Analysing time course microarray data using Bioconductor: a case study using yeast2 Affymetrix arrays

Colin S. Gillespie[1,2,†], Guiyuan Lei[1,2,†], Richard J. Boys[1,2], Amanda Greenall[2,3] and Darren J. Wilkinson[1,2]

November 14, 2012

[1]School of Mathematics & Statistics, Newcastle University, Newcastle upon Tyne, NE1 7RU, UK.
[2]Centre for Integrated Systems Biology of Ageing and Nutrition (CISBAN), Newcastle University, UK.
[3]Institute for Ageing and Health, Newcastle University, Campus for Ageing and Vitality, Newcastle upon Tyne, NE4 5PL, UK.
[†]Both authors contributed equally to the work.

Background: Large scale microarray experiments are becoming increasingly routine, particularly those which track a number of different cell lines through time. This time-course information provides valuable insight into the dynamic mechanisms underlying the biological processes being observed. However, proper statistical analysis of time-course data requires the use of more sophisticated tools and complex statistical models.

Findings: Using the open source CRAN and Bioconductor repositories for R, we provide example analysis and protocol which illustrate a variety of methods that can be used to analyse time-course microarray data. In particular, we highlight how to construct appropriate contrasts to detect differentially expressed genes and how to generate plausible pathways from the data. A maintained version of the R commands can be found at http://www.mas.ncl.ac.uk/~ncsg3/microarray/

Conclusions: CRAN and Bioconductor are stable repositories that provide a wide variety of appropriate statistical tools to analyse time course microarray data.

1. Introduction

As experimental costs decrease, large scale microarray experiments are becoming increasingly routine, particularly those which track a number of different cell lines through time. This is because time-course information provides valuable insight into the dynamic mechanisms underlying the biological processes being observed. However, a proper statistical analysis of time-course data requires the use of more sophisticated tools and complex statistical models. For example, problems due to multiple comparisons are increased by catering for changing effects over time. In this case study, we demonstrate how to analyse time-course microarray data by investigating a data set on yeast. We discuss issues related to normalisation, extraction of probesets for specific species, chip quality,differential expression and network inference. The freely available software system R (see [11,22]) has many benefits for analysing data of this type and so throughout the analysis we give the R commands that produce the numerical/graphical output shown in this paper. A maintained version of the R commands can be found at http://www.mas.ncl.ac.uk/~ncsg3/microarray/

1.1. Description of the data

The data were collected according to the experimental protocol described in [12]. Briefly, three biological replicates were studied on each of a wild-type (WT) yeast strain and a strain carrying the *cdc13-1* temperature sensitive mutation (in which telomere uncapping is induced by growth at temperatures above around 27°C). These replicates were sampled initially at 23°C (at which *cdc13-1* has essentially WT telomeres) and then at 1, 2, 3 and 4 hours after a shift to 30°C to induce telomere uncapping. The thirty resulting RNA samples were hybridised to Affymetrix yeast2 arrays. The microarray data are available in the ArrayExpress database (see [21]) under accession number E-MEXP-1551 .

2. Loading microarray data into Bioconductor

2.1. Installing Bioconductor and associated packages

Assuming that R is already installed, Bioconductor is fairly straightforward to obtain installation script, viz:

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

This installs a number of base packages, including `affy`, `affyPLM`, `limma`, and `gcrma` (see [6,10,25]). Additional non-standard packages can also be easily installed. For example, the additional packages needed for this paper can be installed by using

```
##From Bioconductor
biocLite(c('ArrayExpress', 'Mfuzz', 'timecourse', 'yeast2.db',
    'yeast2probe', 'yeast2cdf'))
##From cran
install.packages(c('GeneNet', 'gplots'))
```

Bioconductor packages are updated regularly on the web and so users can easily update their currently installed packages by starting a new R session and then using

```
update.packages(repos = biocinstallRepos())
```

See [5] for further details on installation. A list of packages used in this paper is given in Appendix C.

2.2. Entering data into Bioconductor

The data used in this paper can be downloaded from ArrayExpress into R using the commands

```
library(ArrayExpress)
yeast.raw = ArrayExpress("E-MEXP-1551")
```

A brief description of the `yeast.raw` object can be obtained by using the `print(yeast.raw)` command:

```
AffyBatch object
size of arrays = 496x496 features (3163 kb)
cdf = Yeast_2 (10928 affyids)
number of samples = 30
number of genes = 10928
annotation = yeast2
```

If the Affymetrix microarray data sets have been downloaded into a single directory, then the `.cel` files can be loaded into R using the `ReadAffy` command.

Also available from ArrayExpress are the experimental conditions. However, some preprocessing is necessary:

```
ph = yeast.raw@phenoData
exp_fac = data.frame(data_order = 1:30,
    strain = ph@data$Factor.Value..GENOTYPE.,
    replicates = ph@data$Factor.Value..INDIVIDUAL.,
    tps = ph@data$Factor.Value..TIME.)
levels(exp_fac$strain) = c('m', 'w')
exp_fac = with(exp_fac, exp_fac[order(strain, replicates, tps), ])
exp_fac$replicate = rep(1:3, each=5, 2)
```

The data frame `exp_fac` stores all the necessary information, such as strain, time and replicate, which are necessary for the statistical analysis.

Note that there are two yeast species on this chip, *S. pombe* and *S. cerevisiae*. Also, amongst the 10,928 probesets (with each probeset having 11 probe pairs), there are 5,900 *S. cerevisiae* probesets.

3. Pre-processing

3.1. Extraction of S. cerevisiae probesets

As these microarrays contain probesets for both *S. cerevisiae* and *S. pombe*, we first need to extract the *S. cerevisiae* data before normalisation. This can be done by filtering out the *S. pombe* data using the `s_cerevisiae.msk` file from the Affymetrix website (see [2]). Note that users first need to register with the Affymetrix website before downloading this file. Also note that in our analysis, the transcript id i.e. the systematic orf name (obtained from [3]) is used for genes with no name.

We obtain a data frame containing lists of *S. cerevisiae* genes, probes and transcripts (using the function `ExtractIDs()` in Appendix A) as follows

```
# Read in the mask file
s_cer = read.table("s_cerevisiae.msk", skip = 2, stringsAsFactors = FALSE)
source("ExtractIDs.R")
c_df = ExtractIDs(s_cer[, 1])
```

We also need to restrict the view of `yeast.raw` to the $x-$ and $y-$coordinates of the *S. cerevisiae* probesets in the `cdf` environment by using

```
# Get the raw dataset for S. cerevisiae only
library(affy)
library(yeast2probe)
source("RemoveProbes.R")
cleancdf = cleancdfname(yeast.raw@cdfName)
RemoveProbes(s_cer[, 1], cleancdf, "yeast2probe")
```

Note that the commands in `RemoveProbes.R` are listed in Appendix A. Thus the attributes of `yeast.raw`, obtained via `print(yeast.raw)`, are now

```
AffyBatch object
size of arrays = 496x496 features (3167 kb)
cdf = Yeast_2 (5900 affyids)
number of samples = 30
number of genes = 5900
annotation = yeast2
```

and the number of genes (actually probesets here) is 5,900 now that the *S. pombe* probesets have been removed.

3.2. Data Quality Assessment

Before any formal statistical analysis, it is important to check for data quality. Initially, we might examine the perfect and mismatch probe-level data to detect anomalies. Images of the first five arrays can be obtained using

```
for (i in 1:5) {
    plot_title = paste("Strain: ", exp_fac$strain[i], "Time: ", exp_fac$tps[i])
    d = exp_fac$data_order[i]
    image(yeast.raw[, d], main = plot_title)
}
```

These commands produce the image shown in Appendix B: Figure 7. Data quality can be assessed by examining such images for anything that appears non-random such as rings, shadows, lines and strong variations in shade. The images for our data set do not appear to have any non-random structure and so data quality is probably high.

Another useful quality assessment tool is to examine density plots of the probe intensities. The command

```
d = exp_fac$data_order[1:5]
hist(yeast.raw[,d], lwd=2, ylab="Density", xlab="Log (base 2) intensities")
```

produces the image shown in Appendix B: Figure 8. Typically, differences in spread and position are corrected by normalisation. However, the appearance of significant multi-modality in the distribution or many outlying observations are indicative of poor data quality.

Other exploratory data analysis techniques that should be carried include MAplots, where two microarrays are compared and their log intensity difference for each probe on each gene are plotted against their average. Also of interest is to examine RNA degradation (see [6]), although [4] cast some doubt over the validity of this method. For details on how to carry out both of these methods in R, see [1, 13] for detailed instructions.

3.3. Normalising Microarray Data

There are number of methods for normalising microarray data. Two of the most popular methods are GeneChip RMA (GCRMA) and Robust Multiple-array Average (RMA); see [16,28]. Essentially, GCRMA and RMA differ in how they deal with background noise, with GCRMA using a more sophisticated correction algorithm. However, the approach adopted by GCRMA means that it can be time-consuming to use with large data sets in contrast to RMA. A potential drawback of using RMA is that it assumes that the overall levels of expression are similar for each array. However this assumption may be invalid if, for example, mutant cells have a radically different level of transcriptional activity than the WT. For further information regarding normalising microarray data sets, see for example [14, 17].

Since we have thirty microarray data sets and believe that the levels of transcriptional activity are similar across strains, we will use the RMA normalisation method. This technique normalises across the set of hybridizations at the probe level. The data can be normalised via

```
yeast.rma = rma(yeast.raw)

## Background correcting
## Normalizing
## Calculating Expression

yeast.matrix = exprs(yeast.rma)[, exp_fac$data_order]
colnames(yeast.matrix) = paste(exp_fac$strain, exp_fac$tps, sep = "")
exp_fac$data_order = 1:30
```

The normalisation procedure consists of three steps: model-based background correction, quantile nomralisation and robust averaging. The aim of the quantile normalisation is to make the distribution of probe intensities for each array in a set of arrays the same. We illustrate its effect by studying boxplots of the raw *S. cerevisiae* data against their normalised counterparts values, shown in the Appendix B: Figure 9. Boxplots provide a useful graphical view of data distributions and contain their median, quartiles, maximum and minimum values. The `boxplot` command is in the `affyPLM` package and so the figure is produced by using

```
library(affyPLM)
# Raw data intensities
boxplot(yeast.raw, col = "red", main = "", ylim = c(2, 16))
# Normalised intensities
boxplot(yeast.rma, col = "blue", ylim = c(2, 16))
```

3.4. Principal Component Analysis

Principal component analysis (PCA) is useful in exploratory data analysis as it can reduce the number of variables to consider whilst still retaining much of the variability in the data. In particular, PCA is useful for identifying patterns in the data. Essentially, principal components partition the data into orthogonal linear components which explain different contributions to the variability in the data. The first component explains the largest contribution to variability in the original dataset, that is, retains most information, with the second component explaining the next largest contribution to variability, and so on. The following commands calculate the principal components

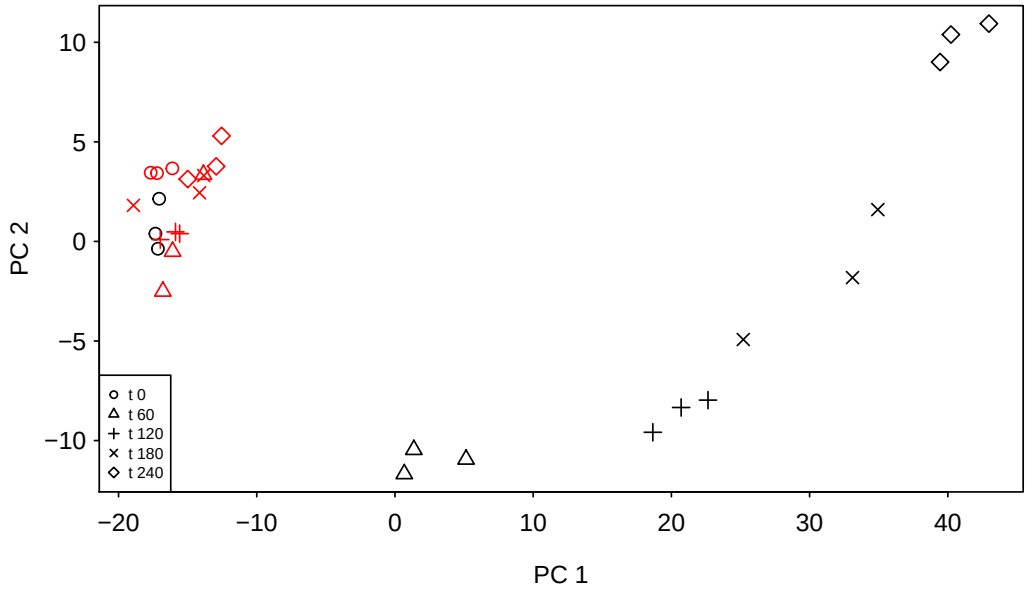

Figure 1: A plot of the first two principal components. The red symbols correspond to the wild-type strain.

```
yeast.PC = prcomp(t(yeast.matrix))
yeast.scores = predict(yeast.PC)
```

which we can then plot using

```
#Plot of the first two principal components
plot(yeast.scores[,1], yeast.scores[,2], xlab='PC 1', ylab='PC 2',
    pch=rep(1:5, 6), col=as.numeric(exp_fac$strain))
legend("bottomleft", pch=1:5, cex=0.6,
    c('t 0', 't 60', 't 120', 't 180', 't 240'))
```

Figure 1 highlights a clear (and expected) time effect in the mutant yeast which is not present in the wild-type strain. In particular, mutant samples are clustered by their time points; for example, the three mutant replicates at time point 4 are clustered at the bottom right of the figure.

4. Identifying differentially expressed genes

In this experiment, interest lies in differences in gene expression over time between the wild-type and mutant yeast strains. It is expected that the wild-type expression level is independent of time. Also we anticipate that the mutant expressions at time $t = 0$ are the same as the wild-type expression level. This hypothesis is supported by the PCA plot in Figure 1.

There are currently two main packages available to detect differentially expressed genes using this kind of data: the `timecourse` package and the `limma` package. We illustrate how to analyse these data using both packages.

4.1. Using the timecourse package

This package assesses treatment differences by comparing time-course mean profiles allowing for variability both within and between time points. It uses the multivariate empirical Bayes model proposed by [27]. Further details of the `timecourse` package can be found in [26]. After installing the `timecourse` package, we construct a `size` matrix describing the replication structure using

```
library(timecourse)
size = matrix(3, nrow = 5900, ncol = 2)
```

To extract a list of differentially expressed we calculate the Hotelling statistic $\tilde{T}^2$ via

```
c.grp = as.character(exp_fac$strain)
t.grp = as.numeric(exp_fac$tps)
r.grp = as.character(exp_fac$replicate)
MB.2D = mb.long(yeast.matrix, times = 5, method = '2',
    reps = size, condition.grp = c.grp, time.grp = t.grp,
    rep.grp = r.grp)
```

The top (say) one hundred genes can be extracted via

```
gene_positions = MB.2D$pos.HotellingT2[1:100]
gnames = rownames(yeast.matrix)
gene_probes = gnames[gene_positions]
```

The expression profiles can also be easily obtained. The profile for the top ranked expression is found using

```
plotProfile(MB.2D, ranking = 1, gnames = rownames(yeast.matrix))
```

and is shown in the Appendix B: Figure 10.

4.2. Using the limma package

The `limma` package uses the moderated t-statistic described by [24,25]. The function `lmFit` within the `limma` library fits a linear model for each gene for a given series of arrays, where the coefficients of the fitted models describe the differences between the RNA sources hybridised to the arrays. Precisely, we fit the model $E[y_g] = X\alpha_g$, where $y_g = (y_{g,1}, \ldots, y_{g,n})^T$ contains the expression values for gene g across the n arrays, X is a design matrix which describes key features of the experimental design used and α_g is the coefficient vector for gene g. In the analysis studied here, the yeast data consists of data from $n = 30$ arrays. The entries in the columns of X depend on the experimental design used: there are two yeast strains (mutant and wild type), each measured at five separate time points, and we are interested in comparing the gene expressions between mutant and wild type strains over time. Thus we seek a linear model describing the ten strain×time combinations by determining values for the ten coefficients in the coefficient vector α_g. We will label these ten coefficients as ('m0', 'm60, 'm120', 'm180', 'm240', 'w0', 'w60', 'w120', 'w180', 'w240'), where the first five coefficients represent the levels of the mutant strain at time points $t = 0, 1, 2, 3, 4$ and the remaining five coefficients are the equivalent versions for the wild type strain. Statistically speaking, the model has a single factor with ten levels. The design matrix X

links these factors to the data in the arrays by having zero entries except when an array contributes an observation to a particular strain×time combination. For example, array 26 measures the expression of the first wild type microarray at time $t = 0$ and so contributes an observation to level 'w0', the sixth strain×time combination. Thus the entry in row 26, column 6 of the design matrix $X(26, 6) = 1$. Further, the arrays are arranged in groups of three replicates. Thus the overall experimental structure (`expt_structure` below) has three arrays on level 'm0', then three arrays on 'm60', and so on. Setting up the factor levels and the design matrix is done in R by using

```
library(limma)
expt_structure = factor(colnames(yeast.matrix))

#Construct the design matrix
X = model.matrix(~0 + expt_structure)
colnames(X) =  c('m0', 'm60', 'm120', 'm180', 'm240',
    'w0', 'w60', 'w120', 'w180', 'w240')
```

and then the coefficient vector α_g is estimated via the command

```
lm.fit = lmFit(yeast.matrix, X)
```

Determining the differentially expressed genes amounts to studying contrasts of the various strain×time levels, as described by a contrast matrix C. For these data, we are mainly interested in differences at the later time points, and so a possible set of contrasts to investigate is that of differences between the mutant and wild type strains at each time point, that is, `('m60-w60', 'm120-w120', 'm180-w180', 'm240-w240')`. The `limma` package allows complete flexibility over the choice of contrasts, however this necessarily includes an additional level of complexity. The values in the coefficient vector of contrasts, $\beta_g = C^T \alpha_g$ for gene g, describe the size of the difference between strains at each time point. The relevant R commands are

```
mc = makeContrasts("m60-w60", "m120-w120", "m180-w180", "m240-w240", levels = X)
c.fit = contrasts.fit(lm.fit, mc)
eb = eBayes(c.fit)
```

The final command uses the `eBayes` function to produce moderated t-statistics which assess whether individual contrast values β_{gj} are plausibly zero, corresponding to no significant evidence of a difference between strains at time point j. The moderated t-statistic is constructed using a shrinkage approach and so is not as sensitive as the standard t-statistic to small sample sizes. It also gives a moderated F-statistic which can be used to test whether all contrasts are zero simultaneously, that is, whether there is no difference between strains at all time points.

4.3. Ranking differentially expressed genes

There are a number of ways to rank the differentially expressed genes. For example, they can be ranked according to their log-fold change

```
# see help(toptable) for more options
toptable(eb, sort.by = "logFC")
```

or by using F-statistics

```
topTableF(eb)
```

The advantage of using F-statistics over the log fold change is that the F-statistic takes into account variability and reproducibility, in addition to fold-change.

Our analysis is based on a large number of statistical tests, and so we must correct for this multiple testing. In our example we use the (very) conservative Bonferroni correction since we have a large number of differentially expressed genes and the resulting corrected list is still long. Another common method of correcting for multiple testing is to use the false discovery rate (fdr) (use the command `?p.adjust` to obtain further details). The following commands rank genes according to their (corrected) F-statistic p-value and annotates the output by indicating the direction of the change for each contrast for each gene: $+1$ for up-regulated expression (mutant type having higher expression than wild type at a particular time point), -1 for down-regulated expression and 0 for no significant change.

```
modFpvalue = eb$F.p.value
##Change 'bonferroni' to 'fdr' to use the
##false discovery rate as a cut-off
indx = p.adjust(modFpvalue, method='bonferroni') < 0.05
sig = modFpvalue[indx]

#No. of sig. differential expressed genes
nsiggenes = length(sig)
results = decideTests(eb, method='nestedF')

modF = eb$F
modFordered = order(modF, decreasing = TRUE)

#Retrieve the significant probes and genes
c_rank_probe = c_df$probe[modFordered[1:nsiggenes]]
c_rank_genename = c_df$genename[modFordered[1:nsiggenes]]

#Create a list and write to a file
updown = results[modFordered[1:nsiggenes],]
write.table(cbind(c_rank_probe, c_rank_genename, updown),
    file='updown.csv', sep=',',
    row.names=FALSE, col.names=FALSE)
```

The following code (adapted from lecture material found at [1]) plots the time course expression for the top one hundred differentially expressed genes according to their F-statistic (see Figure 2)

```
#Rank of Probesets, also output gene names
par(mfrow=c(3, 3), ask=TRUE)
for(i in 0:99){
    indx = rank(modF) == nrow(yeast.matrix) - i
    id = c_df$probe[indx]
    name = c_df$genename[indx]
    exprs.row = yeast.matrix[indx,]
    genetitle = paste(sprintf('%.30s', id), sprintf('%.30s', name),
```

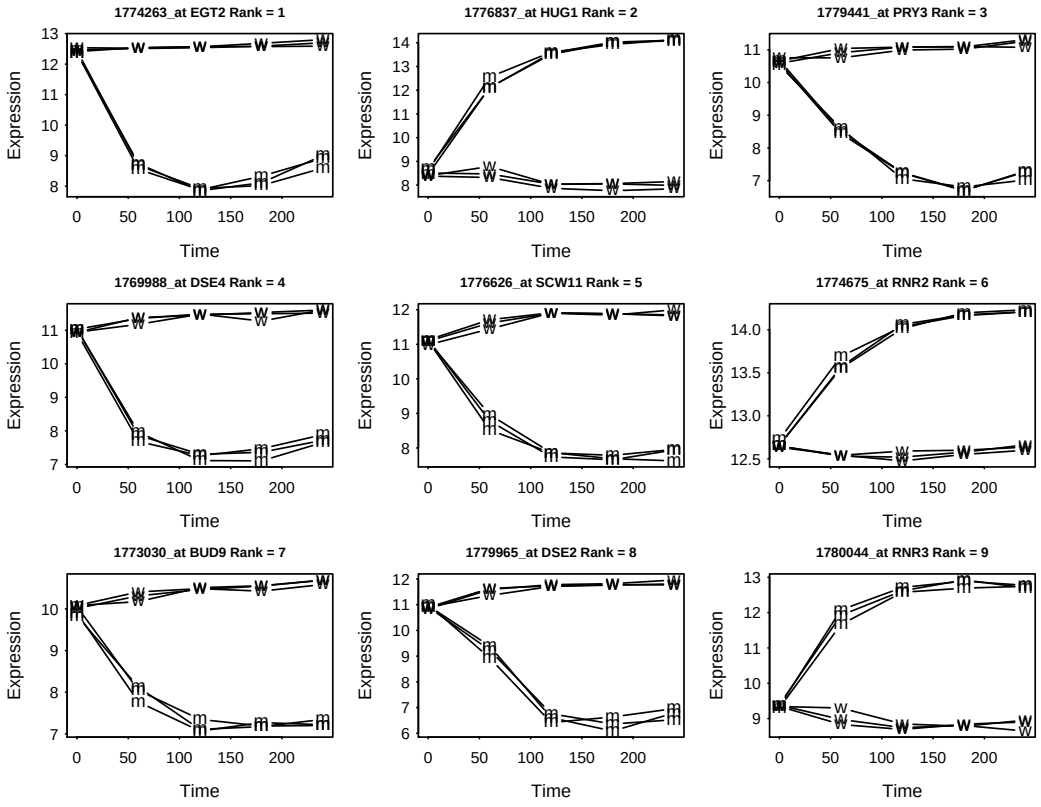

Figure 2: Time course expression levels for the top 9 differentially expressed genes, ranked by their F-statistic. The triangles and circles correspond to the wild-type and mutant genes respectively.

```
                    'Rank =', i+1)

    plot(0, pch=NA, xlim=range(0, 240), ylim=range(exprs.row),
         ylab='Expression', xlab='Time', main=genetitle)

    for(j in 1:6){
        pch_value = as.character(exp_fac$strain[5*j])
        points(c(0, 60, 120, 180, 240), exprs.row[(5*j-4):(5*j)],
               type='b', pch=pch_value)
    }
}
```

When interpreting rank orderings based on statistical significance, it is important to bear in mind that a statistically significant differential expression is not always biologically meaningful. For example, Figure 2 contains *RNR2*. This gene is highly significant because of low variation in its time course. However the actual difference in expression levels between wild-type and mutant stains is relatively small. We address this problem in the next section.

Comparison of the `timecourse` and `limma` packages

Both packages have different strengths. One advantage of the `timecourse` package over the `limma` package is that it allows for correlation between repeated measurements on the same experimental

unit, thereby reducing false positives and false negatives; these false positives/negatives are a significant problem when the variance-covariance matrix is poorly estimated. An advantage of the `limma` package is that it allows more flexibility by allowing users to construct different contrasts. In general we might expect both packages to produce fairly similar lists of say the top 100 probesets. In the analysis of the yeast data, we can determine the overlap of the top 100 probesets by using

```
N = 100
gene_positions = MB.2D$pos.HotellingT2[1:N]
tc_top_probes = gnames[gene_positions]
lm_top_probes = c_df$probe[modFordered[1:N]]
length(intersect(tc_top_probes, lm_top_probes))

## [1] 53
```

The result is a moderately large overlap of fifty-three probesets. We note that changing the ranking method in the `limma` package also yields similar results as those given by the `timecourse` package.

4.4. Two fold-change list

When looking for "interesting" genes it can be helpful to restrict attention to those differential expressed that are both statistically significant and of biological interest. This objective can be achieved by considering only significant genes which show, say, at least a two-fold change in their expression level. This gene list is obtained using the following code (adapted from [13])

```
# Obtain the maximum fold change but keep the sign
maxfoldchange = function(foldchange) foldchange[which.max(abs(foldchange))]
difference = apply(eb$coeff, 1, maxfoldchange)
pvalue = eb$F.p.value
lodd = -log10(pvalue)

# hfc: high fold-change
nd = (abs(difference) > log(2, 2))
ordered_hfc = order(abs(difference), decreasing = TRUE)
hfc = ordered_hfc[1:length(difference[nd])]

np = p.adjust(pvalue, method = "bonferroni") < 0.05

# lpv: low p-value (large F-value)
ordered_lpv = order(abs(pvalue), decreasing = FALSE)
lpv = ordered_lpv[1:length(pvalue[np])]

oo = union(lpv, hfc)
ii = intersect(lpv, hfc)
```

Figure 3 contains a "volcano" plot which illustrates the effect of using different levels of fold change and significance thresholds. The figure is produced by using the following code

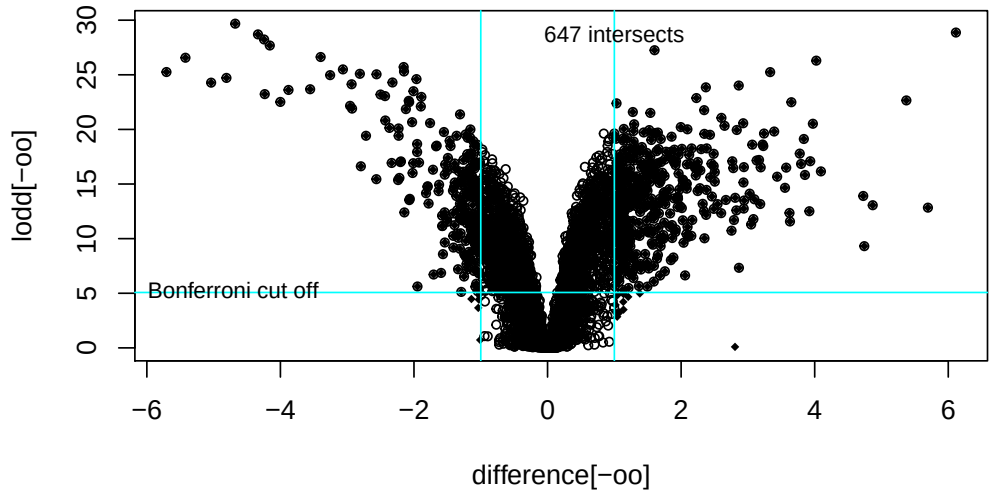

Figure 3: Volcano plot showing the bonferroni cut-off and the two-fold change.

```
#Construct a volcano plot using moderated F-statistics
plot(difference[-oo], lodd[-oo], xlim=range(difference),
ylim=range(lodd), cex=0.7)
points(difference[hfc], lodd[hfc], pch=18, cex=0.7)
points(difference[lpv], lodd[lpv], pch=1, cex=0.7)

#Add the cut-off lines
abline(v=log(2, 2), col=5); abline(v=-log(2, 2), col=5)
abline(h=-log10(0.05/5900), col=5)

text(min(difference) + 1, -log10(0.05/5900) + 0.2,
    'Bonferroni cut off', cex=0.8)
text(1, max(lodd) - 1, paste(length(ii), 'intersects'), cex=0.8)
```

5. Cluster Analysis

Biological insight can be gained by determining groups of differentially expressed genes, that is, groups of genes which increase or decrease simultaneously. This can be achieved by using cluster analysis.

5.1. Traditional cluster analysis

In this section, we separate the top fifty differentially expressed genes into groups of similar pattern (clusters). Clearly different genes will have different overall levels of expression and so we first standardise their measurements by taking the expression level of the mutant strain (at each time point) relative to the wild-type at time $t = 0$:

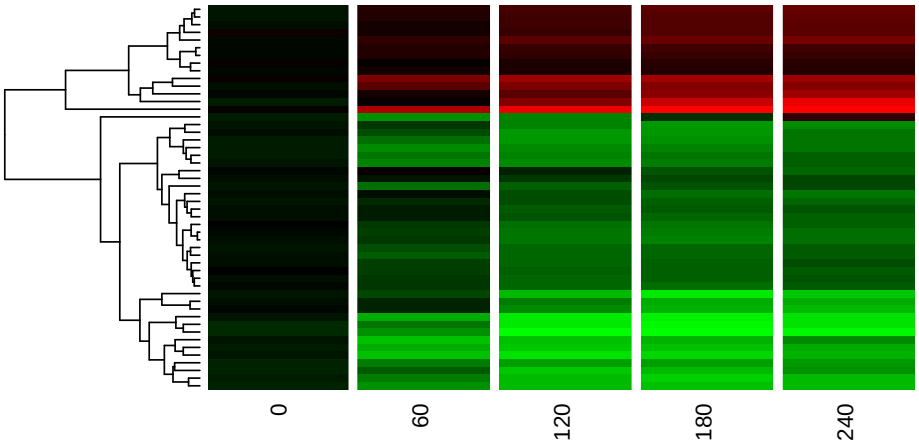

Figure 4: Clustering of the top fifty differentially expressed genes. Red and green correspond to up- and down-regulation respectively.

```
c_probe_data = yeast.matrix[ii, ]
# Average of WT
wt_means = apply(c_probe_data[, 16:30], 1, mean)
m = matrix(nrow = dim(c_probe_data)[1], ncol = 5)

for (i in 1:5) {
    mut_rep = c(i, i + 5, i + 10)
    m[, i] = rowMeans(c_probe_data[, mut_rep]) - wt_means
}
colnames(m) = sort(unique(exp_fac$tps))
```

The heatmap in Figure 4 is obtained by using the function `heatmap.2` from the library `gplots` via the following code

```
library(gplots)
#Cluster the top 50 genes
heatmap.2(m[1:50,], dendrogram ='row', Colv=FALSE, col=greenred(75),
          key=FALSE, keysize=1.0, symkey=FALSE, density.info='none',
          trace='none', colsep=rep(1:10), sepcolor='white',
          sepwidth=0.05, labRow = NA, cexCol=1,
          hclustfun=function(c){hclust(c, method='average')})
```

Figure 4 shows the relative expression levels for the mutant strain at each time point ('0', '60', '120', '180', '240'). As expected, the relative expression levels at time $t = 0$ are very similar. However, as time progresses, groupings of genes appear whose levels are up-regulated (red) or down-regulated (green). Note that the intensity of the colour corresponds to the magnitude of the relative expression. Gene names appear on the right side of the figure and on the left side, the

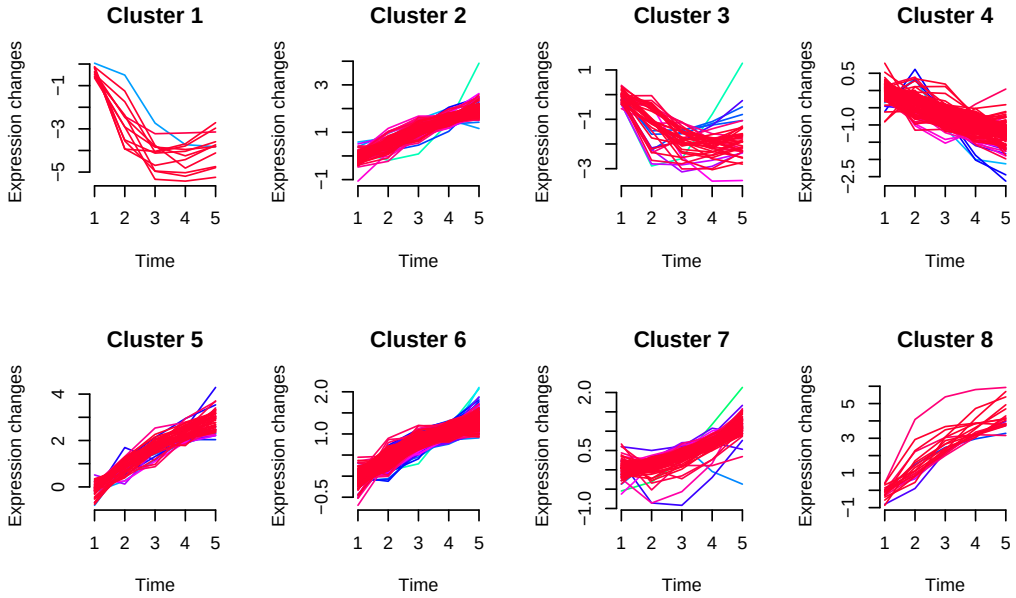

Figure 5: Eight clusters obtained from the extttMfuzz package.

cluster dendrogram shows which genes have similar expression. The dendrogram suggests that there are perhaps six to ten clusters.

5.2. Soft clustering

Soft clustering methods have the advantage that a probe can be assigned to more than one cluster. Furthermore, it is possible to grade cluster membership within particular groupings. Soft clustering is considered more robust when dealing with noisy data; for more details see [8, 9]. The `Mfuzz` package implements soft clustering using a fuzzy c-means algorithm. Analysing the data for $c = 8$ clusters is achieved by using

```
library(Mfuzz)
tmp_expr = new("ExpressionSet", exprs = m)
cl = mfuzz(tmp_expr, c = 8, m = 1.25)
mfuzz.plot(tmp_expr, cl = cl, mfrow = c(2, 4), new.window = FALSE)
```

Of course, it is usually not clear how many clusters there are (or should be) within a dataset and so the sensitivity of conclusions to the choice of number of clusters (c) should always be investigated. For example, if c is chosen to be too large then some clusters will appear sparse and this might suggest choosing a smaller value of c. Figure 5 shows the profiles of the eight clusters obtained from the `Mfuzz` package. The probes present within each cluster can be found by using

```
cluster = 1
cl[[4]][, cluster]
```

6. Genetic regulatory network inference

Recent research in the analysis of microarray experiments has produced several methods for determining plausible transcriptional regulation networks from the data; see, for example, [7, 15, 18, 29]. Such networks can provide valuable insight into the underlying biological mechanisms producing the data. The networks typically consist of nodes (representing genes or proteins) and edges between nodes (representing relationships between genes). Methods also exist for inferring dynamic gene association networks in which the direction of causation is represented by an arrow. One such method (with an easy-to-use R package developed by the Strimmer lab) uses a shrinkage approach to calculate the partial correlation coefficient and works for both static and dynamic (time-course) data; see [19, 20, 23]. Their heuristic algorithm is fast and so provides a quick insight in the structure of the network. It works by first creating a `longitudinal` R object. In our illustration, we use data on the top one hundred differentially expressed genes. The data are stored in a matrix `m`, where the rows are genes and the fifthteen columns are the arrays. In order to use the R functions in their package, as we have time course data rearrange the row order according to the time points. The first three rows of the resulting matrix `mnew` are data on the three mutant arrays at time $t = 0$, the next three arrays at time point $t = 60$, and so on. This is achieved by using the commands

```
exp_fac = with(exp_fac, exp_fac[order(strain, tps, replicates), ])
# Construct a longitudinal object
library(GeneNet)
ngenes = 100
m = yeast.matrix[ii[1:ngenes], ]
mnew = m[, exp_fac$data_order[1:15]]
mlong = as.longitudinal(t(mnew), repeats = 3, time = 0:4)
```

Next the partial correlations are computed and then (local) values are assigned to all possible edges, from which the important edges can be determined according to a threshold criteria. Finally a Graphviz[1] file is outputted to generate the graph. Figure 6 shows the resulting network for these data and is generated by using

```
# Compute partial correlations
pcor.dyn = ggm.estimate.pcor(mlong, method = "dynamic")
# Assign (local) fdr values to all possible edges
m.edges = network.test.edges(pcor.dyn, direct = TRUE)
# Construct graph containing top edges
m.net = extract.network(m.edges, method.ggm = "number", cutoff.ggm = 100)
# Construct a Graphviz dot file
rnames = vector("list", length(1))
rnames = c_df$genename[ii[1:ngenes]]
network.make.dot(filename = "net.dot", m.net, rnames, main = "Yeast Network")
```

Note that, in the function `ggm.estimate.pcor`, the default method `static` employs the function `pcor.shrink`, whereas the `dynamic` method uses `dyn.pcor`. The difference between the two estimators is that the latter takes the spacings between time points into account if the input consists of multiple time course data (which must be provided as a `longitudinal` object).

[1] See http://www.graphviz.org/ for details.

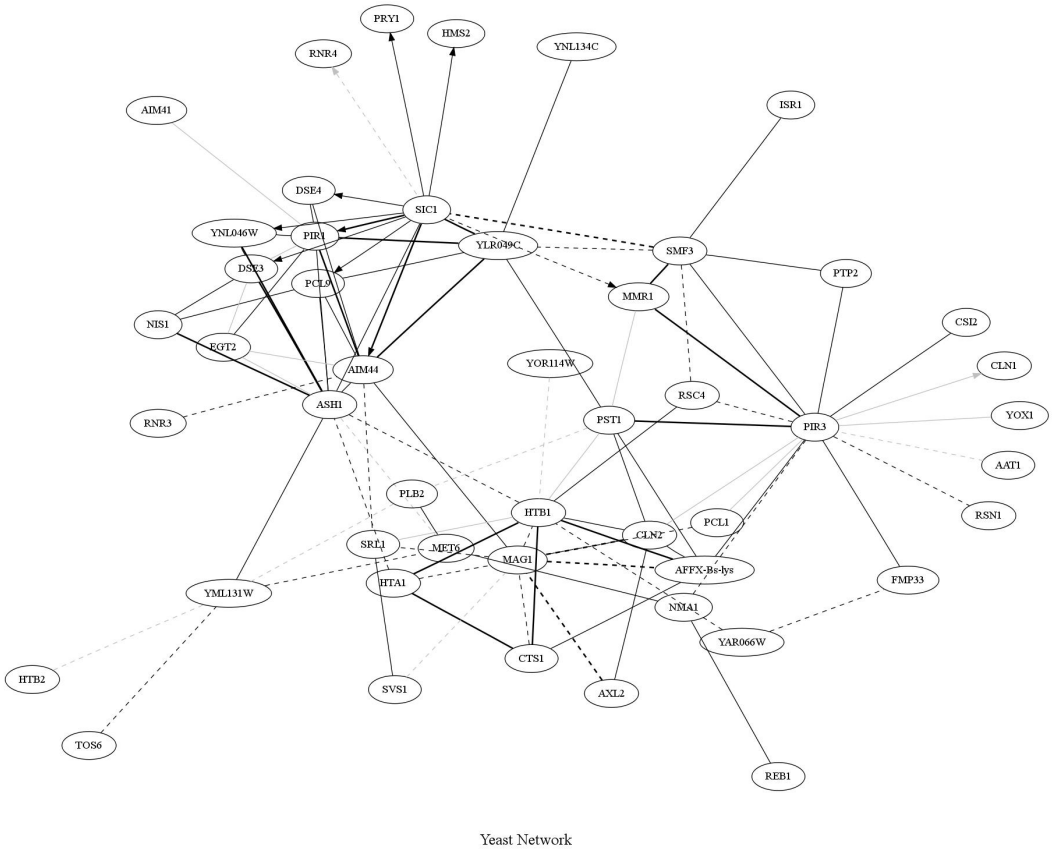

Figure 6: Gene network inferred from the yeast microarray data set. Black and grey indicate positive and negative (partial) correlation respectively.

7. Conclusion

The response to telomere uncapping in *cdc13-1* strains was expected to share features in common with responses to cell cycle progression, environmental stress, DNA damage and other types of telomere damage. The statistical analysis determined lists of probesets associated with genes involved in all of these processes. The techniques used focussed on making best use of the temporal information in time-course data. The use of *cdc13-1* strains, which uncap telomeres quickly and synchronously, also allowed the identification of genes involved in the acute response to telomere damage. This case study has demonstrated the power of R/Bioconductor to analyse time-course microarray data. Whilst the statistical analysis of such data is still an active research area, this paper presents some of the cutting-edge tools that are available to the life science community. All software discussed in this article is free, with many of the packages being open-source and subject to on-going development.

Competing interests

The authors declare that they have no competing interests.

Authors contributions

AG conducted the microarray experiments. All authors participated in the analysis of the data and in the writing of the manuscript. CSG maintains this document.

Acknowledgements

We wish to thank Dan Swan (Newcastle University Bioinformatics Support Unit) and David Lydall for helpful discussions. The authors are affiliated with the Centre for Integrated Systems Biology of Ageing and Nutrition (CISBAN) at Newcastle University, which is supported jointly by the Biotechnology and Biological Sciences Research Council (BBSRC) and the Engineering and Physical Sciences Research Council (EPSRC).

A. R commands for extracting S. cerevisiae ids, removing unwanted probesets and converting probesets to genes

R function for extracting S. cerevisiae ids

As these microarrays contain probesets for both *S. cerevisiae* and *S. pombe*, we first need to extract the *S. cerevisiae* data before normalisation. This can be done by filtering out the *S. pombe* data using the `s_cerevisiae.msk` file from the Affymetrix website (see [2]). Note that the transcript id, i.e. the systematic orf name (obtained from [3]) will be used for genes with no name.

```
ExtractIDs = function(probe_filter) {
    #probe_filter: a vector of S. cerevisiae genes
    #Get both S. pombe & S. cerevisiae ids from yeast2GENENAME library
    require(yeast2.db)
    genenames = as.list(yeast2GENENAME)
    probes = names(genenames)

    #Get all transcript ids from yeast2annotation.csv
    annotations = read.csv(file='yeast2annotation.csv', header=TRUE,
                           stringsAsFactors=FALSE)
    transcript_id = annotations[ ,3]
    probeset_id = annotations[ ,1]

    #Reorder the transcript_id to match probes
    transcript_id = transcript_id[match(probes, probeset_id)]

    #Retrieve the probeset and transcript ids for S. cerevisiae
    c_probe_id = probes[-match(probe_filter, probes)]
    c_transcript_id = transcript_id[-match(probe_filter, probes)]

    #We need the TranscriptID if the gene name is `NA'
    yeast_genenames = transcript_id
    for(i in seq(along=probeset_id)) {
        gname = genenames[i][[1]]
        if(!is.na(gname))
            yeast_genenames[i] = gname
    }

    #Set the gene name
    c_genename = yeast_genenames[-match(probe_filter, probes)]
    df = data.frame(probe=c_probe_id, transcript=c_transcript_id,
                    genename=c_genename, stringsAsFactors=FALSE)
    return(df)
}
```

R function for removing unwanted probesets

If an Affymetrix microarray chip contains more than one species then it can be useful to focus on a particular species and filter out the unwanted probesets. For example, the `yeast2.db` Affymetrix chip contains both *S. pombe* and *S. cerevisiae* yeast species. To filter out the *S. pombe* probesets, we remove the mappings from the $x-$, $y-$coordinates to the *S. pombe* probesets in the cdf environment. The following function (adapted from [13]) removes the unwanted *S. pombe* instances:

```
RemoveProbes=function(listOutProbeSets, cdfpackagename, probepackagename){
    #listOutProbeSets: Probes sets that are removed.
    #cdfpackagename: The cdf package name.
    #probepackagename: The probe package name.
    require(cdfpackagename, character.only=TRUE)
    require(probepackagename, character.only=TRUE)

    probe.env.orig = get(probepackagename)

    #Remove probesets from the CDF environment
    rm(list=listOutProbeSets, envir=get(cdfpackagename))

    ##Set the PROBE env accordingly
    ## (idea originally from gcrma compute.affinities.R)
    tmp = get('xy2indices', paste('package:', cdfpackagename, sep=''))

    newAB   = new('AffyBatch', cdfName=cleancdf)
    pmIndex =  unlist(indexProbes(newAB, 'pm'))
    subIndex = match(tmp(probe.env.orig$x, probe.env.orig$y,
                         cdf=cdfpackagename), pmIndex)

    iNA = which(is.na(subIndex))

    ##Need to unlock the environment binding to alter the probes
    ipos = grep(probepackagename, search())
    env = as.environment(search()[ipos])

    unlockBinding(probepackagename, env)
    assign(probepackagename, probe.env.orig[-iNA,], env=env)
    lockBinding(probepackagename, env )
}
```

Probeset level to gene level

It is worth noting that RMA yields probeset expression levels. However, there can be several probesets that map to a single gene in an Affymetrix array. If gene level expression is required (instead of probeset expression) the following code will average over probesets for each gene

```
# Function to average the expression of probesets which map to same gene
probeset2genelevel = function(onesample) return(tapply(onesample, factor(c_df$genename),
    mean))
# Average for each column/array
c_gene_data = apply(exprs(yeast.rma), 2, probeset2genelevel)
```

For the *S. cerevisiae* data, the majority of genes only have a single probeset. However, thirty five genes have two or three associated probesets. In this case study, we will study genes at the probeset level.

B. Additional figures

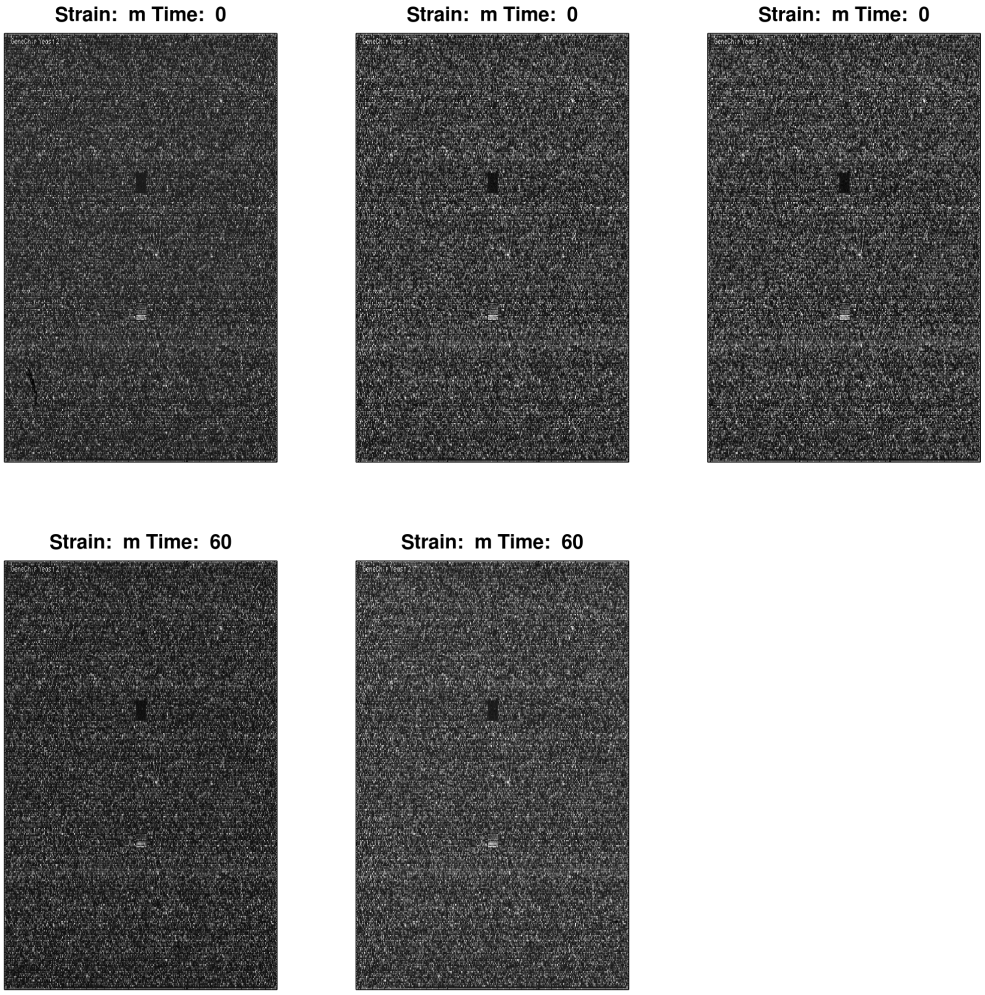

Figure 7: Image plots of the mismatch and perfect match probe intensities for the first replication of the mutant yeast strain. The corresponding times are indicated in the plot.

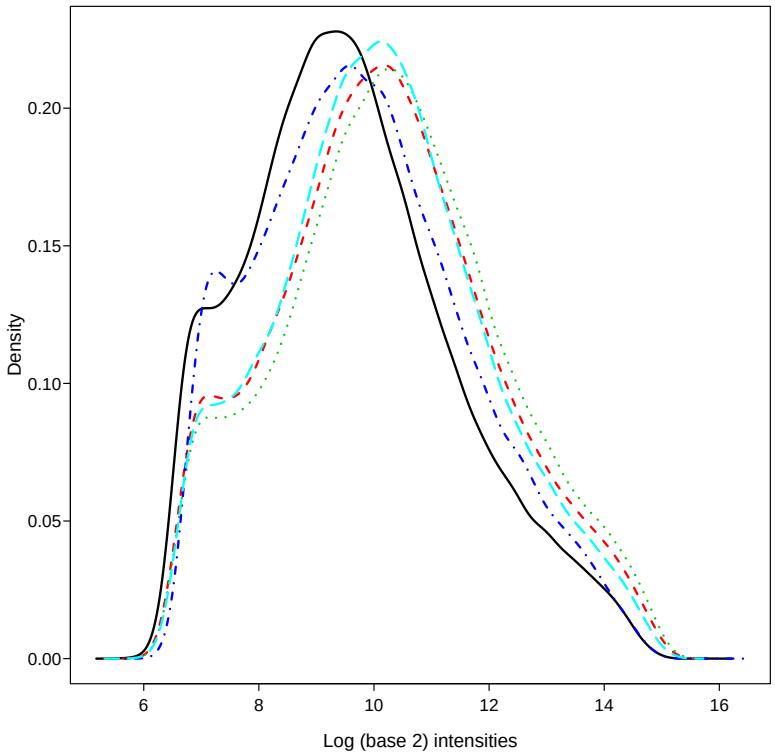

Figure 8: Density plots for the first replication of the mutant yeast strain.

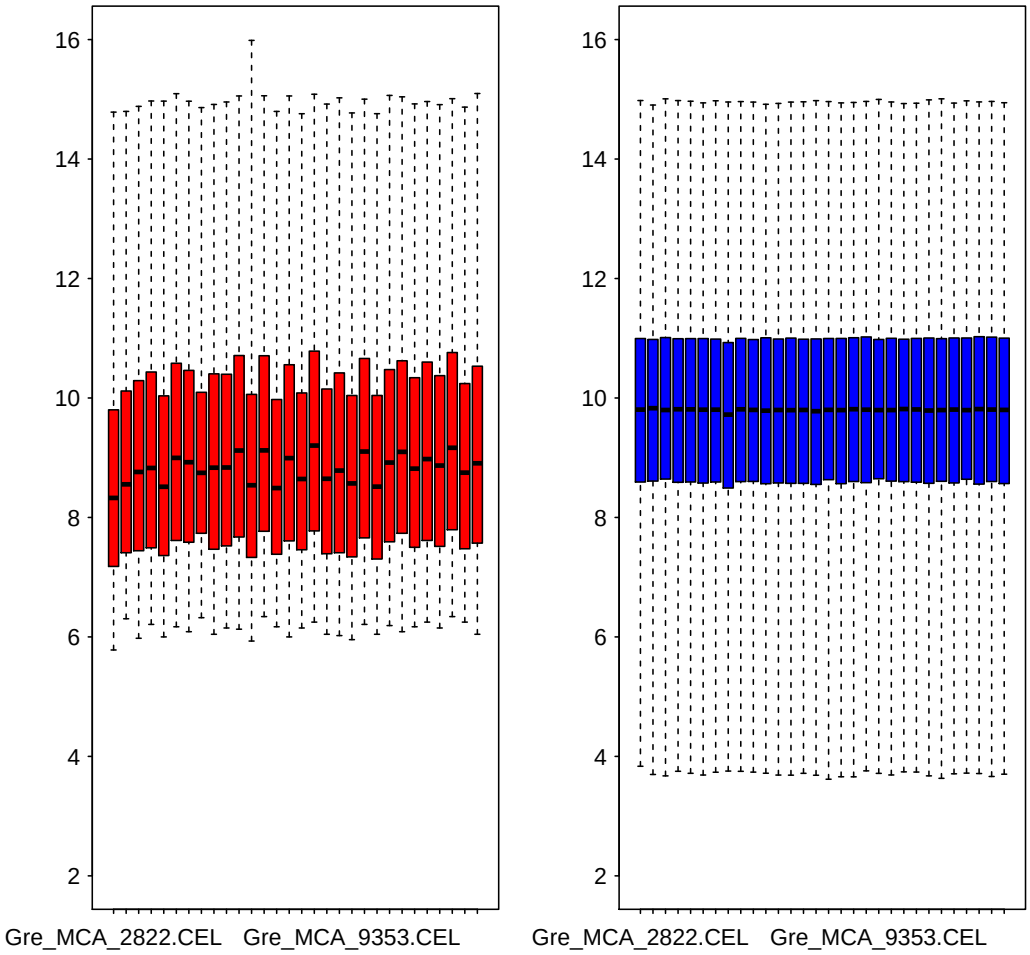

Figure 9: Boxplots of the raw and normalised intensities. The default boxplot is to include both PM and MM intensities, whereas for the density plots in Figure efF2 the default is for only the PM intensities.

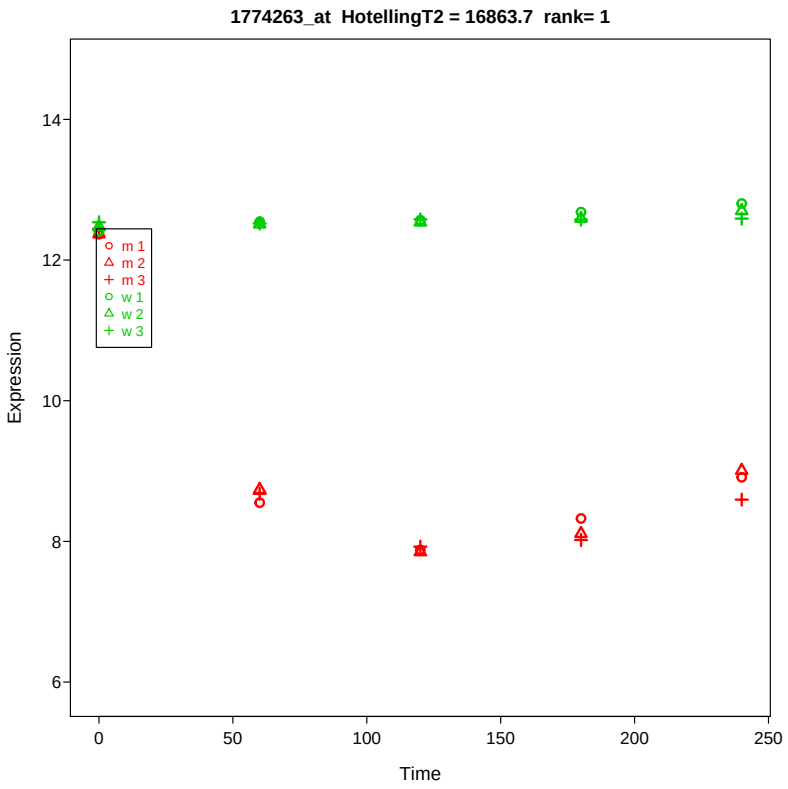

Figure 10: Time course expression levels for the top differentially expressed gene, ranked by their Hotelling statistic using the extttimecourse library.

C. Session Information

```
sessionInfo()

## R version 2.15.0 (2012-03-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
##
## locale:
##  [1] LC_CTYPE=en_GB.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=en_GB.UTF-8
##  [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_GB.UTF-8
##  [7] LC_PAPER=C                 LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] grid      tcltk     stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] ArrayExpress_1.16.0   yeast2.db_2.7.1       org.Sc.sgd.db_2.7.1
##  [4] RSQLite_0.11.2        DBI_0.2-5             yeast2probe_2.10.0
##  [7] yeast2cdf_2.10.0      AnnotationDbi_1.18.1  timecourse_1.28.0
## [10] limma_3.12.1          gplots_2.11.0         MASS_7.3-16
## [13] KernSmooth_2.23-7     caTools_1.13          bitops_1.0-4.1
## [16] gdata_2.12.0          gtools_2.6.2          Mfuzz_2.14.0
## [19] DynDoc_1.34.0         widgetTools_1.34.0    e1071_1.6-1
## [22] class_7.3-3           GeneNet_1.2.5         fdrtool_1.2.10
## [25] longitudinal_1.1.7    corpcor_1.6.4         affyPLM_1.32.0
## [28] preprocessCore_1.18.0 gcrma_2.28.0          BiocInstaller_1.4.7
## [31] affy_1.34.0           Biobase_2.16.0        BiocGenerics_0.2.0
## [34] knitr_0.8
##
## loaded via a namespace (and not attached):
##  [1] affyio_1.24.0     Biostrings_2.24.1 digest_0.5.2
##  [4] evaluate_0.4.2    formatR_0.6       IRanges_1.14.4
##  [7] marray_1.34.0     plyr_1.7.1        splines_2.15.0
## [10] stats4_2.15.0     stringr_0.6.1     tkWidgets_1.34.0
## [13] tools_2.15.0      XML_3.6-2         zlibbioc_1.2.0
```

References

[1] Analysis of Gene Expression Data Short Course, JSM2005.

[2] Pombe and Cerevisiae Filter.

[3] Yeast Annotation File.

[4] Kellie J Archer, Catherine I Dumur, Suresh E Joel, and Viswanathan Ramakrishnan. Assessing quality of hybridized RNA in Affymetrix GeneChip experiments using mixed-effects models. *Biostatistics (Oxford, England)*, 7(2):198–212, 2006.

[5] Bioconductor. Installation instructions.

[6] B M Bolstad, F Collin, J Brettschneider, K Simpson, L Cope, R A Irizarry, and T P Speed. *Bioinformatics and Computational Biology Solutions Using R and Bioconductor*, pages 33–48. Statistics for Biology and Health. Springer-Verlag, New York, 2005.

[7] A Dobra. Sparse graphical models for exploring gene expression data. *Journal of Multivariate Analysis*, 90(1):196–212, 2004.

[8] Matthias Futschik. *Mfuzz: Soft clustering of time series gene expression data*, 2007.

[9] Matthias E Futschik and Bronwyn Carlisle. Noise-robust soft clustering of gene expression time-course data. *Journal of Bioinformatics and Computational Biology*, 3:965–88, August 2005.

[10] Laurent Gautier, Leslie Cope, Benjamin M Bolstad, and Rafael A Irizarry. affy–analysis of Affymetrix GeneChip data at the probe level. *Bioinformatics (Oxford, England)*, 20(3):307–15, 2004.

[11] Robert C Gentleman, Vincent J Carey, Douglas M Bates, Ben Bolstad, Marcel Dettling, Sandrine Dudoit, Byron Ellis, Laurent Gautier, Yongchao Ge, Jeff Gentry, Kurt Hornik, Torsten Hothorn, Wolfgang Huber, Stefano Iacus, Rafael Irizarry, Friedrich Leisch, Cheng Li, Martin Maechler, Anthony J Rossini, Gunther Sawitzki, Colin Smith, Gordon Smyth, Luke Tierney, Jean Y H Yang, and Jianhua Zhang. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology*, 5:R80, 2004.

[12] Amanda Greenall, Guiyuan Lei, Daniel C Swan, Katherine James, Liming Wang, Heiko Peters, Anil Wipat, Darren J Wilkinson, and David Lydall. A genome wide analysis of the response to uncapped telomeres in budding yeast reveals a novel role for the NAD+ biosynthetic gene BNA2 in chromosome end protection. *Genome Biology*, 9:R146, 2008.

[13] W Gregory Alvord, Jean A Roayaei, Octavio A Quiñones, and Katherine T Schneider. A microarray analysis for differential gene expression in the soybean genome using Bioconductor and R. *Briefings in Bioinformatics*, 8:415–31, 2007.

[14] Bettina Harr and Christian Schlötterer. Comparison of algorithms for the analysis of Affymetrix microarray data as evaluated by co-expression of genes in known operons. *Nucleic Acids Research*, 34:e8, January 2006.

[15] Dirk Husmeier. Sensitivity and specificity of inferring genetic regulatory interactions from microarray experiments with dynamic Bayesian networks. *Bioinformatics (Oxford, England)*, 19(17):2271–82, November 2003.

[16] Rafael A Irizarry, Bridget Hobbs, Francois Collin, Yasmin D Beazer-Barclay, Kristen J Antonellis, Uwe Scherf, and Terence P Speed. Exploration, normalization, and summaries of high density oligonucleotide array probe level data. *Biostatistics (Oxford, England)*, 4:249–64, 2003.

[17] Aurelie Labbe, Marie-Paule Roth, Pierre-Hugues Carmichael, and Maria Martinez. Impact of gene expression data pre-processing on expression quantitative trait locus mapping. *BMC Proceedings*, 1 Suppl 1(Suppl 1):S153, January 2007.

[18] Insuk Lee, Shailesh V Date, Alex T Adai, and Edward M Marcotte. A probabilistic functional network of yeast genes. *Science (New York, N.Y.)*, 306:1555–8, 2004.

[19] R Opgen-Rhein and K Strimmer. Using regularized dynamic correlation to infer gene dependency networks from time-series microarray data. In *Proceedings of the 4th International Workshop on Computational Systems Biology*, pages 73–76, 2006.

[20] Rainer Opgen-Rhein and Korbinian Strimmer. From correlation to causation networks: a simple approximate learning algorithm and its application to high-dimensional plant gene expression data. *BMC Systems Biology*, 1:37, 2007.

[21] Helen Parkinson, Misha Kapushesky, Nikolay Kolesnikov, Gabriella Rustici, Mohammad Shojatalab, Niran Abeygunawardena, Hugo Berube, Miroslaw Dylag, Ibrahim Emam, Anna Farne, Ele Holloway, Margus Lukk, James Malone, Roby Mani, Ekaterina Pilicheva, Tim F Rayner, Faisal Rezwan, Anjan Sharma, Eleanor Williams, Xiangqun Zheng Bradley, Tomasz Adamusiak, Marco Brandizi, Tony Burdett, Richard Coulson, Maria Krestyaninova, Pavel Kurnosov, Eamonn Maguire, Sudeshna Guha Neogi, Philippe Rocca-Serra, Susanna-Assunta Sansone, Nataliya Sklyar, Mengyao Zhao, Ugis Sarkans, and Alvis Brazma. ArrayExpress update–from an archive of functional genomics experiments to the atlas of gene expression. *Nucleic Acids Research*, 37:D868–72, 2009.

[22] R Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2009.

[23] Juliane Schäfer and Korbinian Strimmer. An empirical Bayes approach to inferring large-scale gene association networks. *Bioinformatics (Oxford, England)*, 21:754–64, 2005.

[24] Gordon K Smyth. Linear models and empirical Bayes methods for assessing differential expression in microarray experiments. *Statistical Applications in Genetics and Molecular Biology*, 3:Article3, 2004.

[25] Gordon K Smyth. Limma: linear models for microarray data. In R Gentleman, V Carey, S Dudoit, and W Huber R. Irizarry, editors, *Bioinformatics and Computational Biology Solutions using R and Bioconductor*, pages 397–420. Springer, New York, 2005.

[26] Yu Chuan Tai. *timecourse: Statistical Analysis for Developmental Microarray Time Course Data*, 2007.

[27] Yu Chuan Tai and Terence P. Speed. A multivariate empirical Bayes statistic for replicated microarray time course data. *The Annals of Statistics*, 34:2387–2412, 2006.

[28] Z Wu, R A Irizarry, R Gentleman, R Martinez-Murillo, and R Spencer. A Model Based Background Adjustment for Oligonucleotide Expression Arrays. *Journal of the American Statistical Association*, 99:909–917, 2004.

[29] Jing Yu, V Anne Smith, Paul P Wang, Alexander J Hartemink, and Erich D Jarvis. Advances to Bayesian network inference for generating causal networks from observational biological data. *Bioinformatics (Oxford, England)*, 20(18):3594–603, 2004.