting the data structure of typical expression experiments.

Exercise 15
Load the limma package and generate simulated expression data (100 probes, 6 microarrays; construct so that the first two probes are differentially expressed.

```
> library(limma)
> sd <- 0.3 * sqrt(4/rchisq(100, df = 4))
> ourExprs <- matrix(rnorm(100 * 6, sd = sd), nrow = 100, ncol = 6)
> rownames(ourExprs) <- paste("Gene", 1:100)
> colnames(ourExprs) <- paste("Sample", 1:6)
> ourExprs[1:2, 4:6] <- ourExprs[1:2, 4:6] + 2
```

The `matrix` function creates a matrix from a single vector of data. The vector is `100 * 6` elements long, with the elements drawn randomly from a normal distribution with a particular standard deviation. The matrix has 100 rows (corresponding to the number of genes) and 6 columns. We provide the rows and columns with names, for easy identification later on; these would be the feature- or sample-names of real data. Our 'over-expressed' genes are in rows 1 and 2, sample 4 through 6, and have 2 units added to their expression values. Use `head(y)` to make sure the simulate data looks reasonable.

Exercise 16
Linear models usually contrast the effects of different treatments. Create a `data.frame` *to contain phenotypic information about our samples, and combine the expression and phenotypic data into an* ExpressionSet, *so that we can deal with the information in a coordinated way.*

```
> ourPData <- data.frame(Group = c(rep("A", 3), rep("B", 3)))
> rownames(ourPData) <- colnames(ourExprs)
> exSet <- new("ExpressionSet", phenoData = new("AnnotatedDataFrame",
+     data = ourPData), exprs = ourExprs)
```

More typically, we would arrive at this point through reading in CEL or other data files, and performing preprocessing and other tasks.

Exercise 17
Linear models are described by a model matrix summarizing the model under investigation. Create the model matrix for our experiment:

```
> modelMatrix <- model.matrix(~exSet$Group, data = pData(exSet))
> modelMatrix
```

```
         (Intercept) exSet$GroupB
Sample 1           1            0
Sample 2           1            0
Sample 3           1            0
Sample 4           1            1
Sample 5           1            1
Sample 6           1            1
attr(,"assign")
[1] 0 1
attr(,"contrasts")
attr(,"contrasts")$`exSet$Group`
[1] "contr.treatment"
```

This is moderately cryptic, but the first argument to `model.matrix` is an R `formula`; the left-hand-side describes the response variables ('all', in this case), the right hand side the independent variables (the `Group` part of `exSet`, which we just created). A model matrix is defined for a particular data frame, and we can access the phenotypic data describing our experiment with the `pData` function (`pData` retrieves `ourPData`, which we just stored in `exSet`). The structure of the model matrix will be familiar to the statisticians in the crowd; we are going to compare expression levels between the two groups.

Exercise 18
Use the limma package provides a convenient way to fit our linear model to all *features in our expression set:*

```
> fit <- lmFit(exSet, modelMatrix)
> names(fit)
```

```
 [1] "coefficients"     "rank"             "assign"           "qr"
 [5] "df.residual"      "sigma"            "cov.coefficients" "stdev.unscaled"
 [9] "pivot"            "genes"            "Amean"            "method"
[13] "design"
```

The function `names(fit)` summarizes the information in the fitted model.

Features are not independent of one another, so it is likely that in assessing individual probes we can 'borrow' statistical information from the expression patterns of other probes. The author of the limma package has a particular implementation of this idea, and we can update our fit with his empirical Bayes approach:

```
> moderated <- eBayes(fit)
```

Exercise 19
Did our model detect differences between our groups?

Let's look at the most differentially expressed genes:

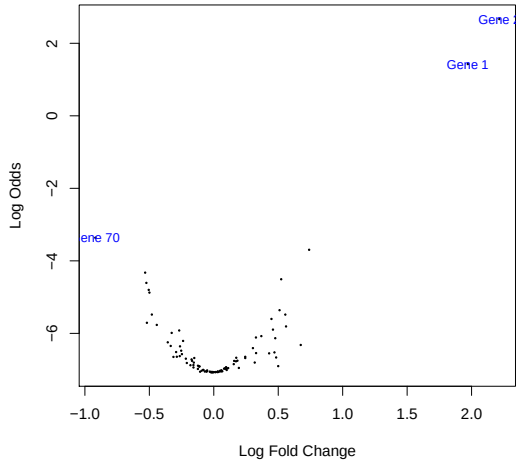

Figure 2: Volcano plot for simulated data contrasting two experimental groups

```
> topTable(moderated)[1:5, -2]
```

```
        ID exSet.GroupB     AveExpr          F      P.Value    adj.P.Val
2   Gene 2    2.2152943  1.55033973 106.846140 3.236566e-06 0.0003236566
1   Gene 1    1.9711057  0.95215694  49.124517 5.078202e-05 0.0025391009
90 Gene 90    0.5230701  0.54855070  16.982238 1.671551e-03 0.0557183761
68 Gene 68   -0.5048777 -0.38991418   9.072433 1.005482e-02 0.2513704090
70 Gene 70   -0.9189472  0.04271758   5.367992 3.594721e-02 0.7189442585
```

(the number of columns displayed is truncated for layout purposes). The P-values, adjusted for multiple comparisons, suggests that we have indeed been successful (consult the `topTable` manual page for a more complete description of this table).

A picture is worth a thousand words; we can see the significant differential expression in a so-called volcano plot:

```
> volcanoplot(moderated, coef = 2, highlight = 3)
```

That our genes are in the top right corner of Figure 2 indicates that they are highly unusual, and that they are over-expressed in the 'B' group.

limma allows for analysis of many, though not all, experimental designs. R's machinery for linear models can be employed to fit whatever linear or generalized linear model may be appropriate.

1.6 Etcetera

There are over 200 Bioconductor packages, and we have really just scratched the surface of a small handful. Visit the 'Browse packages' link of Bioconductor web site for some attempts to categorize the diversity of offerings.

2 A bus-man's holiday

2.1 From scratch

The most efficient way of installing new packages 'on the fly' is to use the `biocLite` function:

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

The first line illustrates the `source` function, which reads a file containing R commands. Surprisingly, `source` knows how to read files that are accessible via http, in addition to files available locally. The file `biocLite.R` contains the definition of `biocLite`; `source` has read that definition into the current R session.

The `biocLite` function invoked without any arguments causes R to install a default list of *packages* by consulting several on-line *repositories*. A package is a set of R instructions and documentation that provide specific functionality; it is the basic building-block for organizing R software into coherent and modular pieces that can be assembled by the user to perform specific tasks.

The `biocLite` function attempts to find each package and all of the *dependencies* (defined below). These are downloaded to the local computer and installed in the R system. Options to `biocLite` control which repositories are searched, whether package dependencies are also downloaded, and how the download and installation proceed.

The `biocLite` function can be invoked with a character vector of package names. Each package and its dependencies would then be located and installed. For instance, the command

```
> biocLite(c("Biobase", "biglm"))
```

downloads and installs the Biobase package (part of the Bioconductor project) and the biglm package (an R package for linear models applied to large data sets; not part of Bioconductor).

R packages are versioned. To compare the version of currently installed packages with the most recent version available online, and to update installed packages as needed, evaluate:

```
> library(Biobase)
> update.packages(repos = biocReposList())
```

This uses the R function `update.packages`, with a list of repositories specific to Bioconductor provided by the function `biocReposList` which is defined in the Biobase package.

Exercise 20
Are any packages in your installation out-of-date? Use the `update.packages` *to determine this. It is* not *necessary to update any out-of-date packages; see the help page for* `update.packages` *for information on how to check status without performing the update.*

2.2 A package and its parts

Start a new R session, and load the Biobase package

```
> library(Biobase)
```

On your system, this command generates a line `Loading required package: tools`. The tools package is an R package that Biobase depends on. By this we mean first that tools contains R code that Biobase uses and second that tools is not loaded, and hence its functionality is not available, by default. The authors of Biobase had to instruct R to load tools before loading Biobase.

Biobase is important to Bioconductor, and displays a 'splash screen' with hopefully useful information for new users (most packages do not display a splash screen). The splash screen points the user to important sources of information about the package and how it can be used.

2.2.1 Vignettes and help

The function `openVignette` produces a list of *vignettes* found in loaded packages. A vignette tries to convey a sense of package functionality. The best vignettes outline the problem that the package addresses, and walks the user through key steps in the use of the package. Vignettes are usually available to the user as PDF documents, making them easy to view and reference.

Package developers write vignettes in LaTeX, with additional facilities (provided by the Sweave package) for including R code that is evaluated when the package is installed. This *literate programming* technique provides a very useful illustration of how the code is meant to be used. Since the commands in a vignette are evaluate when the package is installed, vignettes provide a crude way of ensuring that the package is functioning: if the vignette does not build correctly, then something in the package code has gone amiss. The document you are currently reading is written using LaTeX and Sweave.

The splash screen also points to the `help` function. The equivalent commands `help(Biobase)` and `?Biobase` open a help page the package authors have written to summarize Biobase. `help(package=Biobase)` provides an overview of Biobase data classes, methods, and functions. Commands such as `?ExpressionSet` take the user to help pages describing particular objects, in this case the *ExpressionSet* class used in many Bioconductor packages to represent microarray expression data.

Package authors write help pages in a LaTeX-like syntax. Checks during package creation ensure that all objects visible to the package user are documented to a specific level of detail, i.e., package authors must eventually document their software! Tools in R create templates for manual pages, facilitating the documentation process.

Exercise 21
What is an ExpressionSet*? What can you do with an* ExpressionSet *once you have one? Briefly explore the vignettes and help pages for Biobase. How could the vignettes be made more useful to the new user?*

Vignettes and help pages provide enough information and guidance for the user to write R scripts that perform specific tasks. More advanced users can access and explore packages in additional ways, as we now discover.

2.2.2 Classes and methods

Many Bioconductor packages are based on the S4 object system. The S4 system is a way to formally represent complicated data objects and the methods that operate on them. Here, let's take a look at some basic features.

The S4 system allows package authors to create *classes* that encapsulate complicated data structures. *ExpressionSet* is an example of a class, and the help page provides some indication of how it is structured.

Exercise 22
To get a feeling for classes, take a look at the class definition of ExpressionSet*:*

```
> getClass("ExpressionSet")
```

ExpressionSet consists of *slots*. Each slot (slots are named `assayData`, `phenoData`, `featureData`, `experimentData`, `annotation`, and `.__classVersion__`) represents a different portion of the complex data an *ExpressionSet* is trying to represent. For instance, the `assayData` slot contains expression values resulting from several different chips in a single microarray experiment. The `phenoData` and `featureData` slots contain information about the samples used in the experiment, and the features present on the microarray.

Each slot contains an object of a particular class. The class of some slots (e.g., `annotation`, which has class *character*), are basic R data types. Other slots (e.g., `assayData`, with class *AssayData*) are themselves complex classes with their own class definition. The class of most *ExpressionSet* slots are defined in the Biobase package, but this is not necessary or even desirable: a key benefit to object systems like S4 is the reuse of complex objects that others have created. The oddly named slot `.__classVersion__` probably reflects an *ad hoc* convention on the part of the author to designate this slot as somehow special, perhaps suggesting that even advanced users do not normally need to worry about its content.

Exercise 23
The ExpressionSet *class* extends eSet *and other classes. The meaning of 'extends' is explored later, but use* `getClass` *to look at the structure of an* eSet. *Can you speculate on what 'extends' might represent, in terms of shared slots or other functionality? How might this notion contribute to object reuse, and to efficient programming?*

While classes encapsulate data structure, *methods* describe operations available on classes.

Exercise 24
Programmatically view all (well, most, there are bugs) of the methods available on an ExpressionSet.

To do this, evaluate the command

```
> showMethods(classes = "ExpressionSet")
```

One of the methods available for *ExpressionSet* is *exprs*. To find out what class of objects *exprs* works on, evaluate the command

```
> showMethods("exprs")
```

exprs is a fairly specialized method. What classes have *initialize* (used when new objects are created) and *show* (used to display objects) methods defined? What about `[` (the square bracket is, in fact, a function, see `?[`). Load the affy package; does this change the *initialize* and *show* methods available?

2.2.3 Name spaces

When the user types a symbol name (corresponding to a variable or function, for instance) at the prompt, R searches for the corresponding symbol definition. R maintains a *search path* of places to look for the symbol. R starts at the front of the search path (the *.GlobalEnv*, recording symbols that the user has entered at the R prompt) and proceeds down the search path until an appropriate symbol is found. Loading an R packages adds the package and its dependencies to the search path, immediately after .GlobalEnv.

Exercise 25
Use `search` *to view the search path of your current R session. Where are the Biobase and tools packages in the search path? What other packages are attached? When where those packages loaded?*

As packages become more complicated, it is usually the case that package authors write functions that are very useful internally, but whose visibility to the user would serve primarily to obscure the package functionality. It is also often advantageous to maintain a separation between how the user interacts with the package (the package interface) and the actual way in which the package implements calculations or represents data. This separation makes it easy

to update packages to more efficient data representations or analytic routines, without requiring the user to learn new ways of interacting with the package. A package *name space* provides a way to selectively expose package functionality.

Exercise 26
Use `getNamespaceExports` *to retrieve the symbols exported in the* Biobase *name space, and* `length` *to determine the number of exported symbols.*

```
> nmSpace <- getNamespaceExports("Biobase")
```

Symbols defined in a package but not visible in the name space are still accessible to the user.

Exercise 27
The symbol `assayDataStorageMode` *is defined in the* Biobase *name space, but not visible to the user. Use the help page* `?:::` *to determine how to access the definition of* `assayDataStorageMode`*. Why might the* Biobase *author have chosen not to expose this function? The help page you consulted also contains information on* `::`*, which provides access to publicly visible symbols. Why might you want to access publicly visible symbols in this way?*

2.3 From the outside...

To conclude this portion of the lab, quit R and navigate to a directory containing a package (using the linux command `cd` or the Windows command `chdir`). Take a quick look at the package structure (using `ls -l` or `dir`):

```
-rw-r--r-- 1 mtmorgan compbio  321 2007-03-14 13:26 DESCRIPTION
drwxr-xr-x 3 mtmorgan compbio 4096 2007-03-14 13:26 man
-rw-r--r-- 1 mtmorgan compbio  119 2007-03-14 13:26 NAMESPACE
drwxr-xr-x 3 mtmorgan compbio 4096 2007-03-14 13:26 R
drwxr-xr-x 3 mtmorgan compbio 4096 2007-03-14 13:26 src
```

The contents of these files and directories map to the topics covered today as follows:

File or directory	contents
DESCRIPTION	Dependencies and package description
NAMESPACE	Exposed methods, functions, and classes
R	R source code
man	Help files
inst/doc	Vignettes
src	C- or other source code

3 Summary

These tours provide a very brief overview of the packages and facilities available in R and Bioconductor. One final feature is the idea of *literate programming*: the document you have been reading was written using the Sweave package. All of the commands and figures were generated by running the written document through R and then LaTeX. This provides a *rich* and *reproducible* mechanism for documenting work flows and research approaches. The specific packages used for this document are

```
> toLatex(sessionInfo())
```

- R version 2.5.0 Patched (2007-04-24 r41305), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_US;LC_NUMERIC=C;LC_TIME=en_US;LC_COLLATE=en_US;LC_MONETARY=en_US;LC_`
- Base packages: base, datasets, graphics, grDevices, methods, stats, tools, utils
- Other packages: affy 1.15.0, affydata 1.11.2, affyio 1.5.0, annotate 1.15.0, Biobase 1.15.8, biomaRt 1.11.3, codetools 0.1-1, digest 0.3.0, geneplotter 1.15.1, GO 1.17.0, hgu133acdf 1.17.0, hgu95av2 1.17.0, lattice 0.15-4, limma 2.11.0, RCurl 0.8-0, weaver 1.3.0, XML 1.7-3