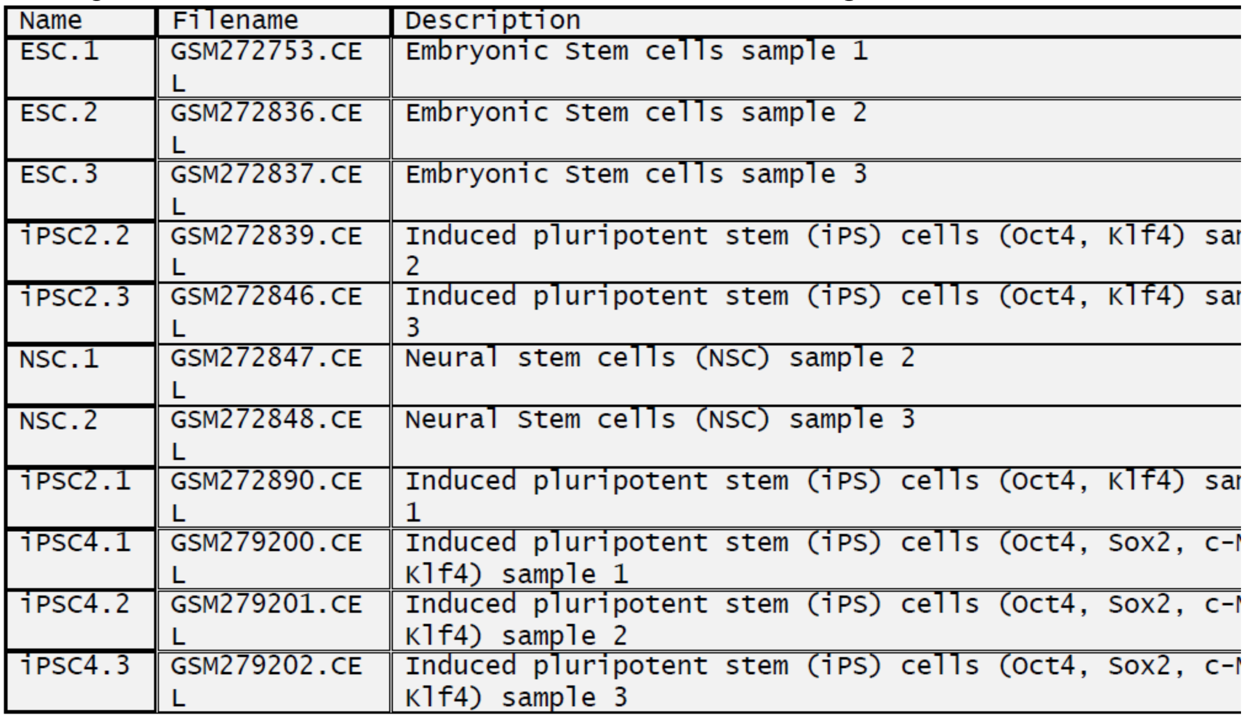
FGT: Analysing Affymetrix Single-Channel Gene Expression Arrays in R

Data input

Here we use the `affy` package to load data. The `affy` package has one major class: `AffyBatch`. The `AffyBatch` class is a representation of the Affymetrix GeneChip probe level data and extends the container class `eSet`. To create an `AffyBatch` object which contains the expression values of all the CEL files in your working directory you can use the wrapper function `ReadAffy()`. In this tutorial we will use pre-saved example data from the experiment (GSE10806: http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE10806). Note that you can browse a collection of available datasets in the Gene Expression Omnibus (GEO, http://www.ncbi.nlm.nih.gov/geo/).

To begin the tutorial log into your server account, create a directory for the tutorial. You do not need to download this data, it is provided on the shared folder. Copy the data from ***`/home/shared_files/FGT_T3_GSE10806_RAW.tar`*** ***and unpack the archive into a new folder for this tutorial.*** **A targets file that describes the samples can also be copied provided from `/home/shared_files/FGT_T3_targets.txt`**

The *targets,txt* file contains the first two columns of the following information



Name	Filename	Description
ESC.1	GSM272753.CEL	Embryonic Stem cells sample 1
ESC.2	GSM272836.CEL	Embryonic Stem cells sample 2
ESC.3	GSM272837.CEL	Embryonic Stem cells sample 3
iPSC2.2	GSM272839.CEL	Induced pluripotent stem (iPS) cells (Oct4, Klf4) sa 2
iPSC2.3	GSM272846.CEL	Induced pluripotent stem (iPS) cells (Oct4, Klf4) sa 3
NSC.1	GSM272847.CEL	Neural stem cells (NSC) sample 2
NSC.2	GSM272848.CEL	Neural Stem cells (NSC) sample 3
iPSC2.1	GSM272890.CEL	Induced pluripotent stem (iPS) cells (Oct4, Klf4) sa 1
iPSC4.1	GSM279200.CEL	Induced pluripotent stem (iPS) cells (Oct4, Sox2, c- Klf4) sample 1
iPSC4.2	GSM279201.CEL	Induced pluripotent stem (iPS) cells (Oct4, Sox2, c- Klf4) sample 2
iPSC4.3	GSM279202.CEL	Induced pluripotent stem (iPS) cells (Oct4, Sox2, c- Klf4) sample 3

We are now ready to load the data- note that use of the targets file is optional here, but becomes more essential as the number of samples increases.

Overview of affy: Methods for Affymetrix Oligonucleotide Arrays

The `affy` package contains methods for the processing of oligonucleotide arrays and has been a part of the Bioconductor suite since its first release in 2002. The `affy` package provides a range of statistical analysis methods that cover all aspects of the analysis pipeline including data input, quality control, data normalization and plotting. For a detailed documentation of the `affy` package, please check the Reference Manual available on the Bioconductor website (http://www.bioconductor.org/packages/release/bioc/manuals/affy/man/affy.pdf). In order to install the `affy` package, you need to install R and some related Bioconductor packages. There is no need to download files, as the required packages have been already installed in the R environment of this tutorial. For this tutorial the examples should be run in sequential order. Data outputs generated in earlier sections are assumed to be present for later section examples. We use `affy` here but note that some expression arrays from Affymetrix cannot use `affy`- consider `oligo` instead.

From the current working directory, start R.

```
# Commands to download and install the repackage
# These packages are already installed on the class computer
#Copy this code and uncomment the three commands below to
#install the required files on another machine
#
#if (!requireNamespace("BiocManager", quietly = TRUE))
#    install.packages("BiocManager")
# BiocManager::install("affy")
# BiocManager::install("limma")

#load the required libraries for the tutorials...

library(affy)
library(limma)
```

```
# Load the target file into an AnnotatedDataFrame object
adf<-
read.AnnotatedDataFrame("FGT_T3_targets.txt",header=TRUE,row.names=1,as.is=
TRUE)
# Load the expression values of all the CEL files in the targets file
#mydata <- ReadAffy(filenames=pData(adf)$FileName,phenoData=adf)

# Or just to quickly load all CEL files in the R working directory
mydata <- ReadAffy()
# View a summary of the example data
mydata
```

In the above block of code we load a targets file, although this is optional. We then load the CEL data using `ReadAffy()`. Finally, we check the loaded data by listing the contents of the `mydata` AffyBatch object.

Quality control of raw data

```
# Quality control plots
hist(mydata)

# And a boxplot with different colour per sample group
colours <- c(rep("yellow",3),rep("red",2),rep("blue",2),
"red", rep("green",3))

boxplot(mydata, col=colours, las=2)
```

Remember when generating plots you can use save the plot as a png image using `png(<afilename>)`, followed by the command to generate the plot, followed by `dev.off()`. You can also exit R and issue from the Unix command `line- display <afilename> or download and view the image.` Of course `<afilename>` is the name of the output png file you have generated and would like to view. Of course you can also view these files from within R by issuing a `system` command eg `system("display myfilename.png")`

Here we create a simple histogram using `hist` and also coloured boxplot using `boxplot`. Note the colours object is an array of strings representing colours, the `rep command` is used to specify repeats of each string. These colours in the array have to be in the same order as the samples in the AffyBatch object.

Data normalisation

The normalisation of microarray data is a necessary step in order to account for variation that arises from the way the samples have been generated in the lab. There are numerous normalisation methods available that can work with Affymetrix data i.e. MAS5 , RMA, gcRMA and VSN among others. Here we will apply the Robust Multichip Averaging (RMA) method which is the most commonly used approach for this type of data. The RMA method includes three basic steps: an array-specific background adjustment, quantile normalization to ensure that the distribution of the expression values of each array in the comparison is identical and, finally, summarisation to generate the normalised measurement for each probe. Note that the normalised values returned by RMA are log2-transformed although this is not always the case- the `mas5` function generates non-log transformed data.

```
# Normalise the data using RMA
eset <- rma(mydata)
eset
# To obtain a matrix of the expression values, use exprs()
values <- exprs(eset)
# Boxplot to observe the results of normalisation
# Notice differences with the boxplot from the raw data
boxplot(values, col=colours, las=2)
```

Quality control after normalisation

One way to examine dependencies between the log-ratio of two variables and their mean intensity level is the MA plot. In the MA plot, the log-ratio (M) is plotted on the vertical axis and is defined as M = log2 Xi - log2 Xj while the mean intensity (A) is plotted on the horizontal axis and is defined as A = 1/2 (log2 Xi +log2 Xj), where Xi, Xj the intensity values of samples i and j. The general assumption in a microarray experiment is that the majority of the genes do not change between the two samples. So we can expect that the majority of the points in the MA plot are around 0.
Plotting the MA plots for all possible combinations of samples in the experiment takes some time. In the interest of brevity, we demonstrate how to examine samples 1 and 4. You can plot different comparisons in a similar way.

```
# MA plot of the samples 1 and 4
mva.pairs(values[, c(1,4)])
# The same plot for the non-normalised raw data
# Note that the mva.pairs call below only plots a few of the
#samples - you may wish to plot them all but this is slow
mva.pairs(pm(mydata)[, c(1,4)])
```

Note that in this code the `pm()` command is used to plot only the perfect match oligonucleotides. The index used for the `values` array is a vector with two sample indexes- sample 1 and sample 4. Of course, if no sample index is provided all samples are plotted in the panel plot generated by `mva.pairs`.

Hierarchical clustering of normalised data

Hierarchical clustering finds the pair of features that are most similar and joins them together as a node in a tree. Subsequently, the algorithm finds the next most similar pair of features. It iterates in a similar way across the whole set of features resulting to a dendrogram representation of the relationship between features. In a microarray experiment, features can be either the genes or the samples. In this way, we can examine the similarities between different features. In the following examples, we will perform hierarchical clustering of the samples based on the Pearson correlation coefficient.

```
# To facilitate interpretation, let’s replace the columns
# header,currently
# displaying the filename, to show the name of each sample
# (if you have a targets file)
colnames(values) <- rownames(pData(adf))
# Performs hierarchical clustering with average linkage based on
# Pearson’s Correlation Coefficient
hc<-hclust(as.dist(1-cor(values, method="pearson")), method="average")
plot(hc)
```

Principal Components Analysis (PCA) of normalised data

For a detailed tutorial on PCA and the mathematics behind it, please have a look here: http://www.sccg.sk/~haladova/principal_components.pdf. Briefly, PCA uses the observed features in your dataset to identify artificial features that can account for most of the variability in the observed features. Basically, it can be used to reduce redundancy in our feature set in cases where we expect that there exist features that don't convey additional information because i.e. they are correlated with other features. We can then plot our data using these artificial features, the principal components, which represent a linear combination of weighted features, to identify patterns in the data and highlight similarities and differences. Here, we will perform PCA to the samples of the experiment and plot it using the scatterplot3d package which will represent the three first principal components in a 3D plot.

```
#here we request a specific package from a specific archive...
#install.packages("scatterplot3d",repo="http://cran.ma.imperia
#l.ac.uk")

#Then load the library
library(scatterplot3d)
# Perform PCA
pca <- prcomp(t(values), scale=T)
# Plot the PCA results

s3d<-scatterplot3d(pca$x[,1:3], pch=19, color=rainbow(1))
s3d.coords <- s3d$xyz.convert(pca$x[,1:3])
text(s3d.coords$x, s3d.coords$y, labels = colnames(values),pos
= 3,offset = 0.5)
```

Notice that the data matrix, called `values`, is used as input to the `t()` function, this transposes the matrix so that rows become columns and columns become rows.

Fold Filtering

Obtaining Expression Values

To obtain a matrix of the expression values, use the `exprs()` function:

```
#obtaining a matrix of expression values
exprsvals <- exprs(eset)
#RMA outputs log2 data while MAS5 outputs linear data
#To convert from log…
exprsvals10 <-2^exprsvals
#check conversion
exprsvals[1:100,]
#converted
exprsvals10[1:100,]
```

Check Sample Name Order

For vectors of the names of the samples (essential for checking the order of the samples are read in) and to obtain the probe IDs of the genes use the functions `sampleNames()` and `probeNames()` respectively.

```
#check order of sample names
mysamples <- sampleNames(eset)
#display the list
mysamples
#it is useful to obtain a vector of ProbeIDs here
probesets <- probeNames(mydata)
#display the first 100 ProbeSets
probesets[1:100]
```

Build a Fold Change Vector

Here we output the final summary data containing the expression means of each replicate group and the fold changes. We first use the apply method to apply the mean function to the selection of columns from each replicate group. We then build up each mean list. The `cbind` function is then used to add all the columns together into a table in order that the summary data can be conveniently output. Note that here the mean is calculated from the non-log transformed values. Means are also calculated from average log values- in this case this is equivalent to the geometric mean on the non-transformed scale. Logs can be of any base- conventionally base 2 is used as one unit of change eg +1 or -1 is equal to 2 fold change.

```
#Calculate the means
#Note mean of the log is not the same as the log of the mean!!
ES.mean <- apply(exprsvals10[,c("GSM272753.CEL",
"GSM272836.CEL","GSM272837.CEL")],1,mean)
iPS_OK.mean <- apply(exprsvals10[,c("GSM272839.CEL",
"GSM272846.CEL","GSM272890.CEL")],1,mean)
iPS_4F.mean <- apply(exprsvals10[,c("GSM279200.CEL",
"GSM279201.CEL","GSM279202.CEL")],1,mean)
NSC.mean <-
apply(exprsvals10[,c("GSM272847.CEL","GSM272848.CEL")],1,mean)
#calculate some fold changes
ES_iPS_OK <-ES.mean /iPS_OK.mean
ES_iPS_4F <-ES.mean /iPS_4F.mean
ES_NSC <-ES.mean /NSC.mean
#build a summary table to hold all the data
all.data= cbind(ES.mean,iPS_OK.mean,iPS_4F.mean, NSC.mean,
ES_iPS_OK,
ES_iPS_4F, ES_NSC)
#check the column names
colnames(all.data)
#write the table of means as an output
write.table(all.data,file="group_means.txt", quote=F,
sep="\t",col.names=NA)
```